Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2012-018 June 29, 2012

Integrated Robot Task and Motion Planning in the Now

Leslie Pack Kaelbling and Tomas Lozano-Perez

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Integrated robot task and motion planning in the now

Leslie Pack Kaelbling and Tomás Lozano-Pérez
CSAIL, MIT, Cambridge, MA 02139
{lpk, tlp}@csail.mit.edu

Abstract

This paper provides an approach to integrating geometric motion planning with logical task planning for long-horizon tasks in domains with many objects. We propose a tight integration between the logical and geometric aspects of planning. We use a logical representation which includes entities that refer to poses, grasps, paths and regions, without the need for *a priori* discretization. Given this representation and some simple mechanisms for geometric inference, we characterize the pre-conditions and effects of robot actions in terms of these logical entities. We then reason about the interaction of the geometric and non-geometric aspects of our domains using the general-purpose mechanism of goal regression (also known as pre-image backchaining). We propose an aggressive mechanism for temporal hierarchical decomposition, which postpones the pre-conditions of actions to create an abstraction hierarchy that both limits the lengths of plans that need to be generated and limits the set of objects relevant to each plan.

We describe an implementation of this planning method and demonstrate it in a simulated kitchen environment in which it solves problems that require approximately 100 individual pick or place operations for moving multiple objects in a complex domain.

1 Introduction

Robots act to achieve goals in the world: moving the paint sprayer to paint a car, opening a door to let someone in, moving the plates to empty the dishwasher, and so on. Consider a robot operating in a relatively simple home environment: there are a few rooms, some food items in the kitchen refrigerator, dishes and pans on the shelves, some tables and counter surfaces, a stove with two burners (in red) and a sink (in blue). A simulation of such an environment is shown in figure 1. The robot has the goal of serving food items on the kitchen table and then tidying up. Even ignoring the intricate manipulation required to open jars, cut food, actually cook the items or clean the dishes, achieving these simple goals requires a long sequence of actions: bringing the food items to the preparation area, getting the pans, bringing the food to the pan, cooking the items, serving them on plates and then cleaning up (putting everything back in the refrigerator, washing the pans and even picking up the dirty cup someone left in the living room). As part of this process, the robot must perform actions that were not explicitly mentioned in the goals; for example, moving items out of the refrigerator to reach the desired food items and then replacing them. Throughout, the robot must choose how to grasp the objects, where to stand so as to pick up and put down objects, where to place the objects in the regions, what paths to use for each motion, and so on. The question we address in this paper is: How should we structure the computational process that controls the robot so that it can effectively achieve its goals?

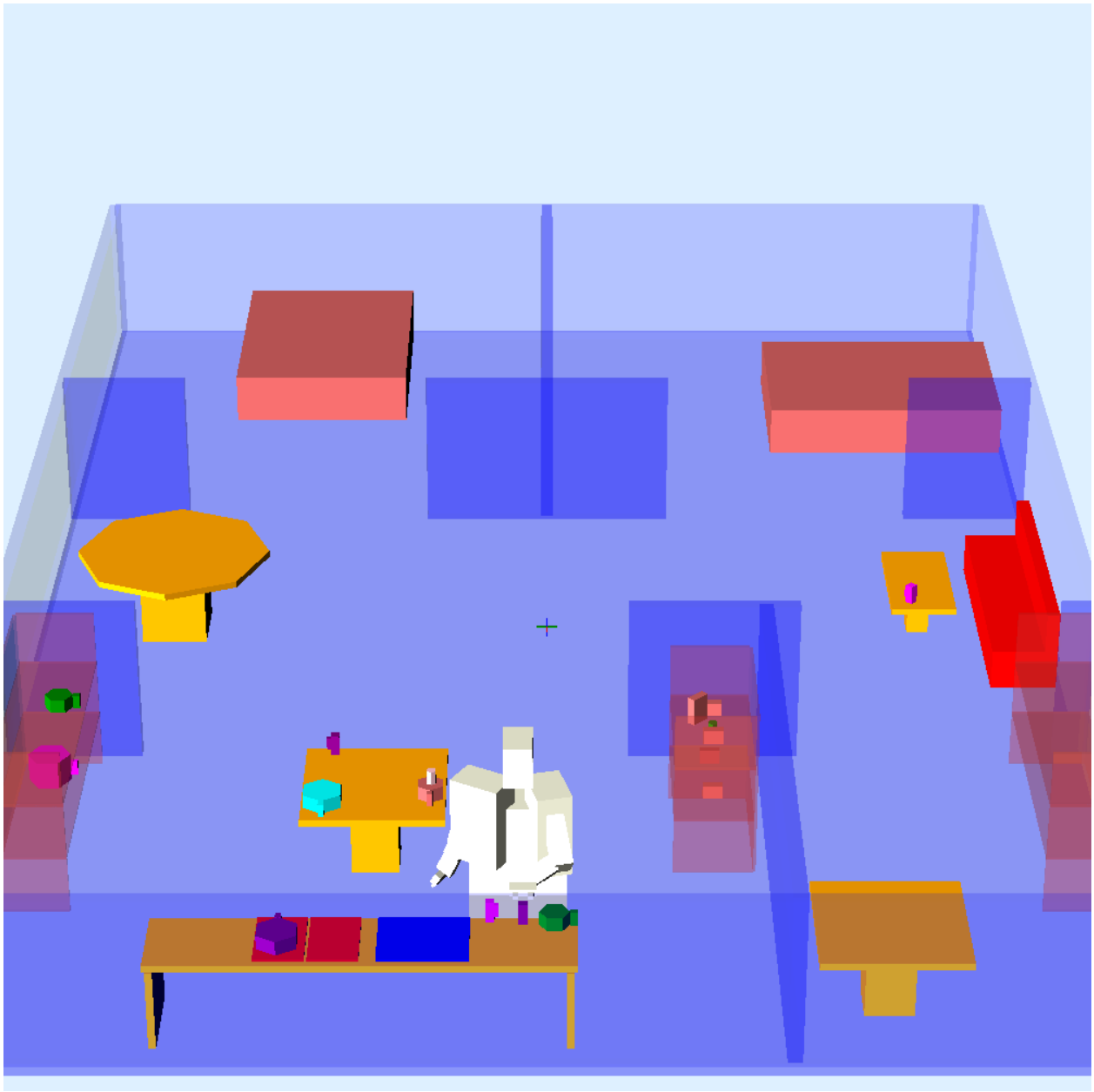


Figure 1: Simulated apartment environment with pans, cups, sink, stove, and furniture.

We propose a strategy for combining symbolic and geometric planning, first introduced by Kaelbling and Lozano-Pérez [2011a], that enables the solution of very large mobile manipulation problems. It sacrifices optimality as well as the ability to solve some highly intricate problems in order to be able to address many problem instances that are too large to solve by other means presently available. The solution concept could be applied to many other large domains, but here we focus on, and tailor the system to, the problem of mobile manipulation.

One strategy for designing such goal-achieving robots is to engineer, compute offline or learn a

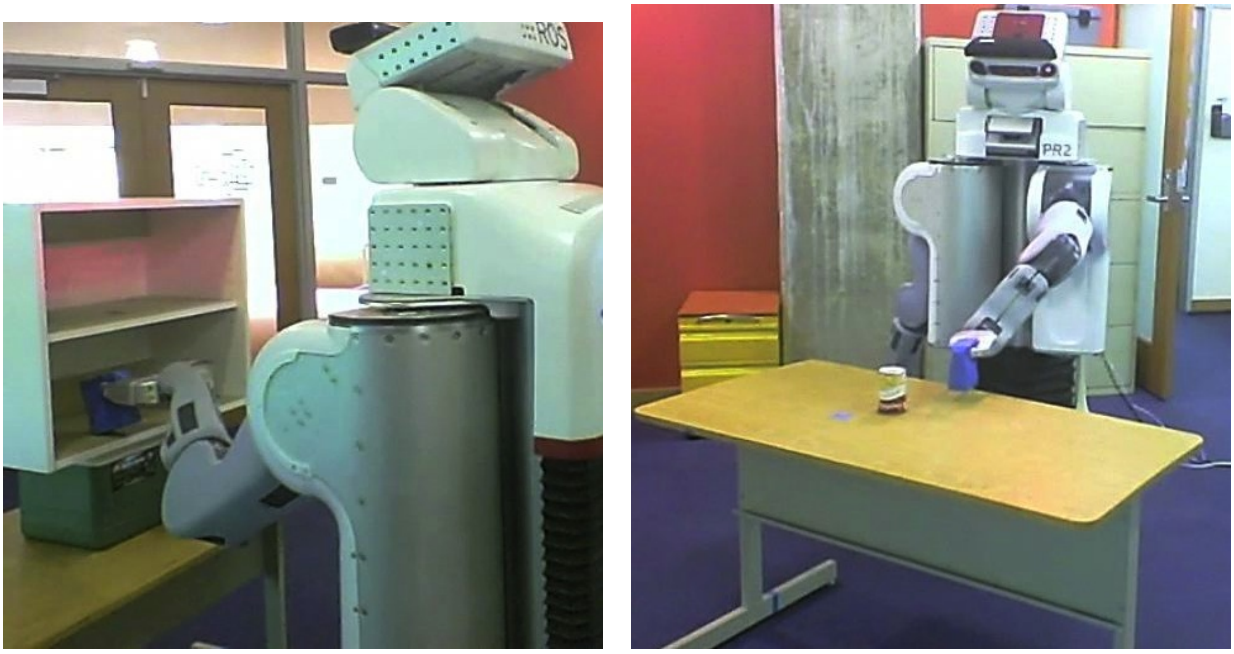


Figure 2: PR2 robot manipulating objects

policy that directly encodes a mapping from world states or beliefs over world states into actions that ultimately achieve the goals. Such policies can vary in complexity, from a fixed sequence of actions independent of the world state, to a feedback loop based on one or more sensed values, to a complex state-machine. Although they can be derived from the engineer's knowledge of the world dynamics and the goals, they would not explicitly represent or reason over the goals or world states. This approach is critical for domains with significant time pressure and can be very successful for tasks with limited scope, such as spray painting cars and vacuuming carpets or controlling the robot while cutting vegetables. However, we expect that, for sufficiently complex tasks such as building a complete robot household assistant, where the action sequences have intricate dependencies on the initial state of the world, on extrinsic events, and on the outcomes of the robot's actions, explicitly represented policies would need to be intractably large.

We pursue an alternative approach to designing robots for complex domains in which the robot chooses its actions based on explicit reasoning over goals and representations of world states. Note that the goals and states must refer to aspects of the world beyond the robot itself, for example, to the people in the household or the food in the refrigerator. Classical artificial intelligence planning [Ghallab et al., 2004] (often known in robotics as task planning) provides a paradigm for reasoning about fairly general goals and states to derive sequences of "abstract" actions. However, we must ultimately choose particular low-level robot motions, and classical task planning methods are not able to address the geometric and kinematic challenges of robot motion planning in high-dimensional continuous spaces.

The original conception, in Shakey [Nilsson, 1984], of the interaction between a task planner and a motion planner was that the task planner would construct "high-level" plans based on some weak geometric information (such as connectivity between rooms) and the steps of these plans would be achieved in the world by "intermediate-level" primitives, which would involve perception and

motion planning, and ultimately issue "low-level" controls. This conception has generally proved unworkable; in particular, a much tighter integration between task planning, geometric planning and execution is generally necessary [Cambon et al., 2009]. The need for tight integration of levels is even more pronounced in the presence of uncertainty and failure [Haigh and Veloso, 1998, Burgard et al., 1998]. An especially challenging case is when "high-level" actions may be motivated primarily by the need to acquire information about object placements.

It is clear that non-geometric goals affect geometric goals: for example, we need to properly place the pan on the stove to cook the food. But the converse is also true: the geometry can affect the structure of the non-geometric plan. For example, when planning to put away objects given some possible containers, the size and locations of the containers affect which object(s) can be placed in which containers. Therefore, we need an approach to planning that allows the non-geometric and geometric considerations to interact closely.

This paper describes a hierarchical approach to integrating task and motion planning that can scale to robot tasks requiring long sequences of actions in domains involving many objects, such as the one in figure 1. However, both task and motion planning are computationally intractable problems; we cannot hope to solve hard puzzles efficiently. Our method exploits structural decompositions to give large improvements in computation time, in general, at some cost in suboptimality of execution, but without compromising completeness in many domains.

There is substantial uncertainty in any real mobile manipulation problem, such as the PR2 robot, shown in figure 2, that matches our simulated robot. This uncertainty stems both from the typical sources of sensor and motion error, but also from fundamental occlusions and uncertainties: how is the robot to know what is in a particular kitchen drawer? Handling uncertainty is a critical problem. In this paper, we establish a planning and execution framework that will serve as a foundation for an approach that deals with deeper issues in non-determinism and partial observability [Kaelbling and Lozano-Pérez, 2011b, 2012].

1.1 Approach

The most successful robot motion planning methods to date have been based on searching from an initial geometric configuration using a sample-based method that is either bidirectional or guided toward the goal configuration. As the dimensionality of the problem increases (due to multiple movable objects) and the time horizon increases (due to the number of primitive actions that must be taken) direct search in the configuration space becomes very difficult. Ultimately, it cannot scale to problems of the size we are contemplating, nor does it provide a method for integrating non-geometric goals and constraints into the problem.

We propose a hierarchical approach to solving long-horizon problems, which performs a temporal decomposition by planning operations at multiple levels of abstraction. By performing a hierarchical decomposition, we can ensure that the problems to be addressed by the planner always have a reasonably short horizon, making planning feasible.

In order to plan with abstract operators, we must be able to characterize their preconditions and effects at various levels of abstraction. For reasons described above, even at abstract levels, geometric properties of the domain may be critical for planning; but if we plan using abstracted models of the operations, we will not be able to determine a detailed geometric configuration that results from performing an operation. To support our hierarchical planning strategy we, instead, plan backwards from the goal using the process known as *goal regression* [Ghallab et al., 2004] in task planning and *pre-image backchaining* in motion planning [Lozano-Pérez et al., 1984]. Starting

from the set of states that satisfies the goal condition, we work backward, constructing descriptions of pre-images of the goal under various abstract operations. The pre-image is the set of states such that, if the operation were executed, a goal state would result. The key observation is that, whereas the description of the detailed world state is an enormously complicated structure, the descriptions of the goal set, and of pre-images of the goal set, are often describable as simple conjunctions of a few logical requirements.

In a continuous space, pre-images might be characterized geometrically: if the goal is a circle of locations in x, y space, then the operation of moving one meter in x will have a pre-image that is also a circle of locations, but with the x coordinate displaced by a meter. In a logical, or combined logical and geometric space, the definition of pre-image is the same, but computing pre-images will require a combination of logical and geometric reasoning. We support abstract geometric reasoning by constructing and referring to salient geometric objects in the logical representation used by the planner. So, for example, we can say that a region of space must be clear of obstacles before an object can be placed in its goal location, without specifying a particular geometric arrangement of the obstacles in the domain. This approach allows a tight interplay between geometric and non-geometric aspects of the domain.

The complexity of planning depends both on the horizon and the branching factor. We use hierarchy to reduce the horizon, but the branching factor, in general, remains infinite: there are innumerable places to put an object, innumerable paths for the robot to follow from one configuration to another, and so on. We introduce the notion of *generators* that make use both of constraints from the goal and heuristic information from the starting state to make choices from these infinite domains that are likely to be successful; our approach can be extended to support sample-based backtracking over these choices if they fail. Because the value-generation process can depend on the goal and the initial state, the values are much more likely to be successful than ones chosen through an arbitrary sampling or discretization process.

Even when planning with a deterministic model of the effects of actions, we must acknowledge that there may be errors in execution. Such errors may render the successful execution of a very long sequence of actions highly unlikely. For this reason, we interleave planning with execution, so that all planning problems have short horizons and start from a current, known initial state. If an action has an unexpected effect, a new plan can be constructed at not too great a cost, and execution resumed. We refer to this overall approach as HPN for "hierarchical planning in the now."

In the rest of this section, we provide a more detailed overview of the HPN approach. We use a simple, one-dimensional environment to illustrate the key points.

1.1.1 Representing objects, relations and actions

Figure 3 shows three configurations of an environment in which there are two blocks, a and b, on a table. They are each 0.5 units wide, and their position is described by the (continuous) position of their left edges. The black line in the figure represents the table and the boxes beneath the line represent regions of one-dimensional space that are relevant to the planning process. The configuration space, then, consists of points in $\mathcal{R}^2$, with one dimension representing the left edge of a and the other representing the left edge of b. However, some (l_a, l_b) pairs represent configurations in which the blocks are in collision, so the free configuration space is $\mathcal{R}^2$ with the illegal locus of points $\{(l_a, l_b) \mid l_a - l_b < 0.5\}$, which is a stripe along the diagonal, removed.

At the lowest level of abstraction, the domain is represented in complete detail, including shapes and locations of all objects and the full configuration of the robot. This representation is used to

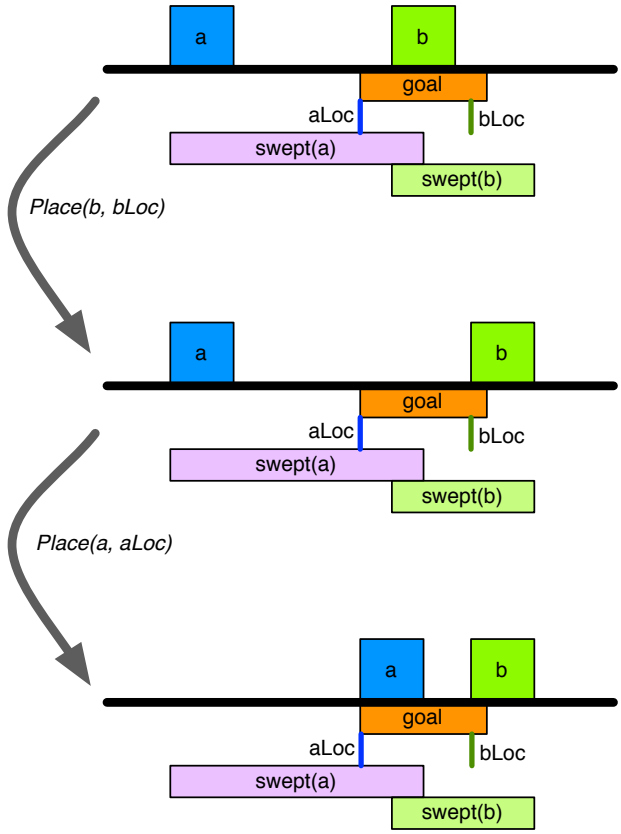


Figure 3: Simple one-dimensional environment: starting configuration, intermediate configuration after moving b out of the way, final configuration with a in the goal region.

support detailed motion planning.

At higher levels of abstraction, we use logical representations to characterize both geometric and non-geometric aspects of the world state. To describe our example domain, we use instances of the following logical assertions:

- $In(o, r)$ indicates that object o is completely inside the region r;
- $ObjLoc(o, l)$ indicates that the left edge of object o is at location l; and,
- $ClearX(r, x)$ indicates that no objects overlap region r, except possibly for objects named in the set x.

Note that by "logical" representations, we mean representations that are made up of explicit references to entities in the domain (including objects, regions, paths, and poses), relations among these entities (including object *in* region, or pose *near* object) that have truth values in a given state, and Boolean combinations of these relations. We emphatically do not mean that these representations are free of numbers, as the examples above attest. The key benefit of logical representations is that they give us compact representations of large, possibly infinite, sets of world states and of the dynamic effects of actions on such large or infinite domains. Our key requirement on the representations will be that, given a logical description, we must be able to determine

whether a geometric world state satisfies that description. For example, we must be able to test whether $In(A, goal)$ is true of a world state.

To complete a domain representation, we must also describe the available actions. We will use *operators* which are loosely modeled on the operators of classical task planning. An operator describes the *preconditions* of an action and its *effects*. In our simple environment, we have three types of operators, each of which has as its effect one of the types of assertions mentioned above. The first one actually causes a physical effect in the environment; the other two are essentially inference rules that reduce one logical condition to another.

- PLACE(o, l_{target}) places object o at a target location l_{target}; the effect is $ObjLoc(o, l_{target})$. The preconditions are that the object be in some location, l_{start}, before the operation takes place and that the region "swept" by the object moving from l_{start} to l_{target} be clear except for o, that is, $ClearX(sweptVol(o, l_{start}, l_{target}), \{o\})$.

- IN(o, r): the effect is $In(o, r)$ if $ObjLoc(o, l)$ is true for some location, such that the volume of object o when located at l is contained in region r.

- CLEAR(r, x): the effect is $ClearX(r, x)$ if all objects in the universe except those in the set x do not overlap region r; that is, that for all objects o not in x, $In(o, \bar{r})$, where $\bar{r}$ is the spatial complement of r (that is, the spatial domain minus r).

1.1.2 Regression and generators

In HPN, the goal of the system is to reach some element in a set of states of the world, described using a logical expression. So, for example, a goal might be for object a to be in the region *goal*, which would be written $In(a, goal)$. The planning algorithm proceeds by applying a formal description of the operations in the domain to find one that might achieve the goal condition, then adopting the preconditions of that operation as a new sub-goal. This process of goal-regression proceeds until all of the conditions in the subgoal are true in the current world state.

To illustrate regression and generation in HPN, we work through a very simple example in detail. The goal of our simple planning problem is for block a to be in the goal region, which is indicated in orange. There is no explicit robot in this environment: to keep the space low-dimensional, we will simply consider motions of the objects along the one-dimensional table.

Figure 4 shows a path in the regression search tree through sets of configurations of the blocks. Recall that the configuration space in this domain is the positions of the left edges of the two blocks, which is $\mathcal{R}^2$, minus the configurations along the diagonal in which the objects would be in collision with one another.

- The root of the tree is the starting state of the regression search. It is the goal condition $In(a, goal)$. That logical condition corresponds to a region of configuration space illustrated in red. The goal does not specify the location of b, so it can have any legal value; but the left edge of a is constrained to lie in an interval that will guarantee that the entire volume of a is in the goal region. For reference in all of the configuration-space pictures in this figure, the starting state, which is a specific position of both boxes, is shown as a black dot.

- The first choice in the search process is to choose a location for a that will cause it to be contained in the goal region. A *generator* will construct or sample possible locations for a; we illustrate a specific choice *aLoc*, shown in figure 3 as a short vertical blue line. The logical

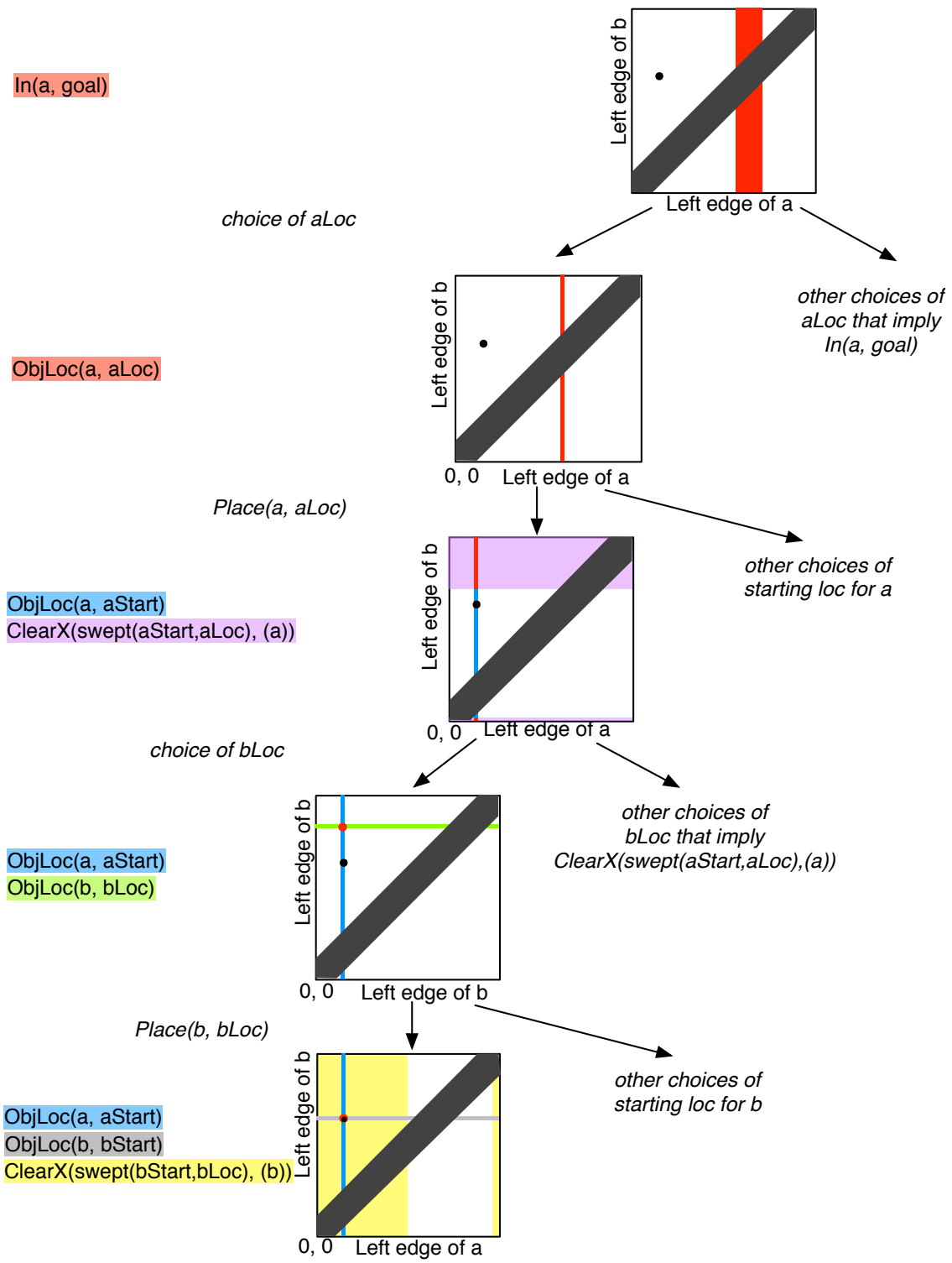


Figure 4: A path through the regression search tree in configuration space for the simple one dimensional environment.

goal $In(a, goal)$ is reduced to $ObjLoc(a, aLoc)$. This is not the entire pre-image of the goal, and for completeness it might be necessary to backtrack and search over different choices.

- The next step in the search is to compute the pre-image of $ObjLoc(a, aLoc)$ under the action $Place(a, aLoc)$. In order for that action to be successful, it must be that a is located at some initial location. The PLACE operator allows the starting location of the object to be chosen from among a set of options including that object's location in the initial state, as well as other salient locations in the domain. In this example, we choose to move it from its location in the initial state, *aStart*; to enable that move, we must guarantee that the region of space through which a moves as it goes from *aStart* to *aLoc* is clear of other objects. The requirement that a be located at *aStart* corresponds to the vertical blue locus of points in the configuration space. The requirement that the swept volume be clear turns into a constraint on the location of b. If the left edge of b is in one of the intervals shown in purple, then it will be out of the way of moving a. The pre-image is the conjunction of these two conditions, which is the intersection of the blue and purple regions in the configuration-space figure, which is shown in red.

- The current configuration of the objects (black dot) is not in the goal region, so the search continues. We search for a placement of object b that will cause it to satisfy the *ClearX* constraint, suggesting multiple positions including *bLoc*. This leads to the configuration-space picture in which the green line corresponds to object b being at location *bLoc*. Intersected with the constraint, in blue, that a be at location *aLoc*, this yields the subgoal of being in the configuration indicated by the red dot.

- Finally, the operator $Place(b, bLoc)$ can be used to move object b, subject to the preconditions that b be located at *bStart* (shown as the gray stripe) and that the swept volume for b from *bStart* to *bLoc* be clear except for object b. The clear condition is a constraint on object a to be in the yellow region. The intersection of all these conditions is a point which is equal to the starting configuration, and so the search terminates. (Note that, in general, the pre-image will not shrink to a single point, and the search will terminate when the starting state is contained in a pre-image).

The result of this process is a plan, read from the leaf to the root of the tree, to move object b to location *bLoc* and then move object a to location *aLoc*, thereby achieving the goal.

1.1.3 Hierarchy

Viewed as motion planning problems, the tasks that we are interested in solving have very high dimensionality, since there are many movable objects, and long planning horizons, since the number of primitive actions (such as linear displacements) required for a solution is very large. They are also *multi-modal* [Siméon et al., 2004, Hauser and Ng-Thow-Hing, 2011, Barry et al., 2012] requiring choosing grasps and finding locations to place objects, as well as finding collision-free paths. As such, they are well beyond the state of the art in motion planning.

Viewed as task-planning problems, the tasks that we are interested in solving require plans with lengths in the hundreds of actions (such as pick, place, move, cook). The combination of long horizon, high forward branching factor, and expensive checking of geometric conditions means that these problems are beyond the state of the art for classic domain-independent task-planning systems.

Our approach will be to use hierarchy to reduce both the horizon and the dimensionality of the problems.

- We reduce the horizon with a temporal hierarchy in terms of abstract actions, which comprise the process of achieving conditions to enable a primitive action as well as the primitive action itself. The lowest layer of actions are hand-built primitive actions, and the upper layers are constructed dynamically by a process of postponing pre-conditions of actions.

- The hierarchy also reduces dimensionality of sub-problems because fewer objects and fewer properties of those objects are relevant to sub-problems; the factored nature of the logical representation, which allows us to mention only those aspects of state that are relevant, naturally and automatically selects a space to work in that has the smallest dimensionality that expresses the necessary distinctions.

We have developed a hierarchical interleaved planning and execution architecture. In HPN, a top-level plan is made using an abstract model of the robot's capabilities, in which the "primitive" actions correspond to the execution of somewhat less abstract plans. The first subgoal in the abstract plan is then planned for in a less abstract model, and so on, until an actual primitive action is reached. Primitive actions are executed, then the next pending subgoal is planned for and executed, and so on. This process results in a depth-first traversal of a planning and execution tree, in which detailed planning for future subgoals is not completed until earlier parts of the plan have both been constructed and executed. It is this last property that we refer to as "in the now."

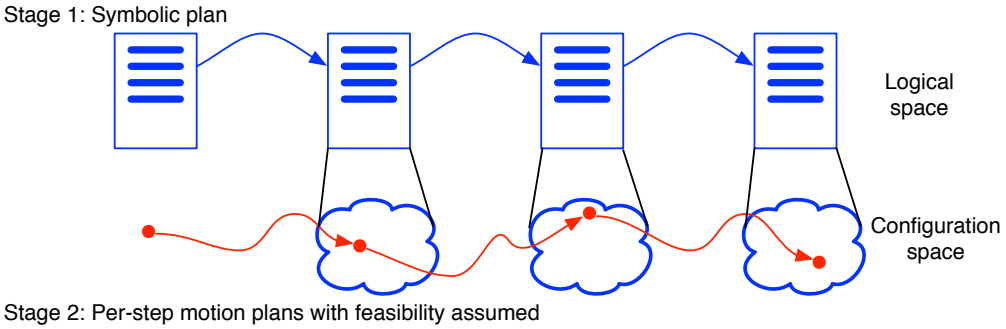
This mechanism results in a natural divide-and-conquer approach with the potential for substantially decreasing complexity of the planning problem, and handles unexpected effects of execution through monitoring and replanning in a localized and efficient way. On the other hand, a consequence of decomposition is that highly coupled problem instances will generally not be handled efficiently (but, there is no method that can handle all problems efficiently). We believe this trade-off is worth making since we expect that highly coupled problems are relatively rare in common-place activities. It also requires care and insight to construct a hierarchical domain specification that works effectively with HPN. We hope that such domain descriptions can eventually be constructed automatically, but experience with some hand-built hierarchies is a pre-requisite to developing an automatic approach.

1.2 Related work

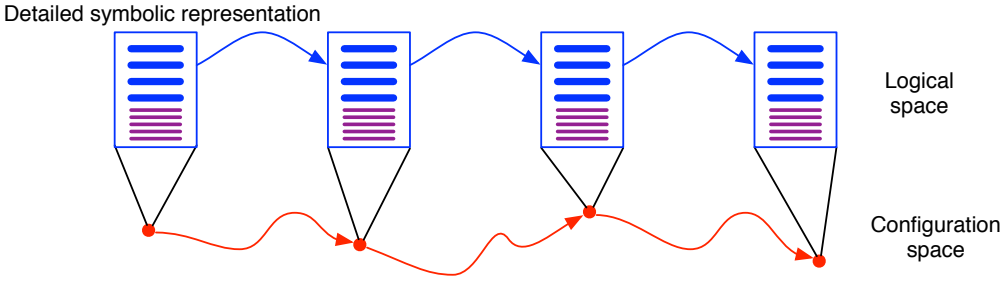
There is a great deal of work related to our approach, in both the robotics and the artificial intelligence planning literatures. We discuss a small part of the related work in robotics manipulation planning, in hierarchical AI planning, and work that seeks to integrate these two approaches.

1.2.1 Integrating logical and geometric planning

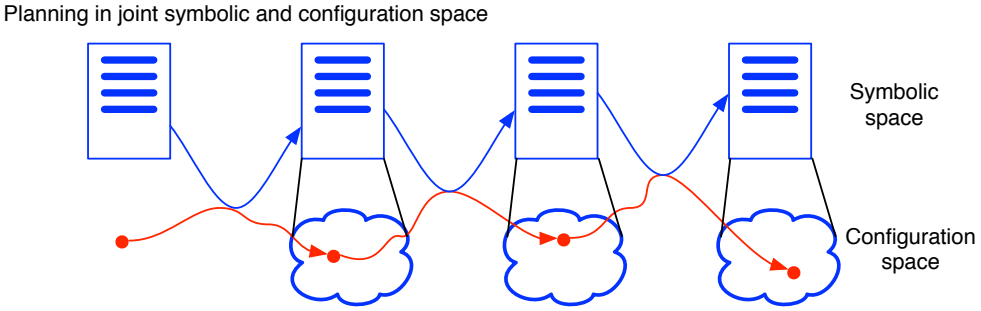
The need for integrating geometric and task planning has long been recognized [Nilsson, 1984, Kambhampati et al., 1991, Malcolm, 1995]. It is a difficult problem, requiring integration or coordination between planning processes in two very different spaces: the space of logical representations and the space of points in a geometric configuration space. Figure 5 gives a schematic illustration of some strategies for performing this integration, including the HPN approach. In these figures, there is a logical state description, represented as a conjunction of logical assertions, and a geometric



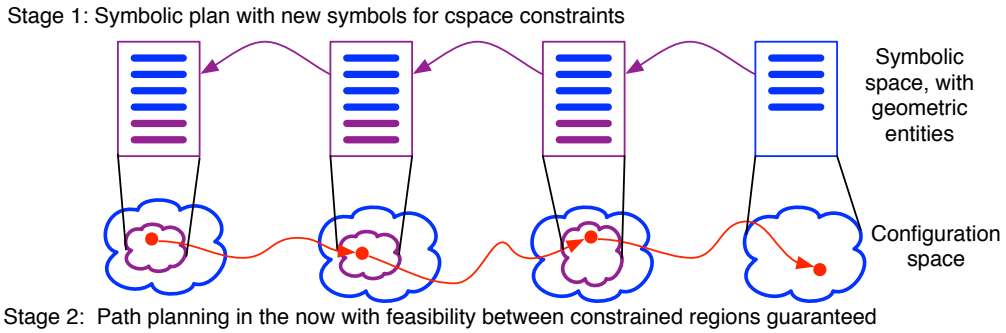
(a) Shakey [Nilsson, 1984]: logical phase followed by geometric phase.



(b) Dornhege et al. [2009]: geometric configuration encoded in logical state.



(c) ASyMov [Cambon et al., 2009]: search in product of logical and configuration spaces.



(d) HPN: Backward search in logical space, dynamically incorporating geometric constraints.

Figure 5: Schematic illustration of different methods for combining logical and geometric planning. In each case, the logical space is illustrated as a list of logical assertions (blue or purple lines), red dots represent points in the complete geometric configuration space of the problem, blue or purple clouds represent subsets of configuration space that are consistent with the associated logical state, and red arrows represent geometric paths through configuration space.

state description, represented as poses of all movable objects and the robot. In our schematics, the logical state is shown as a "document", with horizontal blue lines representing logical assertions as "sentences" in the document. Each logical state corresponds to a subset of the space of possible configurations of the objects and robot. In the figures, that subset is shown as a blue "cloud" in configuration space; individual configurations are red dots within the cloud. A blue arc between logical states indicates that the transition between those states is allowable according to the logical domain dynamics. A red arc between geometric configurations indicates that there is a collision-free path that respects the geometric and dynamic constraints of the objects and robot between those two configurations.

The simplest approach, taken in the Shakey project [Nilsson, 1984] and illustrated in figure 5(a), is to plan in two stages. The environment is represented logically at a high level of abstraction (for example, characterizing which objects are in which rooms or whether an object is blocking a door) and in the first stage an abstract plan is constructed using that representation. It is assumed that robot motion plans can be constructed that will implement the abstract actions in the logical plan: those are constructed step-by-step in a second stage, as the high-level plan is executed.

A risk of this approach is that the high-level actions will not, in fact, be executable. Dornhege et al. [2009] extend this approach by adding the ability to use general computational procedures to evaluate the symbolic predicates and to compute the effects of taking actions. They represent a detailed geometric state (including the robot pose) in the logical state and run a motion-planning algorithm to validate proposed logical actions, so that at the end of the planning process they have a complete, valid, geometric plan. One drawback of this approach is that choices for geometric variables, such as grasps and object placements, need to be discretized in advance and there is a very large forward branching factor.

Instead of having the logical search be prior to the geometric search or to contain it, we may instead think of the search as proceeding in a product space of logical and geometric spaces, representing both discrete properties of the state and the poses of objects and the robot. This approach was pioneered by Cambon et al. [2009] in the ASyMov system, and is illustrated in figure 5(c). This system performs a forward search in the product space, using purely logical plans as a heuristic for searching in the product space. At the geometric level, a sample-based planner is used to find a low-level motion plan that "threads" through the geometric images of a sequence of logical states that satisfy logical transition constraints and ultimately terminates in a goal region. The search in ASyMov is made more efficient by first testing feasibility of geometric transitions in roadmaps that do not contain all movable obstacles, and then doing a complete verification of candidate solutions. Significant attention is paid to managing the search process, because the sample-based geometric planning methods never fail conclusively, making it critical to manage the allocation of time to different motion-planning problems well. Plaku and Hager [2010] extend this approach to handle robots with differential constraints and provide a utility-driven search strategy.

The HPN approach, illustrated in figure 5(d) is neither driven by a static logical representation of the environment nor carried out simultaneously in the geometric and logical spaces. *It can be thought of as dynamically constructing new logical entities that represent geometric constraints such that the existence of a logical plan guarantees the existence of a geometric plan.* It proceeds in two stages. The first stage takes place at the logical level, proceeding backward from the goal, which projects into a set of geometric configurations. Generally speaking, the pre-image of a logical action will require the creation of new entities, in order for its geometric aspects to be described logically. For example, if the last action is to place an object in some region of space, we may need

to create entities representing the object's pose and the volume of space that must be clear in order to place the object. Logical sentences that represent these new geometric entities and constraints are shown in purple in the logical states of the figure (they are purple, because they combine blue logical representation with red geometric representation). The determination of pre-images requires geometric computation, but does not commit to a particular path through configuration space. Thus, after the first logical planning stage, we have a sequence of "subgoals" described logically and shown as purple projections into configuration space. In the second stage, during execution, any motion planner can be called to find a path from the current configuration to any configuration in the next purple cloud.

In contrast to the methods described thus far, which use *generative* planning methods to synthesize novel action sequences from declarative descriptions of the effects of actions, Wolfe et al. [2010] use a *hierarchical task network* HTN [Nau et al., 2003] approach, in which a kind of non-deterministic program for achieving the task is specified. Logical representations are used at the high levels, but the primitive actions are general-purpose programs combining perception and geometric planning. The focus of the algorithm is on finding the cost-optimal sequence of actions that is consistent with the constraints specified by the HTN. Significant efficiency is gained by focusing only on relevant state variables when planning at the leaves. It is difficult to compare the HTN approach directly to the generative planning approaches: it generally requires more knowledge to be supplied by the authors of the domain description; however, the ability to supply that control knowledge may make some problems tractable that would not be solvable otherwise.

Haigh and Veloso [1998] describe the design of ROGUE, an architecture for integrating symbolic task planning, robot execution and learning. The task planner in ROGUE is the Prodigy4.0 planning and learning system [Fink and Veloso, 1996]. Prodigy, like HPN, uses backward chaining to construct sequences of goal-achieving plan steps; it also uses forward simulation of plan steps to update its (simulated) current state. ROGUE builds on this approach, interleaving planning and execution on the real robot. The system executes a step on the robot when Prodigy selects a plan step to simulate. This is similar to HPN's approach to planning and executing "in the now", although ROGUE did not use hierarchical planning. Nourbakhsh [1998] also describes a hierarchical approach to interleaving planning and execution that is similar to ours, but which does not integrate geometric reasoning.

Keller et al. [2010] outline a planning and execution system that is in spirit very similar to ours, using the planning method described by Dornhege et al. [2009]. Uncertainty is handled with online execution monitoring and replanning, including the ability to postpone making plans until sufficient information is available. Beetz et al. [2010] outline an ambitious system for downloading abstract task instructions from the web and using common-sense knowledge to expand them into detailed plans ultimately executable on a mobile manipulator. In the context of that project, Kresse and Beetz [2012] specify a system that combines logical reasoning with rich force-control primitives: the high-level planner specifies constraints on motions and the low-level planner exploits the null space of those constraints to find feasible detailed control policies for the robot.

1.2.2 Manipulation planning with multiple objects

Most work on manipulation planning focuses on interacting with a single object, and is often restricted to path planning for a fixed target configuration or to grasp selection. A few early systems, like Handey [Lozano-Pérez et al., 1987], and more recent systems, like HERB [Srinivasa et al., 2009] and OpenRave [Diankov, 2010], integrate perception, grasp and motion planning, and control. Siméon et al. [2004] present a general decomposition method for manipulation planning

that allows complex plans involving multiple regrasp operations.

Planning among movable obstacles generalizes manipulation planning to situations in which additional obstacles must be moved out of the way in order for a manipulation or motion goal to be achieved. In this area, the work of Stilman and Kuffner [2006] and Stilman et al. [2007] allows solution of difficult manipulation problems that require objects to be moved to enable motions of the robot and other objects. This method takes an approach very similar to ours, in that it plans backwards from the final goal and uses swept volumes to determine, recursively, which additional objects must be moved and to constrain the system from placing other objects into those volumes. It does not, however, offer a way to integrate non-geometric state features or goals into the system or allow plans that move an object more than once. van den Berg et al. [2008] develop a probabilistically complete approach to this class of problems, but it requires explicit computation of robot configuration spaces and so it is limited to problems with few degrees of motion freedom.

Work by Dogar and Srinivasa [2011] augments pick-and-place with other manipulation modalities, including push-grasping, which significantly improve robustness in cluttered and uncertain situations. Planning in hybrid spaces, combining discrete mode switching with continuous geometry, can be used to sequence robot motions involving different contact states or dynamics. Hauser and Latombe [2010] have taken this approach to constructing climbing robots. Recently, Hauser and Ng-Thow-Hing [2011] have formulated a very broad class of related problems, including those with modes selected from a continuous set, as multi-modal planning problems and offer a general sample-based approach to them.

There are other approaches to multi-level treatment of the robot-motion planning problem. Zacharias and Borst [2011] introduce the capability map, which characterizes the difficulty of a motion planning problem in advance of solving it, based on the number of orientations from which a position can be reached. Such a capability map can be used, top-down, to help generate feasible planning problems at more concrete levels of abstraction. Stulp et al. [2012] learn to characterize places from which different types of actions can be executed effectively. Westphal et al. [2011] interweave reasoning about qualitative spatial relationships with sample-based robot motion planning. A qualitative plan is made using highly approximate models of objects and the robot gripper; this qualitative plan is used to guide sampling in the motion planner with a considerable gain in efficiency. Plaku et al. [2010] address robot motion planning with dynamics using a hybrid discrete-continuous approach that uses high-level plans in a discrete decomposition of the space to guide sample-based planning.

1.2.3 Hierarchical and factored planning

Hierarchical approaches to planning have been proposed since the earliest work of Sacerdoti [1974], whose ABSTRIPS method generated a hierarchy by postponing preconditions. Hierarchical planning techniques were refined and analyzed in more detail by Knoblock [1994] and Bacchus and Yang [1993] among others. These approaches may require backtracking across sub-problems, and it has also been shown that abstractions can slow planning exponentially [Bäckström and Jonsson, 1995]. The Fast Downward system of Helmert [2006] and its descendants are some of the most efficient current logical planners: their strategy is to convert the environment into a non-binary representation that reveals the hierarchical dependencies more effectively, then exploit the hierarchical causal structure of a problem to solve efficiently.

Work on *factored planning* generalizes the notion of hierarchical planning, and seeks to avoid the

costly backtracking through abstractions of previous approaches. In factored planning, the problem is divided into a tree of sub-problems, which is then solved bottom-up by computing the entire set of feasible condition-effect pairs (or sometimes more complex relations) for that subproblem, and passing that result up to the parent. This approach was pioneered by Amir and Engelhardt [2003] and advanced by others, including Brafman and Domshlak [2006] and Fabre et al. [2010].

Marthi et al. [2007, 2008] give mathematical semantics to hierarchical domain descriptions based on angelic non-determinism, and derive a planning algorithm that can dramatically speed up the search for optimal plans based on upper and lower bounds on the value of refinements of abstract operators. They also give conditions under which plan steps may be executed before planning is complete without endangering plan correctness or completeness. Goldman [2009] gives an alternative semantics that incorporates angelic non-determinism during planning and adversarial non-determinism during execution.

The angelic-semantics methods are more directed than the factored-planning approaches: they use top-down information to focus computations at the lower levels, but require the decomposition to be done by hand. Mehta et al. [2009] pursue a related approach, but automatically learn a decomposition, in domains with a strict serializability property. Subgoals *serialize* if there is an ordering of them such that one may be achieved first and then maintained true during the achievement of the other.

Our approach differs from these in that it neither backtracks through abstractions during planning, nor computes sets of solutions, nor assumes strong serializability. It makes top-down commitments to subgoals that are such that, if the subgoals do serialize, then the planning and execution are both efficient. If the decomposition assumptions fail, but the environment is such that no actions are irreversible, the system will eventually achieve the goal by falling back to planning in the primitive model. The result is that problems that are decomposable can be solved efficiently; otherwise, the system reverts to a non-hierarchical approach.

1.3 Outline

The rest of the paper is organized as follows. Section 2 introduces the representations and algorithms we use for planning in discrete and continuous domains. It uses as a running example a highly simplified one-dimensional "kitchen" environment: this domain combines logic and geometry in interesting and subtle ways, but is sufficiently simple that it can be presented and understood in its entirety. Section 3 introduces the HPN recursive planning and execution architecture, explains our flexible approach to hierarchical planning, and illustrates these ideas in the one-dimensional kitchen environment. Section 4 describes the formalization, in HPN, of a large-scale mobile manipulation environment with a Willow Garage PR2 robot as well as results of tests and experiments. In section 5 we describe the strengths and limitations of the system and articulate directions for future work. Throughout the paper, many of the detailed definitions and examples are postponed to appendices; it is safe to delay reading them until more detail is desired.

2 Representation and planning

Fluents are logical assertions used to represent aspects of the state of the external physical world that may change over time; conjunctions of fluents describe sets of world states, and are used to to specify goals and to represent pre-images. In a finite domain, it is possible to describe a world state

completely by specifying the truth values of all possible fluents. In continuous geometric domains, it is not generally possible to specify the truth values of all possible fluents; but fluents remain useful as a method for compactly describing infinite sets of world states.

We represent goals and subgoals as conjunctions of fluents that characterize subsets of the underlying state space, including both geometric and non-geometrical properties, and represent the current state of the world using a typical non-logical world model that describes positions, orientations and shapes of objects and the configuration of the robot, as well as other properties such as whether plates are clean or food is cooked.

We begin this section by introducing an extension of our simple planning environment, which we use to motivate and make concrete the descriptions in the following sections. We then describe the geometric representation of world states and the logical representation of goals. The logical representation forms the basis for describing the operations in the environment. We show how to extend a standard operator-description formalism to handle our infinite domain, and present the regression planning algorithm in detail.

2.1 One-dimensional kitchen environment

So that we can build intuition and provide completely detailed examples during the narrative of this section, we will work with a highly simplified one-dimensional kitchen environment. In this environment:

- Objects have finite extent along a line. They can be moved along the line but cannot pass through each other.
- The robot operates like a crane, in the sense that it is above the line and need not worry about collision with the objects.
- A segment of the line is defined to be a *sink* region where objects can be washed.
- A segment of the line is defined to be a *stove* region where objects can be cooked.
- Objects must be washed before they can be cooked.
- The goal is to have several objects be cooked.

Figure 6 shows a plan and execution sequence for the goal of cooking a single object. In section 4 we extend this environment to operate in three spatial dimensions, with a real robot model and a large number of both permanent (not movable) and movable objects.

2.2 World states

A *world state* is a completely detailed description of both the geometric and non-geometric aspects of a situation. World states can be represented in any way that is convenient. We never attempt to represent a complete characterization of the world state in terms of fluents. The only requirement is that, for each fluent type, there is a procedure that will take a list of arguments and return the value of that fluent in the world state.

In the one-dimensional kitchen environment, the world state is a dictionary mapping the names of objects to a structure with attributes for location, size, whether or not they are clean and whether or not they are cooked. In this structure, $s[o].loc$ represents the location of object o in world state

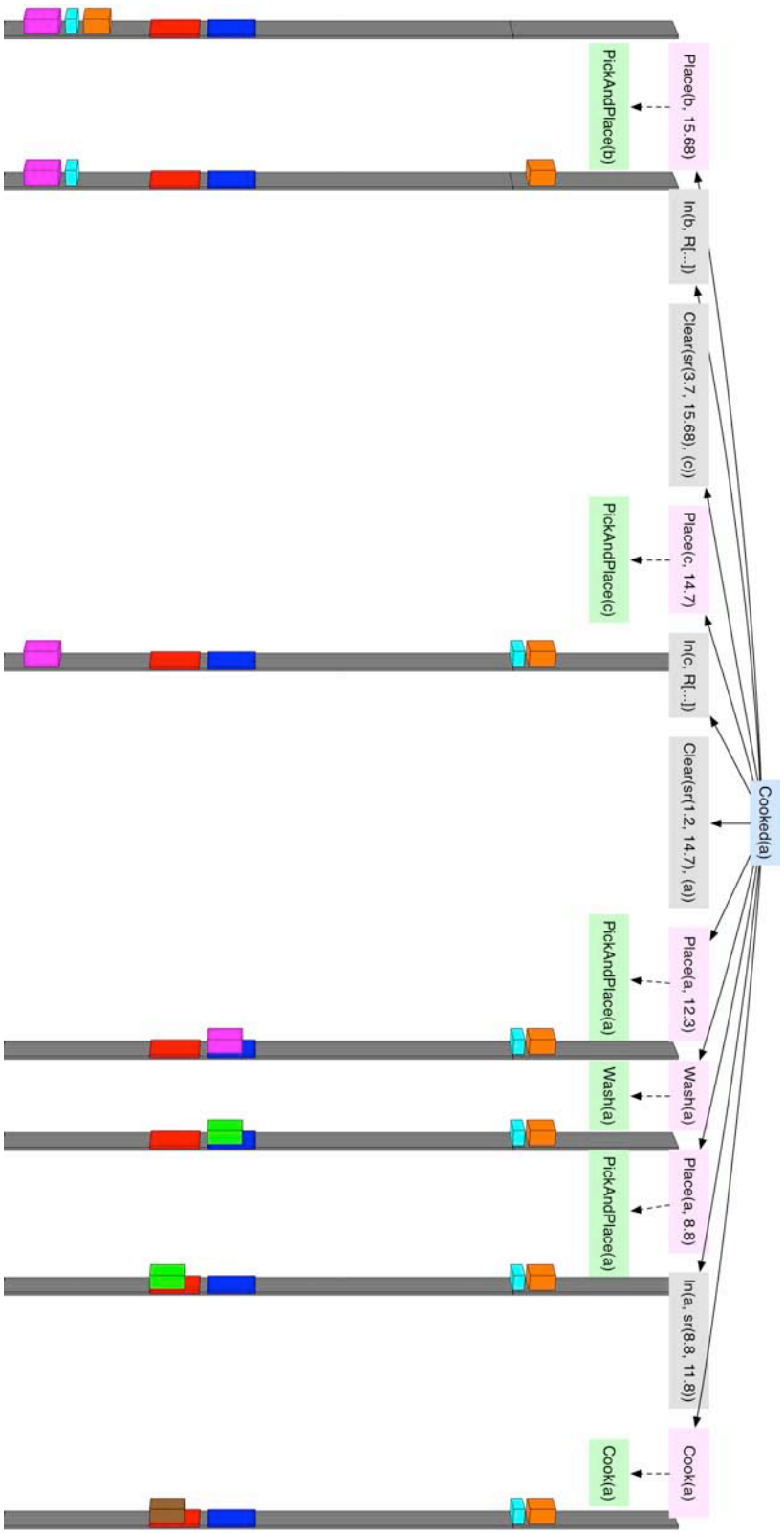


Figure 6: Plan for cooking object a (magenta). It requires moving objects b (orange) and c (cyan) out of the way first, then putting a in the sink, washing it, putting a on the stove, and finally cooking it. When an object is clean, it becomes green; when it is cooked, it becomes brown. The red region is the *stove* and the blue region the *sink*; the leftmost object is a.

s. We also use $s[o].min$ and $s[o].max$ to name the left and right edges of the object (loc is the same as min).

2.3 Fluents

A *fluent* is a condition that can change over time, expressed as a logical predicate applied to a list of arguments, which may be constants or variables. We use these logical expressions to specify conditions on geometric configurations of the robot and the objects in the world, as well as other non-geometric properties, such as whether something is clean or cooked. The meaning of a fluent f is defined by specifying a test,

$$\tau_f : args, s \rightarrow \{\text{TRUE}, \text{FALSE}\} \ ,$$

where *args* are the arguments of the fluent, s is a world state and the value of the test is either TRUE or FALSE.

A *ground* fluent (all of whose arguments are constants) then represents the subset of all possible world states in which the associated test is true. A conjunction of fluents represents a set of world states that is the intersection of the world state sets represented by the elements of the conjunction; this is the same as the set of world states in which the tests of *all* of the fluents evaluate to true.

The fluents we use to describe the one-dimensional kitchen domain, together with their tests, are the following:

- $ObjLoc(o, l)$: the left edge of object o is at (real-valued) position l,

  $$\tau_{ObjLoc}((o, l), s) := |s[o].loc - l| < \delta \ .$$

  where δ is a small value used throughout this implementation to make the numerical operations robust.

- $In(o, r)$: object o is entirely contained in region r,

  $$\tau_{In}((o, r), s) := (r.min \leq s[o].min + \delta) \ \& \ (r.max \geq s[o].max - \delta) \ .$$

- $ClearX(r, x)$: region r is clear, except for exceptions, x, which is a set of objects,

  $$\tau_{ClearX}((r, x), s) := \forall o \notin x. \ s[o] \cap r = \emptyset \ .$$

- $Clean(o)$: object o is clean,

  $$\tau_{Clean}((o), s) := (s[o].clean = \text{TRUE}) \ .$$

- $Cooked(o)$: object o is cooked,

  $$\tau_{Cooked}((o), s) := (s[o].cooked = \text{TRUE}) \ .$$

A goal for our planning and execution system is a conjunction of ground fluents describing a set of world states. The goal of having object a be clean and object b be cooked can be articulated as:

$$Clean(a) \ \& \ Cooked(b) \ .$$

During the course of regression-based planning, intermediate goals are represented as conjunctions of fluents as well. We will write $s \in G$ to mean that world state s is contained in the set of goal states G.

2.4 Operator descriptions

We describe the dynamics of the environment (that is, the effects of the robot's actions on the state of the world) using operator descriptions. They are based on ideas from STRIPS [Fikes and Nilsson, 1971] and PDDL [McDermott, 2000], but are extended to work effectively in continuous domains as described in section 2.5 and to form a hierarchy in section 3. Rather than defining a traditional formal language for describing operators, our implementation makes use of procedures to compute the preconditions of an operation. Each operator description has the following components:

- **variables:** Names occurring in the **effect** fluents and **choose** clauses. An operator description is actually a *schema*, standing for concrete operator descriptions for all possible bindings of the variables to entities in the domain.
- **effect**: A list of fluents with variables as arguments, characterizing the effects that this operator can be used to achieve.
- **precondition**: A list of fluents that describe the conditions under which, if the operator is executed, the effect fluents will be guaranteed to hold.
- **primitive**: An actual primitive operation that the robot can execute in the world.
- **choose**: A list of variable names and, for each, a set of possible values it can take on.

The semantics of an operator description is that, if the primitive action is executed in any world state s in which all of the precondition fluents hold, then the resulting world state will be the same as s, except that any fluent specified as an effect will have the value specified in the resulting state.

The formalization of the dynamics of the one-dimensional kitchen environment in terms of operator descriptions illustrates the use of operator descriptions and provides a completely concrete description of the planning domain. Each operator description takes the state of the world at planning time, s_{now}, and the goal to which it is being applied during goal regression, γ, as arguments. We explain the roles of s_{now} and γ when we discuss generators in section 4.1.4.

- An object can be washed if it is in the sink.

  WASH$((o), s_{now}, \gamma)$:
  effect: $Clean(o)$
  pre: $In(o, sink)$

- An object can be cooked if it is clean and on the stove.

  COOK$((o), s_{now}, \gamma)$:
  effect: $Cooked(o)$
  pre: $In(o, stove)$
  $Clean(o)$

- An object can be moved to a target location if there is an unoccluded path for it. The PICKPLACE(o, l_{target}) operator moves object o to target location l_{target}. It has a **choose** variable, l_{start}, which is the *source location* from which the object is being moved to l_t. These

values are provided by *generators*, which are discussed further in section 2.5.3. The location of the object in the plan-time state, $s_{now}[o].loc$, is a reasonable starting location, as are extreme valid positions within each of the important regions of the domain.

For each generated value of l_{start}, there is a instantiation of this operator, which has as its preconditions that object o be located at l_{start} and that the volume of space that o is swept through as it moves from l_{start} to l_{target} be clear of all objects except for o. We use *warehouse* here to refer to some region of space that is "out of the way" that might be use as a temporary storage place for objects.

PickPlace$((o, l_{target}), s_{now}, \gamma)$:
effect: $ObjLoc(o, l_{target})$
choose: $l_{start} \in \{s_{now}[o].loc\} \cup$ generateLocsInRegions$((o, \{warehouse, stove, sink\}), s_{now}, \gamma)$
pre: $ObjLoc(o, l_s)$
$ClearX(sweptVol(o, l_s, l_t), \{o\})$

In this simple one-dimensional world, the swept "volume" is simply an interval comprising the volume of the object at the start and target locations and the volume between them:

$$sweptVol(o, l_s, l_t) = [\min(l_s, l_t), \max(l_s, l_t) + o.size]$$

The remaining two operators are used to achieve the *ClearX* and *In* fluents. They are *definitional* in the sense that they are not associated with a primitive action: they serve as a kind of inference rule during the planning process, removing the effect fluent from the goal and replacing it with a set of "precondition" fluents.

- The conditions for placing an object into a region reduce to the placement of the object at an appropriate target location, that will cause the object to be contained in the region.

  In$((o, r), s_{now}, \gamma)$:
  effect: $In(o, r)$
  choose: $l \in$ generateLocsInRegion$((o, r), s_{now}, \gamma)$
  pre: $ObjLoc(o, l)$

- To specify that region r must be clear except for the set x of "exception" objects that are allowed to overlap the region, we rewrite the condition positively, saying that all objects not in the set x must be completely contained in the region of space that is the complement of r; that is, they are in the set $U \setminus r$, where U is the spatial universe of the domain and $\setminus$ is the set difference operator.

  Clear$((r, x), s_{now}, \gamma)$:
  effect: $ClearX(r, x)$
  pre: $In(o, U \setminus r)$ **for** $o \in (allObjects \setminus x)$

2.5 Extending operator descriptions to continuous domains

In order to adequately formalize operation in continuous domains, we have made several extensions to the typical planning formalisms for discrete domains. These include extending the notion of a primitive action, allowing for limited forms of logical reasoning, and handling variables with infinite domains.

2.5.1 Primitives

Actions that are primitive from the perspective of task planning can themselves be very complex, in terms of computation, sensing, and actuation. For example, if *pick* is a primitive operation, specifying an object and a grasp for that object, it might require the robot to visually sense the object's location, call a motion planner to generate a path to a pre-grasp pose, execute that trajectory, and finally execute a robust grasping subroutine that uses tactile information to close in on the object appropriately.

2.5.2 Limited logical reasoning

In the original STRIPS formulation of planning, all arguments of all fluents were discrete and each STRIPS rule stated all positive and negative effects of each action; reasoning about the effects of actions was then limited to set operations on the fluent sets characterizing the subgoals, preconditions, and effects of operators.

In HPN, fluents may have continuous values or values drawn from very large discrete sets as arguments, making it typically impossible to enumerate every fluent that becomes true or false as a result of taking an action. To account for this, we must perform some limited logical reasoning. We do this by providing a method for each fluent that takes another fluent and determines whether the first fluent *entails* the second fluent and a method that takes another fluent and determines whether the first fluent *contradicts* the second fluent.

A fluent f_1 entails fluent f_2 if and only if the set of states in which f_1 holds is a (possibly proper) *subset* of the set of states in which f_2 holds. A fluent f_1 contradicts fluent f_2 if the set of states in which f_1 holds has an empty intersection with the set of states in which f_2 holds.

Sometimes the effect of an operator on a state (and, therefore, its regression) is more subtle than simply contradicting or entailing another fluent. For instance, the regression of the condition of having x dollars under the action of spending y dollars is having $x+y$ dollars. We additionally allow the specification of a *regression* method for each operator description that takes a fluent as input and returns another fluent that represents the regression of the first fluent under the operation. The entailment, contradiction, and regression methods for the one-dimensional kitchen environment are described in detail in appendix A.

2.5.3 Generators

Most operations can be achieved in multiple ways: a robot can move along multiple paths, an object can be picked up from many starting locations, two objects may be fastened together with one of many different bolts. When an operator description has a clause of the form

$$\textbf{choose } v \in D \ ,$$

it means that there is a possible instantiation of the operator for each possible binding of variable v to a value in the set D.

In very large or continuous environments, however, it is infeasible to consider all possible bindings; in such cases, we may take advantage of special-purpose procedures that can take into account both the state of the world at planning time, s_{now}, and the subgoal being planned for, γ, and generate bindings for v that are likely to be successful in that context. The subgoal γ is used to avoid generating bindings that would result in operator preconditions that directly contradict γ. The current state s_{now} is used for heuristic purposes, to generate bindings for which the operator preconditions are relatively cheap to achieve in the current state:

$$\textbf{choose } v \in \mathit{generator}(\mathit{args}, s_{now}, \gamma) \ .$$

In our current implementation, generators just return a short list of possible bindings; it would be possible to extend the software architecture so that they are generators in the computer-science sense, which can be called again and again to get new values, to support backtracking search. In such an approach, with appropriate sampling methods, it may be possible to guarantee resolution completeness. However, we believe it will be most effective to emphasize learning to generate a small number of useful values, based in the current state and the goal.

For the one-dimensional kitchen environment, we have a single generator. The procedure

$$\textsc{generateLocsInRegion}((o, r), s_{now}, \gamma)$$

takes an object o, a target region r, planning-time state s_{now} and a goal γ, and returns a list of possible locations l of object o such that if o were placed at l,

- o would be contained in region r,
- o would not be in collision with any object that has a placement specified in goal γ, and
- o would not overlap any region required to be clear in goal γ.

It is preferred but not required that this location be clear in s_{now}: however, if necessary an occluding object in that location can be moved out of the way.

There are, generally, a continuum of such locations. This procedure returns the extreme points of each continuous range of legal locations:

$\textsc{generateLocsInRegion}((o, r), s_{now}, \gamma)$:
$C = \{r' \mid \mathit{ClearX}(r', x) \in \gamma \ \& \ o \notin x\}$
$O = \{\mathit{vol}(o', l) \mid \mathit{ObjLoc}(o', l) \in \gamma\}$
$R' = r \setminus (C \cup O)$
return $\bigcup_{\{\rho \in R' \mid f(o,\rho)\}} \{\mathit{min}(\rho), \mathit{max}(\rho) - o.\mathit{size}\}$

where $\mathit{vol}(o, l)$ is defined to be the volume of space taken up by object o if it is placed at location l.

It begins by finding the set C of all the regions r' that are required to be clear in the goal γ and for which o is not in the set of exceptions; then it finds the set O of all regions that are required to be occupied by placed objects in the goal γ; then R' is defined to be the set of contiguous regions that result from subtracting C and O from r. Finally, it constructs and returns a set of locations, by finding all of the regions ρ in R in which o fits, $f(o, \rho)$, and computing the leftmost and rightmost placements of o in ρ. Testing whether an object fits in a region is trivial in one-dimension: it is just necessary to compare the sizes. This procedure is straightforwardly modified to handle the case when r is in fact a non-contiguous region (represented as a set of contiguous regions).

2.6 Regression planning

The regression planning algorithm is an A^* search through the space of conjunctions of fluents, which represent sets of world states. It takes as arguments the planning-time world state s_{now}, the goal γ, and the abstraction level α, which is irrelevant for now but which we will discuss in detail in section 3. Pseudo-code for the goal-regression algorithm is provided in appendix B.1.

- The *starting state* of the search is the planning goal γ, represented as a conjunction of fluents;
- The *goal test* of the search is a procedure that takes a subgoal sg and determines whether all of the fluents are true in the current world state s_{now}.
- The *successors* of a subgoal sg are determined by the procedure APPLICABLEOPS. This procedure determines which instances of the operators are applicable to sg, and performs the regression operation to determine the pre-images of sg under each of those operator instances. It is described in detail in appendix B.1.
- The *heuristic* distance from the current state s_{now} to a subgoal sg is the number of fluents in sg that are not true in s_{now}.

It returns a list, $((-, g_0), (\omega_1, g_1), ..., (\omega_n, g_n))$ where the ω_i are operator instances, $g_n = \gamma$, g_i is the pre-image of g_{i+1} under ω_i, and $s_{now} \in g_0$. The pre-image g_i is the set of world states such that if the world is in some state g_i and action ω_{i+1} is executed, then the next world state will be in g_{i+1}.

In the current implementation, all operators are assumed to have cost 1; it is straightforward to extend the algorithm to handle different costs for different operators or operator instances. The heuristic we use here is not admissible: it is possible for a single action to make multiple fluents true. We expect that learning a heuristic that is both tighter and closer to admissible will also be an important future research question.

An example result of applying the regression planning algorithm to the one-dimensional kitchen environment is shown in figure 6. The plan, shown in the tree on the left, has 11 steps: gray nodes indicate definitional actions that have no primitive associated with them and green nodes indicate primitive actions. By moving object b (the orange object) to the right, it clears the region necessary to move object c (the cyan object) to the right as well. This makes room to move a to the washer. Object a is then washed, moved to the stove, and then cooked. This process is shown pictorially, with the initial configuration at the top of the figure, and subsequent world states shown below the primitive action that generated them.

Even this simple example illustrates the interplay between the geometric and the logical planning: geometric details affect which objects must be moved and in what order and logical goals affect the locations in which objects must be placed.

Thus far, our planning process has operated at only two levels: a completely geometric primitive level and a logical task level. This approach can only solve moderately small problems, but it will serve as the basis for the hierarchical extensions described in the next section.

3 Hierarchy and planning in the now

In this section, we provide a method for planning and execution with a multi-level hierarchical decomposition, illustrate a method for constructing such hierarchical decompositions, and demon-

strate these methods by handling large problem instances in the one-dimensional kitchen environment.

3.1 HPN architecture

The HPN process is invoked by HPN($s_{now}, \gamma, \alpha, world$), where s_{now} is a description of the state of world when the planner is called; γ is the goal, which describes a set of world states; α is a structure that controls the abstraction level, which we discuss in more detail in section 3.2.4; and *world* is an actual robot or a simulator on which primitive actions can be executed. In the prototype system described in this paper, *world* is actually a geometric motion planner coupled with a simulated or physical robot.

HPN calls the regression-based PLAN procedure, which returns a whole plan at the specified level of abstraction, $((-, g_0), (\omega_1, g_1), ..., (\omega_n, g_n))$. The pre-images, g_i, will serve as the goals for the planning problems at the next level down in the hierarchy.

```
HPN(s_now, γ, α, world):
    p = PLAN(s_now, γ, α)
    for (ω_i, g_i) in p
        if ISPRIM(ω_i)
            world.EXECUTE(ω_i, s_now)
        else
            HPN(s_now, g_i, NEXTLEVEL(α, ω_i), world)
```

HPN starts by making a plan p to achieve the top-level goal. Then, it executes the plan steps, starting with action ω_1, side-effecting s_{now} so that the resulting state will be available when control is returned to the calling instance of HPN. If an action is a primitive, then it is executed in the world, which causes s_{now} to be changed; if not, then HPN is called recursively, with a more concrete abstraction level for that step. HPN assumes that the fluents used to describe goals are the same at every abstraction level; that is, there is no state abstraction, only operator abstraction. The procedure NEXTLEVEL takes a level of abstraction α and an operator ω, and returns a new level of abstraction β that is more concrete than α.

Note that the abstract action ω_i does not directly affect the planning at the more concrete level; only the subgoal g_i is relevant. This means that HPN does not refine plans, in the sense of simply adding details to an existing plan; the plan at a more concrete level may not have any connection to the abstract action at the level above. Also note that HPN does not backtrack across abstraction levels; that is, it commits to the subgoals generated by planning at each level of abstraction. We discuss the consequences of this structure in more detail in section 3.2.3. If the hierarchical decomposition is well chosen, the HPN approach may result in a potentially enormous computational savings; if it is not, then the system may execute useless or retrograde actions.

Figure 7 shows a hierarchical version of the plan in figure 6. Blue nodes are goals for planning problems; pink nodes are operations associated with a concrete action; gray nodes are definitional operations. Operation nodes are prefixed with An where n is an integer representing the abstraction value at which that operator is being applied.

Instead of one large problem with an 11-step plan, we now have 5 planning problems, with solutions of length 1, 2, 3, 4, and 5. Because planning time is generally exponential in the length of the plan, the reduction in length of the longest plan is significant.

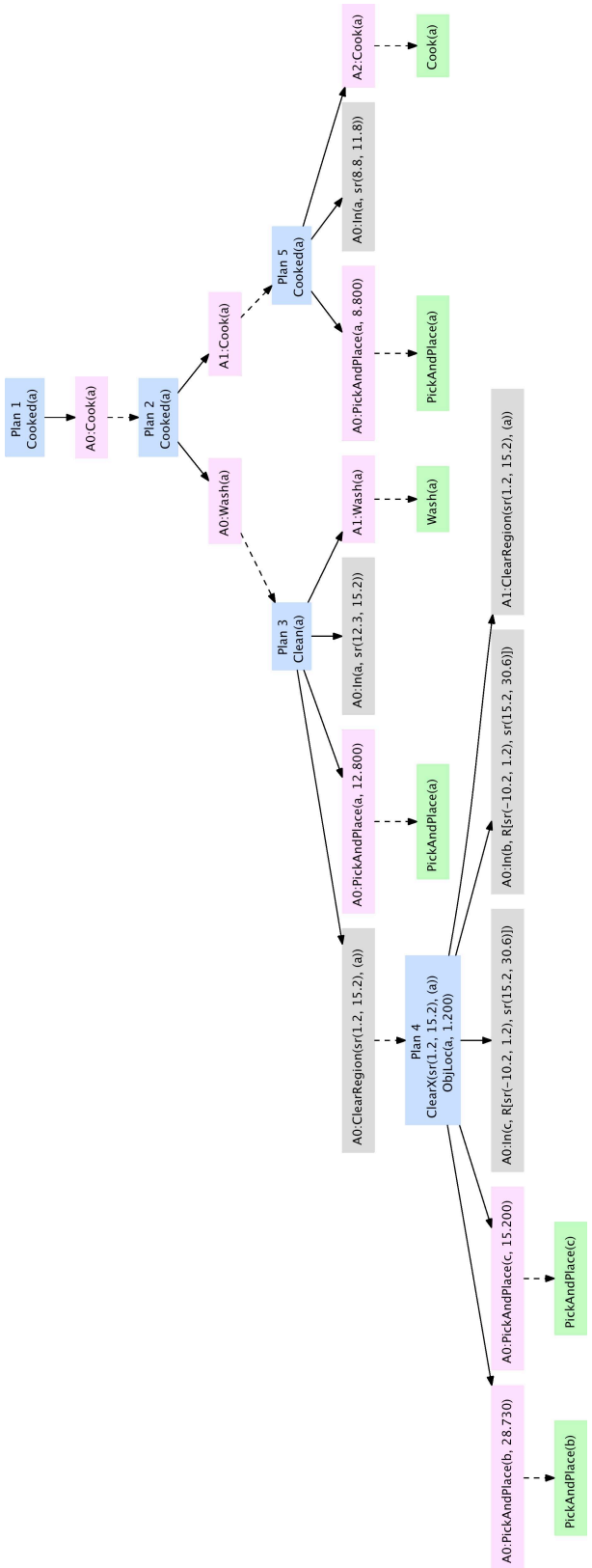


Figure 7: Hierarchical planning and execution trees for cooking object a. This is a solution to the same problem as the flat plan shown in figure 6.

We will discuss each of the planning problems (numbered blue nodes in the figure) in turn.

1. At the top level, there is a single item to be cooked, and the plan is trivial.

2. Next, cooking is considered in more detail, and it is determined that the object needs to be washed and then cooked.

3. To achieve the $Clean(a)$ condition, the planner considers the $Wash(a)$ action more concretely. It determines that it needs to put a into the sink and then wash it; it also determines that the swept volume required to move a into the sink is not clear and adds an operation to clear that volume at the beginning of the plan.

4. To clear the swept volume for a, objects b and c must be put into appropriate locations outside that volume.

   The steps of plan 4 are primitives, which are executed; then control is returned and the remaining steps of plan 3 are executed as well. at this point, object a has been washed.

5. The plan to cook a consists of moving it to the stove and performing the *Cook* operation, resulting in the achievement of the top-level goal.

In section 3.3 we present operator descriptions for this hierarchy; in appendix D we explore different abstraction choices, and see that postponing more preconditions tends to make the hierarchy narrower (plans tend to be shorter) and deeper (there tend to be more planning problems to solve). Generally, the computation time goes down as the hierarchy narrows and deepens; however, if too many preconditions are postponed, the algorithm will begin to make action choices too independently and will frequently have to "undo" and "re-do" conditions it has already achieved, resulting in both increased computation time overall and in increased suboptimality of the executed action sequence.

To handle possible errors in executing the actions, we take a simple approach of execution monitoring and replanning. We retain the same depth-first planning and execution architecture as in the basic HPN algorithm, but monitor the effects of actions in an effort to ensure that the action selected for execution is necessary to achieve the current subgoal and that it remains part of a valid plan for achieving the subgoal. If these conditions are violated at any level of the tree, the plan is discarded and a new plan is constructed.

In appendix C, we show that under the assumptions of reversibility and no infeasible subgoals, in finite domains, HPN is complete and correct, in the sense that it is guaranteed to terminate with the world in a state that satisfies the goal condition. HPN, in finite domains, with a finite set of abstraction levels that is well-founded, it terminates because it either recurses with an abstraction level that is strictly greater than the one in the invocation, or the relevant operation is already at maximum concreteness and is executed as a primitive with no further recursion.

However, in infinite domains, HPN may not be complete and correct. If the basic PLAN procedure is not complete, then an instance of HPN built upon it cannot be complete either. Additionally, the choice of hierarchical structure can affect completeness: the structure that we use (described in section 3.2) is very expressive and flexible, but in a domain in which it is possible for arbitrarily many new objects, such as swept volumes, to be created, the abstraction hierarchy may not be well founded, in which case HPN would not be guaranteed to terminate. This case is easily avoided by imposing a depth-limit in the recursion and attacking the subproblem in its maximally concrete form.

3.2 Constructing a hierarchy

Our goal in doing hierarchical planning is to improve planning efficiency, with some sacrifice to plan optimality but without compromising correctness and completeness. So, among all hierarchies that satisfy the formal requirements for soundness and completeness (discussed in appendix C), which ones will lead to systems with a good trade-off between planning time and execution suboptimality? We are currently unable to address this problem formally, but illustrate some trade-offs in the design of hierarchy through examples.

Ultimately, our goal in constructing a hierarchical abstraction is to find a sequence of subgoals that pose easier planning problems than planning for the whole goal at once. Intuitively, that means that, having achieved a subgoal in the sequence, it is desirable not to have to violate it in the future (else, what was the point of achieving it in the first place?).

More formally, we can see each of the subgoals, $g_0, \ldots, g_n$, in an abstract plan as a set of states. Each planning problem at the more concrete level of abstraction is to go from some starting state in g_i to any resulting state in g_{i+1}. If each of these problems is solved by a short plan, then solving the planning sub-problems will have been significantly easier than solving the whole problem, since planning complexity is generally exponential in the length of the plans. If, on the other hand, the g_i are poorly chosen, they may result in long plans to the next subgoal, which contributes both to longer planning times and longer resulting trajectories in the world. Barry et al. [2011] discuss related issues in the context of Markov decision processes.

Our approach to constructing hierarchies is inspired by the early work of Sacerdoti [1974], which constructs a hierarchy of abstractions of operators by postponing consideration of some or all of their preconditions. We have found, however, that the utility of a precondition-postponement hierarchy depends on the vocabulary of fluents available in the domain formulation. Some formalizations do not offer any way to describe useful subgoals, and so simple precondition-postponement does not offer useful opportunities for abstraction. But, as illustrated in this paper, mobile manipulation environments and other domains with a geometric component offer ample opportunities for building such hierarchies, by naming properties of regions of the space.

To build a hierarchy of models by postponing preconditions, we assign an abstraction value to each precondition, in each operator description. Value 0 is the most abstract, and there may be zero or more preconditions with value 0. There may be multiple preconditions added with any higher value, but there is no useful effect to having abstraction values higher than 0 without any new preconditions. Each operator description has defined $v_{\alpha_{max}}$, which is the most concrete abstraction value for that operation, representing a primitive operation.

Consider an operator description with effect fluent r:

pre: $p_1, \ldots, p_n$
prim: o

By postponing precondition p_n, we effectively create a new operator description that combines the achievement of p_n with the execution of the primitive:

pre: $p_1, \ldots, p_{n-1}$
prim: achieve p_n maintaining $p_1, \ldots, p_{n-1}$; o

where the semi-colon operator stands for sequencing. There are, however, some important issues that need to be considered when doing this. We outline them below and discuss them in more detail in subsequent sections.

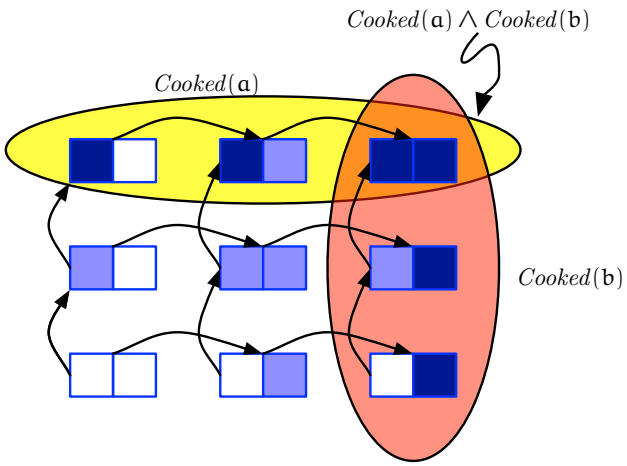


Figure 8: State transition diagram for abstract wash/cook environment: the box on the left indicates the state of a (dark blue is cooked, light blue is washed but not cooked, white is neither) and the box on the right, the state of b.

First, it may not be possible to make p_n true without undoing $p_1, \ldots, p_{n-1}$. In this case, the planner will achieve $p_1, ..., p_n$ as efficiently as possible, and then execute o, causing r to be achieved; this process may introduce significant suboptimality that could have been avoided by re-ordering or interleaving the achievement of the preconditions.

Second, effects of the abstracted operator will generally be different from the effects of the single primitive o, because the process of achieving p_n may have additional effects. There are also some more global issues to consider when using the abstracted operators within HPN. Because HPN does not backtrack across abstraction levels, we must ensure that it does not generate infeasible subgoals.

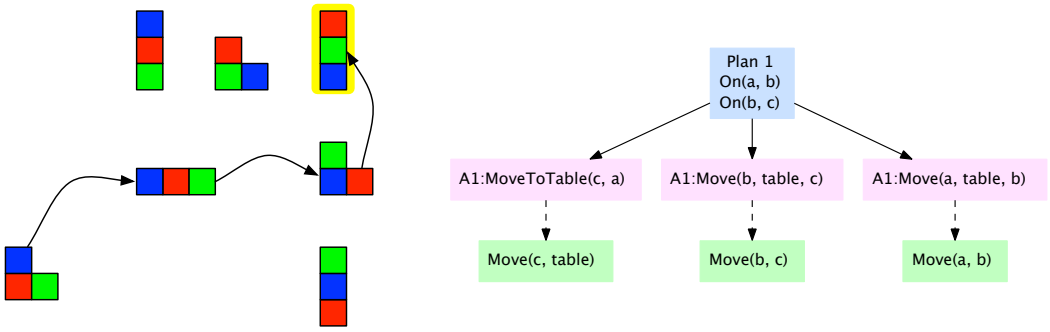
3.2.1 Serialization

One way to think about postponement of preconditions is that we are partially serializing subgoals. Consider a situation in which the goal of the planning process is the conjunction of two subgoals, and where each subgoal can be achieved by an operator with some postponed preconditions. The postponed preconditions can never be interleaved: the planning and execution process will achieve all of the preconditions for whichever operator is selected to be achieved first, then that operator will be executed, then the preconditions for the second operator will considered. When the subgoals are, in fact, serializable, this results in considerable computational savings at no cost in optimality of the plan; when they are not serializable, then we may incur more planning cost than solving the problem at the primitive level and be sub-optimal in execution as well.

If we consider just the WASH and COOK operations, with no *In* preconditions, and two objects, a and b, with the goal

$$Cooked(a) \ \& \ Cooked(b) \ ,$$

then we have the environment pictured in figure 8. The subgoals *Cooked*(a) and *Cooked*(b) are serializable, because we can achieve one of them and then maintain its truth while achieving the other. In the figure, the yellow region corresponds to all of the states in which a is cooked and the



(a) Flat plan in state space

(b) Flat plan execution tree

Figure 9: The optimal solution to the Sussman anomaly, shown as a trajectory through state space and as a plan execution tree.

red region to the states in which b is cooked. It is easy to see that, having achieved one of those conditions, we can stay within that region and achieve the other.

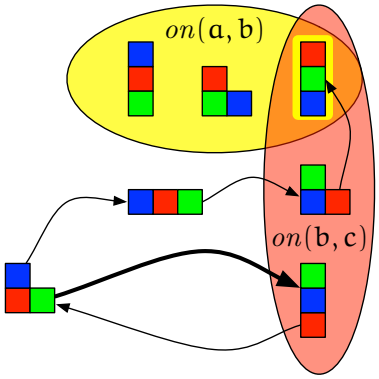
The *Sussman anomaly* [Sussman, 1973] is an example of non-serializable subgoals that was invented to show the shortcomings of the original STRIPS planning algorithm that assumed subgoal serializability. There are three blocks, called a (red), b (green), and c (blue), initially arranged in a configuration with a and b on the table, and c on a, shown in the bottom left of figure 9(a). The optimal plan for this problem is shown in figures 9(a) and 9(b).

What makes this tiny problem interesting is that if we attempt to solve it serially by achieving first one of the conjuncts in the goal, and then the other, we encounter a difficulty. Having achieved one of the subgoals in the most direct way, it is impossible to achieve the second without undoing the first. HPN handles this problem successfully, in the sense that it ultimately achieves the goal, but at the expense of one or two extra actions. This is illustrated in two pairs of figures. Figures 10(a) and 10(b) show what happens when we attempt to achieve $On(b, c)$ first. That subgoal is directly achievable by putting b on c. The configurations in the orange oval correspond to all the configurations in which $On(b, c)$ is true: we can see that there is no trajectory within that subset of configurations that results in the goal. So, when HPN tries to achieve $On(a, b)$, it finds that it has to undo $On(b, c)$ and then re-achieve it. This process is successful because the operators are considered with increasingly more concrete values, and an effective sequence of primitive actions is selected that reach the goal. A similar process, shown in figures 10(c) and 10(d), occurs when we attempt to achieve $On(a, b)$ first.

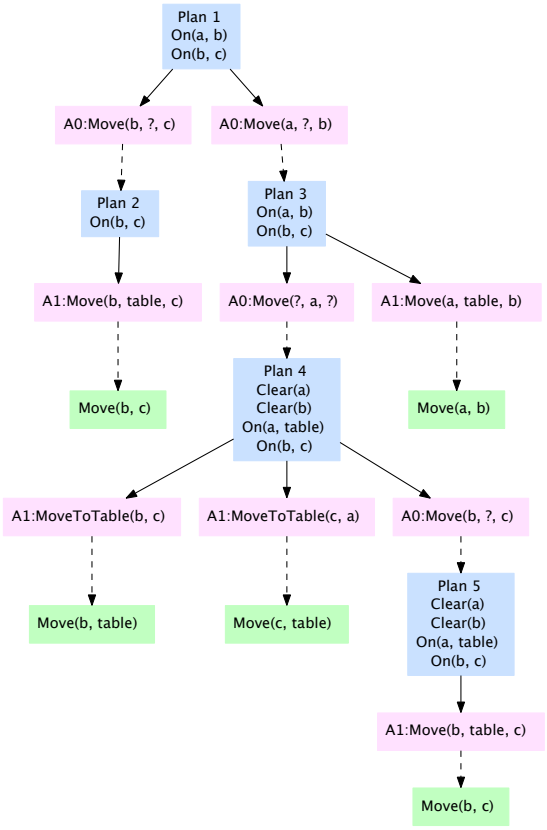
3.2.2 Side effects

An abstract operator may have more "effects" than its entirely concrete version, because it encompasses the actions that are required to establish its preconditions. Thus, any postponed preconditions that are not made false by the action itself are side-effects, as are other potential preconditions of those conditions. Computing these side effects analytically is as hard as planning; it is an important research direction to find methods for learning them. As a preliminary step, we declare them, with different side effects to be declared at different levels of abstraction.

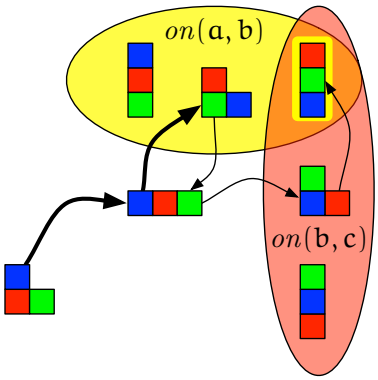
Side effects may be deterministic or non-deterministic. If deterministic, they are treated like regular effects. If non-deterministic (represented by a truth value of NONE), they are treated



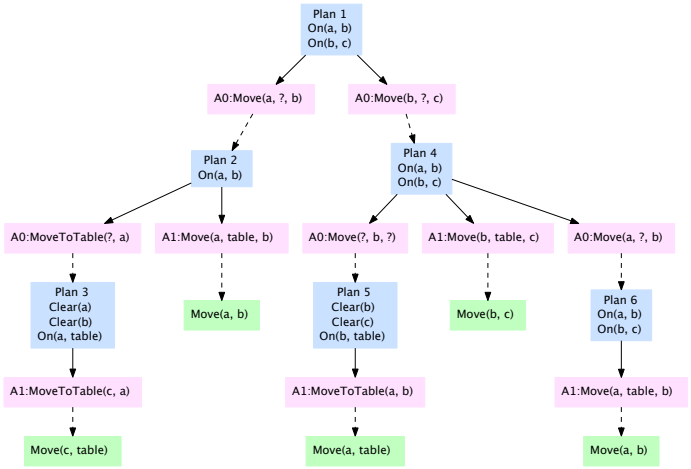
(a) Hierarchical $On(b, c)$ first in state space



(b) Hierarchical: $On(b, c)$ first plan execution tree



(c) Hierarchical $On(a, b)$ first in state space



(d) Hierarchical: $On(a, b)$ first plan execution tree

Figure 10: Hierarchical solutions to the Sussman anomaly using HPN, attempting serialization of the subgoals in both possible orders. Dark arrows show the action(s) taken to achieve the first subgoal; light arrows show the actions taken to achieve the joint goal after achieving the first subgoal.

as being in contradiction with any fluent that has the same predicate and arguments, no matter what the truth value is. They encode the fact that the fluent cannot be relied upon to have any particular value after the abstract operation is executed. Concrete examples of side effects are given in sections 3.3 and 4.1.5.

Failing to declare non-deterministic side-effects does not compromise correctness and completeness of the algorithm, but it may result in poor ordering choices of actions at abstract levels of planning. Declaring many non-deterministic side-effects may result in planning failure at the abstract level: the planner will not be able to find any ordering of the abstract actions that is guaranteed to work. In such cases, an appropriate reaction is to increase the level of abstraction and replan; in the limit at the most concrete abstraction level α_p, a plan will be found if one exists.

3.2.3 Subgoal feasibility

In a regular search with un-abstracted operators, it is not problematic if infeasible goals are generated during the search. They will increase the branching factor and slow the search down, but will not affect the completeness of the method. However, a condition on the completeness of HPN is that any subgoal occurring in a plan at any level of the planning hierarchy be feasible. If an infeasible subgoal is adopted as the goal at a lower level of abstraction, planning will fail, and the algorithm as currently implemented has no recourse. The requirement for subgoal feasibility can be removed if we modify the HPN algorithm so that if a planning failure occurs, the planning process at the level above is repeated, but at a more concrete abstraction level. However, such backtracking will likely incur a significant computational cost; our focus is on exposing enough details at each level of abstraction in planning to complete successfully at every level.

Detecting infeasibility is, in the worst case, as difficult as planning; we concentrate on the weaker condition of logical inconsistency. By logical inconsistency, we mean that the set of fluents cannot all be simultaneously true in the domain; detecting logical inconsistency when the fluents have geometric meaning ultimately requires geometric reasoning. Some inconsistent states are caught by the *contradicts* methods on individual fluents. Other inconsistencies may not be detectable through pairwise operations on fluents: for this reason, we provide the ability to specify a procedure that takes a goal specification as input and decides whether it is consistent.

Of course, the problem of detecting contradictions may also be, in the worst case, as difficult as planning. In section 3.3 we illustrate subgoal feasibility checking in the hierarchical version of the the one-dimensional kitchen environment. In the mobile manipulation environment (see section 4) we ensure feasibility through a combination of pairwise checking among fluents in goals and by having the generators guarantee the feasibility (kinematics, collision-freeness and reachability) of any generated locations and grasps.

3.2.4 Levels of abstraction

For precondition-postponement hierarchies, we now define abstraction levels and the NEXTLEVEL procedure, and show how the abstraction level influences the planning process. Previous work on hierarchical planning based on precondition postponement [Sacerdoti, 1974, Knoblock, 1994] has used static hierarchies that, for example, always postpone consideration of particular fluent types (for example, at one level of abstraction, all *ClearX* preconditions would be ignored, but they would all be considered at the next level). We have developed a more dynamic and flexible strategy for

precondition postponement, which makes the decision about which preconditions to postpone *per ground operator instance*.

An abstraction level α is a mapping from operator instances (that is, operators with variable bindings) to abstraction values, which are non-negative integers. Because there are infinitely many possible instances of any operator schema, the mapping is represented implicitly as a dictionary, such that if an operator ω is not explicitly stored in the dictionary, then $\alpha(\omega) = 0$.

Abstraction level α is lower in the partial order (is less concrete) than β if, for every operator, α specifies a lower or equal abstraction value than β, and if for at least one fluent α specifies a strictly lower abstraction value:

$$\alpha \prec \beta \equiv \forall\omega.\alpha[\omega] \leq \beta[\omega] \;\&\; \exists\omega.\alpha[\omega] < \beta[\omega] \quad .$$

When we plan at a more concrete level of abstraction to achieve a subgoal that was achieved by operator instance ω, we do so with an abstraction level for which the abstraction value $\alpha[\omega]$ is increased by 1, up to a maximum level, at which the operator ω is at its most concrete. The definition of the NEXTLEVEL procedure as well as modifications to the PLAN algorithm to support hierarchy are described in appendix B.2.

3.3 Hierarchical one-dimensional kitchen environment

We explore several different choices of precondition abstraction values, and observe their effect on the resulting planning and execution processes, both in terms of computational effort and (sub)optimality of the resulting execution trajectory. The first case is described in detail in this section, and the others in appendix D.

We will show the abstraction value for each precondition in each operator with a number between brackets after the condition. If we increase the abstraction values for some of the preconditions, in the following definitions and run HPN with the goal $Cooked(a)$ we get the planning and execution tree shown in figure 7.

COOK$((o), s, \gamma)$:
effect: $Cooked(o)$
pre:
$Clean(o)$ **[1]**
$In(o, stove)$ **[2]**

WASH$((o), s, \gamma)$:
effect: $Clean(o)$
pre: $In(o, sink)$ **[1]**

CLEAR$((r, x), s_{now}, \gamma)$:
effect: $ClearX(r, x)$
pre: $In(o, U \setminus r)$ **for** $o \in (allObjects \setminus x)$ **[1]**
sideEffects: for: $o \in possiblyMoved$. $ObjLoc(o, L) =$ NONE

In this model, the COOK operator description has no preconditions with value 0: this is not useful in the current problem instance, but we will see its effect later when there are several things

to be cooked. In addition, in both the COOK and WASH operator descriptions, we have postponed the preconditions that require the object to be located in the relevant place (*sink* or *stove*). Finally, the CLEAR subtask has had all of its preconditions postponed, with value 1. However, we note that, as a side effect, clearing a region might possibly change the object location of any of the objects that are currently in the region, except for those that are required by the goal to be placed at specific locations. The result is that we solve several smaller planning problems, but get an action sequence that is the same as for the original model.

Now, because we are generating subgoals that will serve as goals for planning at the next level of abstraction, most of which are conjunctions of *In* predicates, we must also ensure that they are feasible. We add an extra consistency check to the domain description, which takes a set of constraints that objects either be at particular poses or in particular regions and solves the sub-problem of finding an assignment of poses to the objects that satisfies the constraints or returns failure. This check is used by the APPLICABLEOPS in appendix B.2.

Appendix D shows results for a bigger example in which 5 objects have to be cooked. We would not have been able to solve this problem, which requires 31 primitive actions, with a non-hierarchical formulation. The results show that hierarchical decomposition can yield significant computational improvements without a significant reduction in execution efficiency.

4 Mobile manipulation environment

We demonstrate the HPN planning and execution approach in an apartment environment with a mobile manipulator operating over a long time scale with multiple objects. In this formulation, it is assumed that the environment is deterministic and completely observable. For these reasons, the resulting plans are not feasibly executable on a robot with noisy actions and/or partial observability. The environment consists of a simulated Willow Garage PR2 robot operating in a moderately complex environment, using a single arm, shown in figure 1.

We begin by describing the formulation of the basic robotic pick, place and motor primitives, then we add the formulation of the higher-level cooking and tidying operations. In section 4.2 we present detailed examples of operation in the three-dimensional apartment environment.

4.1 Motion and manipulation

A crucial set of operations for mobile manipulation consists of moving the robot base and arms, picking objects up from supporting surfaces, and placing them back down. We describe here how to formalize this environment for the HPN planning and execution system; it is a significant extension of the formalization of the one-dimensional kitchen environment.

4.1.1 State representation

To support planning and execution, we must represent the underlying state of the environment the robot is interacting with. We assume that all of the objects are unions of convex polyhedra that are extrusions along the z axis (that is, the parts have a constant horizontal cross-section) and that their shapes are known in advance. The world state is not dynamic, in the sense that objects are always at rest unless they are in the robot's gripper and the robot is moving. We assume that all objects that are not in the hand are resting stably on a horizontal surface; this means that their poses can be represented in four dimensions: x, y, z, and θ (which is rotation about the z axis).

A world state, s, is characterized by:

- The current kinematic configuration of the robot, represented by a list of joint angles and base displacements. At higher levels of abstraction, we will consider a restricted Cartesian configuration, which consists of the x, y, θ pose of the robot base, the x, y, z, θ pose of the hand under the assumption that the hand is always rotated so that it is parallel to the ground plane, and the width of the gripper opening.
- The pose of each object, o, written $s[o].pose$.
- The name of the object currently being grasped by the robot (and of any objects that it supports), together with the grasp, which is a pose of the object relative to the hand; this has value *None* if the robot is not currently grasping any object.

We will use $v(o, s)$ as shorthand for $v(o, s[o].pose)$, which is the volume taken up by object o if it were at the pose $s[o].pose$ that it has in state s.

4.1.2 Primitives

From the perspective of the task planner, there are three primitive actions for moving the robot in this environment. In fact, the "primitives" ultimately call a sample-based robot motion planning algorithm to get a joint-space trajectory for the robot, smooth the trajectory, and then execute it. In our current implementation, we use a bi-directional RRT Kuffner and LaValle [2000] to plan base and arm motions inside the primitive.

- PickPrimitive(*obj*, *grasp*) causes the robot to pick up object *obj* using grasp *grasp*, which is specified in terms of a pose for the object relative to the hand of the robot. If the robot is not in a configuration from which the pick operation is possible without moving the base, the primitive fails. This primitive computes a *pre-grasp* configuration of the robot with the hand in front of the object and the gripper open. It then calls the RRT to get a trajectory from the robot's current configuration to the pre-grasp configuration, executes that plan, then moves forward to grasp the object, closes the hand, and then lifts the object up to disengage it from the object it was resting on.
- PlacePrimitive(π) has the robot place the object it is currently holding at the specified pose π. If the robot is not in a configuration from which this is possible without moving the base, the primitive fails. This primitive finds a *pre-place* configuration of the robot with the hand just above the final placement configuration. It then calls the RRT to get a trajectory to the pre-place configuration, executes that plan, moves to the final configuration, releases the grasp, and lifts the hand to disengage from the object.
- MoveRobotPrimitive(*conf*) moves the robot's base and arm to a particular Cartesian configuration, *conf*, specified by a 3D pose for the base, a 4D pose (x, y, z, θ, assuming the hand is held parallel with the floor at the initial and final configurations) for the hand, and a gripper width. It uses an inverse kinematics solver to find a target set of joint angles, plans a path to that pose, and then executes the resulting trajectory.

4.1.3 Fluents

In this section, we provide definitions of the fluents used to formalize the mobile manipulation environment in terms of tests on the state of the world.

- $PoseAt(o, p)$: true if the pose of object o is within δ of pose p, where δ is a constant vector of four tolerance values, one for each dimension:

$$\tau_{PoseAt}((o, p), s) := |s[o].pose - p| \leq \delta \ ,$$

  where the difference is a 4-dimensional vector of componentwise absolute differences (with the angular difference appropriately wrapped), and $\leq$ holds for the vector if it holds for all four components individually.

- $ConfAt(c)$: true if the configuration of the robot is within δ of the configuration c. In this case, it tests to see that both the 3D Cartesian pose of the base and the 4D Cartesian pose of the hand are within δ of those specified in c (in the same manner as for $PoseAt$) and that the actual gripper opening is within δ_g of the gripper opening in c.

- $In(o, r)$: true if object o is entirely contained within a geometric region r:

$$\tau_{In}((o, r), s) := v(o, s) \subseteq r \ .$$

- $ClearX(r, x)$: true if region r is clear of all known objects, except for those in set x:

$$\tau_{ClearX}((r, x), s) := \forall o \notin x.\ (v(o, s) \cap r) = \emptyset \ .$$

  The test is made by checking to see that all objects not in x do not overlap the region r.

- $Holding(o, g)$: true if object o is currently being held by the robot with grasp g:

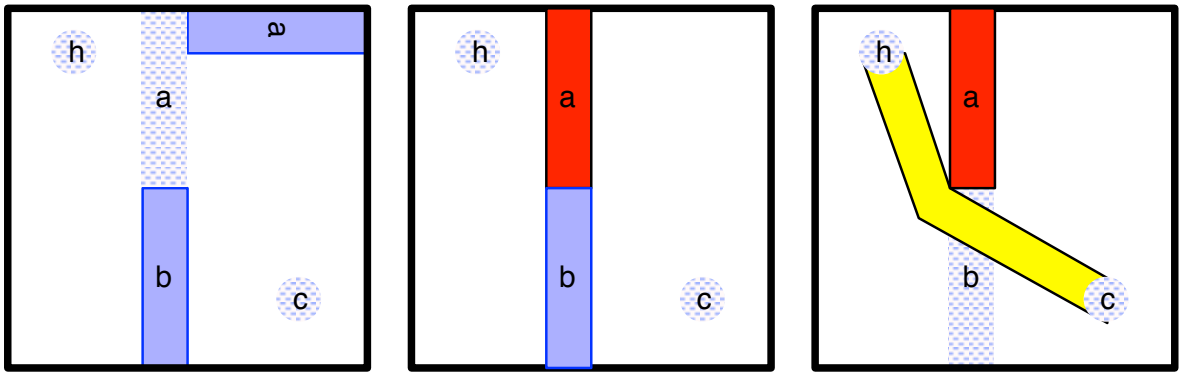
$$\tau_{Holding}((o, g), s) := (o = heldObject(s) \ \& \ |g - heldObjectGrasp(s)| < \delta_g) \ .$$

Entailment and contradiction relationships among these fluents are described in appendix E.1.

4.1.4 Generators

Operations in this domain are parameterized by continuous values including poses and paths. Possible feasible values for these variables are obtained by calling generators, which in this domain are procedures that perform fast, conservative, approximate grasp and motion planning, using a simplified model of the robot and return appropriate poses and swept volumes to construct preconditions of the primitive operators.

Because, in regression-based planning with abstract actions, we do not necessarily know where the robot will be before it moves to pick or place an object, we select a *home* configuration (or region) in the environment and guarantee, during high-level planning, that the robot always has the ability to return to the home configuration. This is sufficient to guarantee that a path for the robot exists to the pre-grasp (or pre-release) pose from any configuration that can also reach the home configuration, and the robot will never block itself in. In the primitives, for efficiency in execution the robot will plan direct paths between locations that will generally not go through



(a) Initial configuration, s_{now} (b) Taboos in red (c) Solution

Figure 11: Simple example illustrating influence of planning-time state s_{now} and goal γ on GENERATEROBOTMOTION. (a) In the initial configuration, the objects a and b are shown in their starting locations in blue, the shaded a region is the target location for a, H is the robot's home configuration and c is the robot's goal configuration. We are trying to find a path for the robot to move to the goal configuration. (b) Because object a is required to be at the location shown in red here, and because in goal regression we are always considering the *last* action in the plan, when the robot is moving to the goal configuration, object a must already be in the target location. Thus, the red volume is a taboo for the path planning of the robot, which must not collide with that volume. (c) The generated robot path is shown in yellow. It is allowed to collide with the current position of object b. This will result in a requirement that somewhere earlier in the plan, the object b be moved out of the way.

the home configuration. This strategy does limit the robot to arrange the movable objects so that the robot is always in the same connected component of the robot's configuration space with the home configuration. This strategy for handling connectivity is not critical to HPN: we have also formalized domains with higher-level connectivity between rooms, and different home locations in each room, which allows multiple different connected components.

The generators use a motion planner that lazily builds a four degree-of-freedom visibility graph [Lozano-Pérez and Wesley, 1979], in which x, y translation constraints are represented as configuration-space polygons for discrete ranges of z and θ. Links in the visibility graph represent pure x, y translation, pure z translation or pure θ rotation. This planner is used to find motions for the base, without moving the arm, and also for Cartesian motions of the hand, without moving the base. It is important to note that, when the primitives are actually called, an RRT is used with an accurate robot model to generate a trajectory for all the degrees of freedom of the actual robot. In our current implementation, the generators return a single choice, which is in no way complete, but is surprisingly effective.

In the one-dimensional kitchen example, we had a single generator, for locations of an object within a region. In the full three-dimensional version, in addition to placements of objects, we also have to select grasps, robot configurations, and paths of the robot. These choices are often quite intricately connected (for example, a placement of an object is only feasible if there are associated grasps and robot configurations that are valid). For these reasons, we have three generators, which build on one another. Internally, there is a great deal more structure.

The simplest generator finds a path for the robot to a desired configuration. It is implemented in the procedure GENERATEROBOTMOTION(c, s_{now}, γ), where c is robot configuration, s_{now} is the current state of the world when the generator is called, and γ is the current subgoal. It finds a path from the home configuration to c, subject to the following constraints and preferences:

- The swept volume of the robot and object along the path **must not** collide with any permanent object or with any object that is placed in a pose required by subgoal γ.
- The swept volume of the robot and object along the path is preferred not to collide with any objects in their current places in state s_{now}.

Figure 11 provides a simple illustrative example, in which the goal is for object a, which is currently horizontal in the upper right corner, to be placed in the shaded region marked a, and for the robot to be in configuration c.

$$\gamma = PoseAt(a, goalA) \ \& \ ConfAt(c) \ .$$

In 11(a) we show the initial configuration, with the robot home configuration marked H and an additional object b. When the generator runs, it extracts *avoid taboos* from the goal: these are regions of space that must be occupied, as required by *PoseAt* fluents. Figure 11(b) shows, in red, the avoid taboo generated by the placement of object a. The generator now tries to find a path from the home configuration H to the goal configuration C. With both the current objects and the taboos, the planning fails. However, it is only a preference to avoid currently existing objects, so the planner is called again, with only permanent and taboo objects, but not the current placements, and a path is found, with the swept volume shown in yellow in figure 11(c).

To continue the example, then, the regression process would generate the subgoal

$$\gamma = PoseAt(a, goalA) \ \& \ ClearX(sweptRobot, \{\ \}) \ ,$$

where *sweptRobot* is the yellow swept volume. If that condition can be established, then it will be possible to achieve the goal, simply by moving the robot to configuration c. Applying the *Clear* operator yields the subgoal

$$\gamma = PoseAt(a, goalA) \ \& \ In(b, warehouse \setminus sweptRobot) \ .$$

Planning would continue, eventually placing a in its target location and moving B away.

To pick up an object, we use the procedure GENERATEPICKPATH$((o, g, l), s_{now}, \gamma)$, where o is an object, g is a grasp of that object (specified as a pose of the object relative to the robot's hand), and l is the pose of the object. It determines a pre-grasp configuration for the robot that is kinematically feasible as well as collision-free. It also computes a path for the base from the home configuration to the pre-grasp configuration and a grasp path for the hand, using the techniques described earlier.

The pick-path generator constructs *avoid taboos* from γ as in GENERATEROBOTMOTION and also prefers not to move through regions of space that are occupied in s_{now}. In addition, it is preferred not to move the robot base from its current configuration in s_{now}.

Slightly more complex is the procedure GENERATEPLACEPATH$((o, r), s_{now}, \gamma)$ where o is an object and r is the region in which o should be placed. Based on the object's current pose, this procedure generates a candidate place motion: it contains a grasp for the object, a pose for the object

within the region, a feasible configuration that the robot should be in before calling PLACEPRIMITIVE, and a path for the base from the home configuration to the pre-place configuration and a place path for the hand and object. In generating motions, the GENERATEPLACEPATH procedure has these constraints and preferences, in addition to those specified for GENERATEPICKPATH.

- If the robot is currently holding the object in grasp g, it should try place the object using the same grasp.
- If the robot is not currently holding the object, then it should try to find a grasp that is feasible both at the object's current pose and at the target pose within the region.

Selecting a pose to place an object requires that:

- There be a grasp of the object and a pose of the object inside the region.
- There is a collision-free *pre-grasp* configuration of the robot from which it can execute the selected grasp of the object.
- There is a path from the home configuration to the pre-grasp configuration.

Within the generators, we sequentially generate grasps, object poses, robot poses and robot paths until we find one that works; this could be done much more efficiently using, for example, the pre-computation methods of Diankov [2010]. We rely on caching to handle multiple requests for similar suggestions efficiently.

When selecting a new pose for an object, we depend on γ in a new way: we define *place taboos* to be regions of space r, such that there is a fluent $ClearX(r, x)$, unless the object under consideration is in x. It would cause an immediate contradiction to place the object in such a region, so they are explicitly excluded during the generation process.

The generators provide a crucial bridge between task planning and motion planning; by computing feasible values for unbound variables with infinite domains, they provide a task-oriented discretization of the infinite geometric and kinematic spaces that underlie robot operation.

4.1.5 Operator descriptions

The formalization of this environment includes three operators corresponding to the three primitive actions (pick, place, and move robot) as well as three operators that are definitional.

4.1.5.1 Pick Following is the formal description of the PICK operation; it is slightly simplified here, omitting some considerations that allow regrasping. It takes two arguments: o is an object to be picked and g is the grasp with which it is to be picked. If the grasp variable is not specified, a value will be selected by a generator (we omit that process in this description, to reduce clutter). The result of this operation is that the robot is holding object o with grasp g.

PICK(o, g, s_{now}, γ):

effect: $Holding(o, g)$

pre: $o \neq None$ **[0]**

choose: $\pi \in \{s_{now}[o].pose, targetPose(\text{GENERATEPLACEPATH}((o, warehouse), s_{now}, \gamma))\}$

choose: $motion \in \text{GENERATEPICKPATH}((o, g, \pi), s_{now}, \gamma)$

$PoseAt(o, \pi)$ **[1]**

$ClearX(sweptVolume(motion), \{o\})$ **[1]**

$Holding(None, None)$ **[2]**

$ConfAt(pregraspConfig(motion))$ **[2]**

sideEffects:

$ConfAt(C) = None$ **[0, 1]**

None [2]

The main section of the operator description is a list of preconditions; each precondition is indicated by a number indicating the abstraction value of that precondition. At the most abstract value, 0, the only requirement is that o not be *None*; that just means that this operation cannot be used to empty the hand.

At the next abstraction value, the operator considers two locations, l, from which the object might be picked up: the first is the pose of the object in the plan-time state, s_{now}, and the second is a temporary location. Picking an object up from another location allows us the option of constructing a plan that is non-monotonic in the sense that it moves the object (at least) twice [Stilman and Kuffner, 2006]: first, from its current pose to a temporary location, and now, from the temporary location to the hand. Swapping the location of two objects illustrates the need for such non-monotonic plans.

For each location, we call GENERATEPICKPATH, which determines a final pre-grasp configuration for the robot and a simplified path through configuration space from the home configuration to the pre-grasp configuration. Given a generated motion, there are several preconditions with different abstraction values. For value 1, we require that the pose of object o be near l because it is to this location that we will guarantee a clear path. We also require that the swept volume of the path be clear, with the exception of object o. If it was impossible to find a path that did not collide with objects in their current poses, then achieving this precondition will require moving objects out of the way.

With abstraction value 2, we require that the robot not be holding anything and that the base pose be near the required pre-grasp configuration of the motion. There are many different ways to organize the precondition hierarchy; this is one version that works effectively.

In addition to preconditions and effects, we also specify side effects, characterizing *possible* effects of abstract versions of the operator. Thus, with values 0 and 1, we assert that any fluent that matches the pattern $ConfAt(C)$, where C is a variable, will be affected unpredictably by this operation: we are not yet sure where the robot will be located. This side effect is not necessary with abstraction value 2, because at that value we have an explicit pre-condition stating what the robot's configuration will be.

There is one more component of this operator description, which is a fluent regression method: it takes a fluent *fl* from the subgoal and returns a new fluent that should replace *fl* in the subgoal when this operator is applied. This definition also refers to the object being grasped, o.

PickRegress(fl):
  if: $fl = ClearX(r, x)$
    return: $ClearX(r, x \cup \{o\})$

In this case, we only change *ClearX* fluents; if a region r was clear except for o and some objects x, then after picking up o, it will be clear except for objects x.

4.1.5.2 Place The place operator is similar to the pick operator. The effect is that the pose of o is near a specified pose.

Place(o, π, s, γ):
  effect:
    $PoseAt(o, \pi)$
  pre:
    choose: $motion \in$ generatePlacePath$((o, v(o, \pi)), s, \gamma)$
    $ClearX(sweptVolume(motion), \{o\})$ **[1]**
    $Holding(o, grasp(motion))$ **[2]**
    $ConfAt(preplaceConfig(motion))$ **[2]**
  sideEffects:
    $ConfAt(C) = None$ **[0, 1]**

Given a pose for an object, the procedure generatePlacePath generates one (or more) candidate place motions. The remaining conditions ensure that the swept volume for the motion is clear, that the robot is holding the object in the correct grasp, and that the robot is in the pre-place configuration specified by the motion. As with the pick operator, the abstract versions of this operator have a non-deterministic side-effect, not knowing where the robot will be at the end. There is no special regression for place.

4.1.5.3 Move robot This operation moves the robot to a desired configuration.

MoveRobot$((c, \delta), s_{now}, \gamma)$:
  effect: $ConfAt(c)$
  pre:
    choose: $motion \in$ generateRobotMotion(c, s_{now}, γ)
    $ClearX(sweptVolume(motion), \{\ \})$ **[1]**

The only requirement is that the swept volume of a path from the home configuration to c be clear of obstacles.

4.1.5.4 Clearing a region To clear a region, we move the objects to an "out of the way" *warehouse* region. In this domain, we use a warehouse as the target, rather than the complement of r, so that we do not have to compute and manipulate non-convex or even non-contiguous regions in three dimensions; but this is an implementation choice and it could be done either way.

ClearX$((r, x), s_{now}, \gamma)$:

effect: $ClearX(r, x)$

pre:

let: $occluders = \{o \mid v(o, s_{now}) \cap r \neq \emptyset\} \setminus x$

$\forall o \in occluders.\ In(o, warehouse \setminus r)$ **[1]**

$ClearX(r, x \cup occluders)$ **[1]**

With the most abstract value, it has no preconditions: we assume we can always make a region clear. At the next abstraction value, it generates a conjunction of requirements that all objects that overlap the region in s_{now} that are not in the set of exceptions be in the part of the warehouse that does not overlap the region being cleared. We also include a condition that there be nothing else in region r except for the exceptions and the set of *occluders* that we just found: this protects against the addition of operations to the plan that would put other objects into this region.

4.1.5.5 Putting an object in a region The condition $In(o, r)$ can be achieved by selecting an appropriate pose p and putting o at pose p. So, this operator reduces an *In* condition to a *PoseAt* condition.

PutIn$((o, r), s_{now}, \gamma)$:

effect: $In(o, r)$

choose: $motion \in$ generatePlacePath$((o, r), s_{now}, \gamma)$

pre: $PoseAt(o, targetPose(motion))$ **[0]**

We make immediate use of this operation when we wish to empty the robot's hands of something. We reduce the condition of not holding anything to the condition of having the currently held object be contained in the warehouse region. Of course, this condition is sufficient, but not necessary, and one could imagine relaxing it, to allow the object to be put down anywhere in the domain that does not violate constraints in the goal.

PutDown(s_{now}, γ):

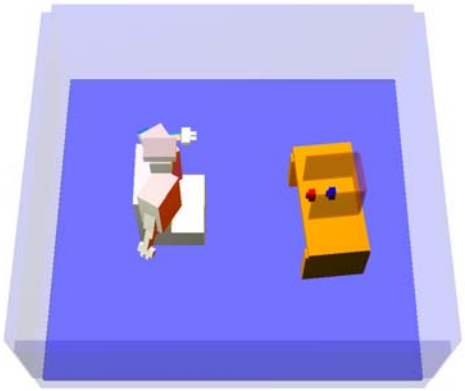
effect: $Holding(None, None)$

pre: $In(currentlyHeldObject(s_{now}),$ warehouse$)$ **[0]**

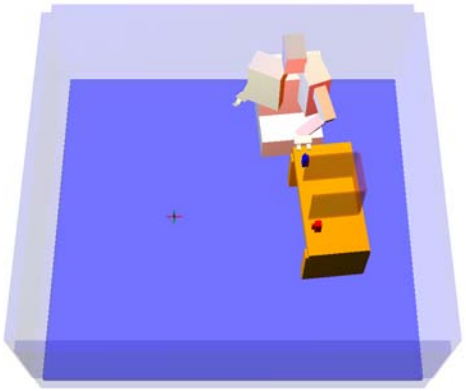
4.1.6 Clearing example

The simple example in figure 12 illustrates these operators with a focus on the role of the swept volumes in the pre-conditions. In this example, the goal is that the blue cup ($cupB$) is *In* a goal region at the left end of the table; the initial and final states are shown in figures 12(a) and 12(b).

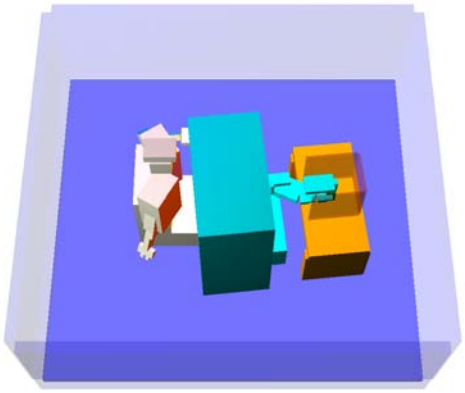
The *PutIn* operator achieves $In(cupB, goalB)$ given that $cupB$ is at an appropriate pose π_B (generated by generatePlacePath, which makes sure that there is a workable grasp and a collision-free configuration for the robot to place it there); the swept volume that arises from a path from the home configuration to the place configuration is shown in figure 12(d). The swept volume is represented by a sequence of instances of the generator's simplified robot model placed along the path; the simplified model is substantially wider than the base of the robot in order to include the space taken up by both arms in their home position.



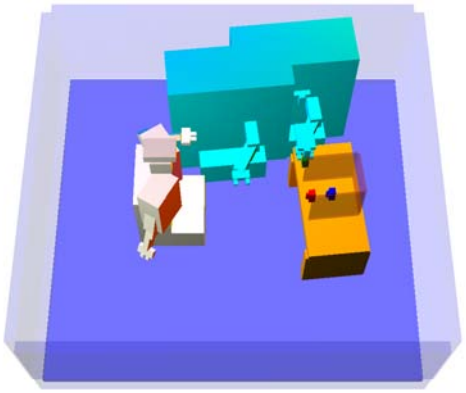
(a) Initial state: Red cup ($cupA$) in front of blue cup ($cupB$).



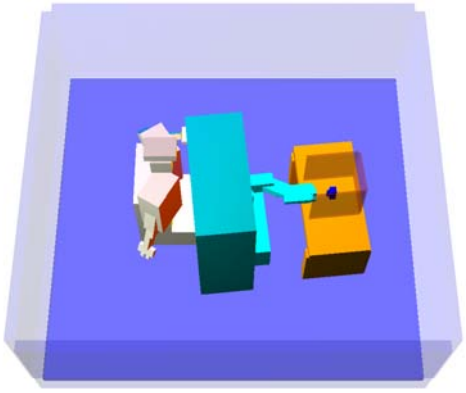
(b) Final state: Blue cup ($cupB$) is on the left of the table.



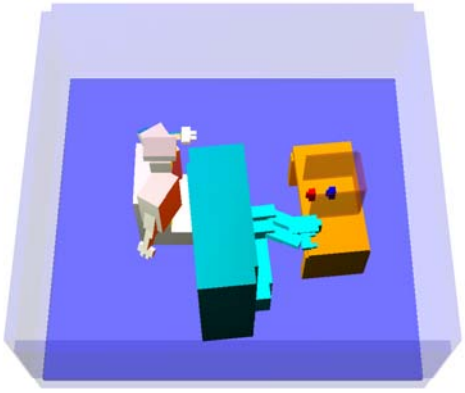
(c) Swept volume to reach the initial location of the blue cup ($cupB$).



(d) Swept volume to reach the final location of the blue cup ($cupB$).



(e) Swept volume to reach the initial location of the red cup ($cupA$).



(f) Swept volume to reach the warehouse location of the red cup ($cupA$).

Figure 12: Swept volumes for clearing example.

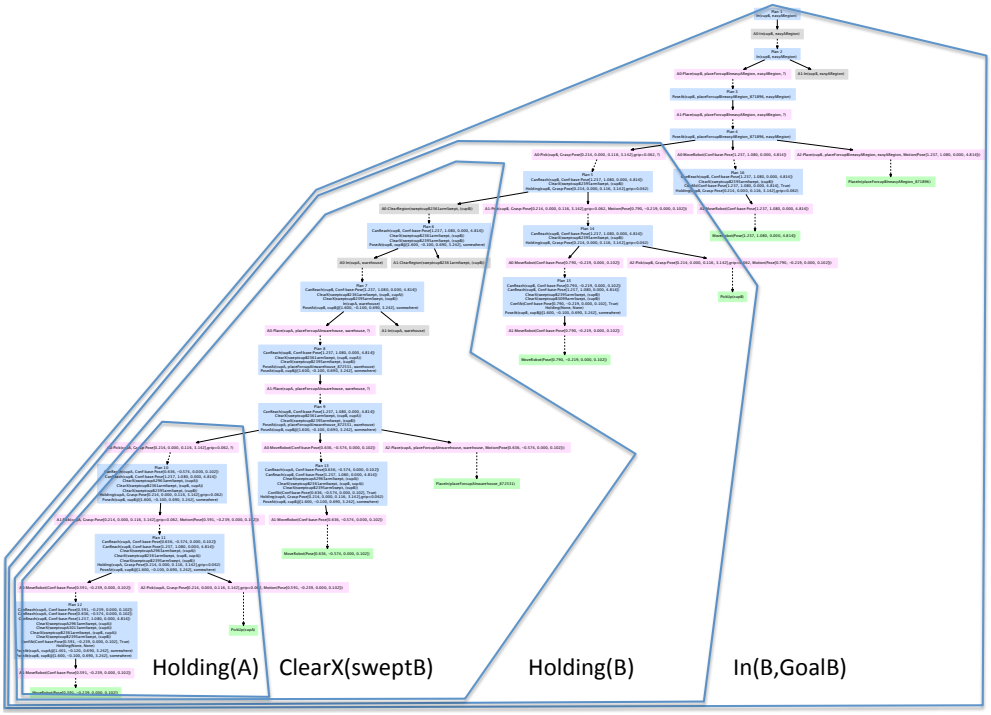


Figure 13: Hierarchical plan and execution tree for the pick example. Each sub-graph outlined in blue corresponds to the process of achieving the fluent written in the bottom of the outlined region.

By executing the *Place* operator, the robot can place the object at the desired pose, but it must be hHolding *cupB* at the appropriate grasp and have the based placed in a configuration that allows the object to be placed using that grasp; the base placement configuration and grasp were obtained from a generator. To ensure that this configuration is reachable, the swept volume in figure 12(d) must be kept clear. To satisfy the *Holding* precondition of *Place*, we use the *Pick* operator, which in turn requires that the robot be not holding anything, that it have its base in a configuration from which the object can be grasped, and that the swept volume for the motion to reach the object be clear. However, this swept volume for the pick operation (figure 12(c)) is not clear; the red cup is in this volume. To achieve the *ClearX* condition, we use the *ClearX* operator, which generates the condition that the red cup be placed in a warehouse. This subgoal is established in a way similar to the original goal; the pick and place swept volumes for the red cup can be seen in figures 12(e) and 12(f). After executing these motions, the swept volumes for picking and placing the blue cup are free of obstruction, and the robot can proceed to grasp it and move it to its goal. The whole planning and execution tree is shown in figure 13, with some salient subgoal structure highlighted.

4.1.7 Swap example

A more complex interaction can be seen in Figure 14. Here we are "swapping" the positions of the two cups shown in figure 14. There are two possible orders for the top-level subgoals, to move the red cup (*cupA*) to its goal position first or to move the blue cup (*cupB*) to its goal position first.

If *cupA* is chosen first, the resulting plan is shown in figure 15(a). Note that the swept volume

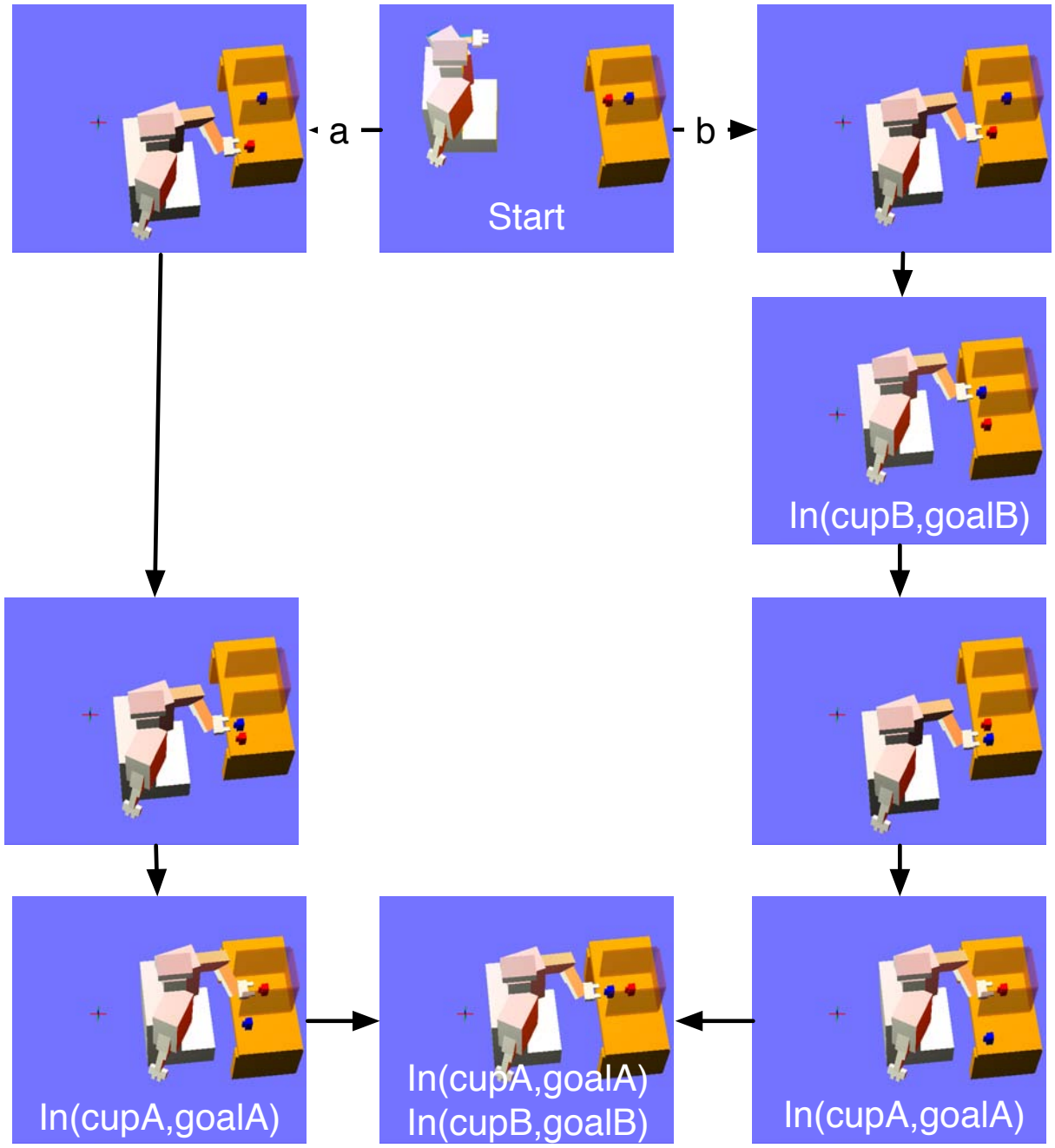
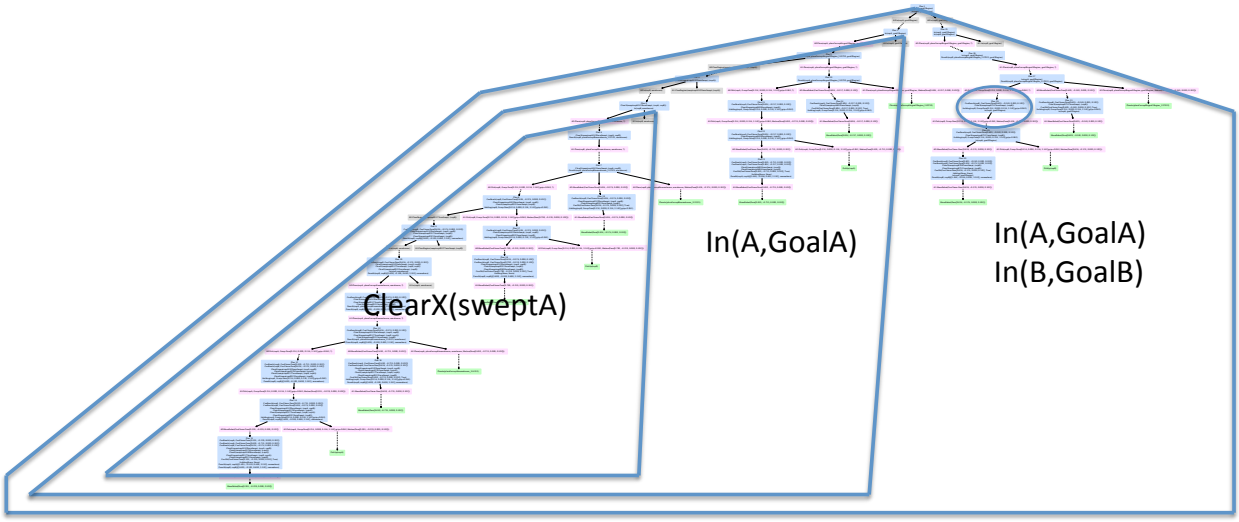
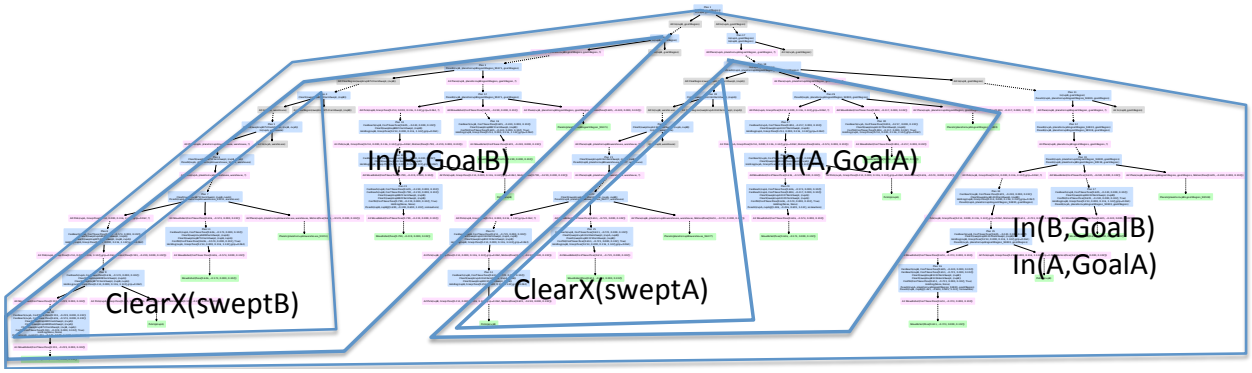


Figure 14: Two execution sequences for swapping *cupA* (red) and *cupB* (blue): (a) Start by achieving $In(cupA, goalA)$ first and (b) Start by achieving $In(cupB, goalB)$ first.



(a) Plan for swapping, moving *cupA* before *cupB*.



(b) Plan for swapping, moving *cupB* before *cupA*.

Figure 15: Plans for swapping *cupA* and *cupB*. (a) Start by trying to achieve $In(cupA, goalA)$ first and (b) Start by trying to achieve $In(cupB, goalB)$ first.

of the *Place* path for *cupA* will overlap *cupB*, so we must move *cupB* to the warehouse, but the swept volume to *Pick* *cupB* will overlap *cupA* and so we must move it to the warehouse. Once both cups are in the warehouse, then we can place them at their intended positions. The ellipse in Figure 15(a) shows the first time we plan to place *cupB* at its destination. At that point, *cupB* has already been moved and placed in the warehouse. This is an example of planning in the now.

If *cupB* is chosen first, the resulting plan is shown in Figure 15(b). Once again, the swept volume of the *Place* for *cupB*, will overlap *cupA*, so we must move *cupA* to the warehouse, but now the path to *cupB*'s goal is free, so the robot places *cupB* at its goal. Now, in preparing to place *cupA* at its goal, we discover that *cupB* is in the way, so we must move it to the warehouse.

The details of these plans would be different depending on different choices of which preconditions to postpone at what level. But even when a poor ordering choice is made, HPN recovers and achieves the goal.

4.2 Cooking and tidying

To motivate and demonstrate mobile manipulation in complex situations, we construct a environment in which the robot must "cook" some food and "tidy" a house. The cooking is highly abstract and none of the subtle details of manipulating non-rigid food objects are addressed. The robot must get food items out of a somewhat crowded "refrigerator," place them in pans and on plates, put pans on a stylized stove, and "wash" objects by putting them in a sink area.

A more careful formalization of this environment could be done by augmenting the planner with more serious temporal planning capabilities; the following is a simple way of demonstrating the ability to handle long-horizon planning in a complex geometric environment. The details of the representation of this environment can be found in Appendix F.

Figure 1 shows a particular simulated instance of this planning environment, in which the robot must plan long sequences of primitive motions that result in high-level logical consequences. There is a shelf on the left side of the kitchen which holds three pans of different sizes and two plates. The refrigerator is on the right-hand side and holds 10 items. There are two "dirty" cups in the environment, one on the sofa table and one on the kitchen table. All the objects have handles to simplify grasping; more complex manipulation strategies are entirely compatible with this planning and execution approach, but have not yet been implemented. There are two "burners" (the red regions on the bench in the center of the figure), and a *sink* (the blue region next to the burners). Each of the food items requires two time units of cooking. Our current planner implementation does not handle parallelism or time constraints and so we do not effectively control actual cooking times; this is a subject for future work.

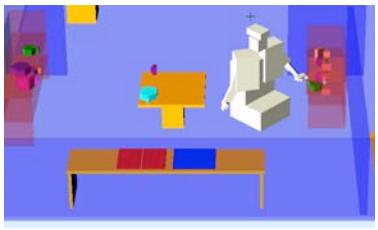
We tested the HPN planner by giving it the following goal:

$$\begin{aligned} &Served(food_1, plate_1)\ \&\ Served(food_2, plate_2) \\ &\ \&\ In(plate_1, regionAbove(kitchenTable))\ \&\ In(plate_2, regionAbove(kitchenTable)) \\ &\ \&\ Tidy(house, (food_1, food_2, plate_1, plate_2)) \end{aligned}$$

To achieve this goal, there are 12 non-motion actions required: two each of PREPARE, START-COOK, COOK , SERVE, and four CLEAN. The minimum number of motions (PICK, PLACE, and MOVEROBOT) to carry out this task is 90, which can be computed by tracking the motions required to move each object along its path. For example, go to pan at shelf, then go with pan to stove, then go with pan to sink and then go with pan to shelf; that is, four MOVEROBOT and three PICK and PLACE pairs. So, the minimal plan length is 102. Note, also, that each of these motions is itself a multi-step trajectory in the configuration space of the robot.

The A* search in the planner implementation is non-deterministic, due to non-determinism in Python's hashing; as a result, different runs produce substantially different plans. As we saw in the case of swapping two cups in section 4.1.7, switching the order of subgoals in the high-level plans can lead to very different geometric situations during planning and very different sequences of primitive actions. In 10 runs of the planner in the cooking and tidying example, we observed an average of 117 primitive actions, with a low of 100 and a high of 130. Thus we see some of the inefficiencies that we expect from our aggressive hierarchy. We'll look at one case in detail—the run that required 130 actions—and point out some of these inefficiencies.

Figure 16 shows several snapshots during the execution of the plan. In figure 16(c) we see that the plan has committed to using the same burner for both cooking operations. When planning to place the second pan in the burner, the previous pan is in collision with the swept volume for



(a) A plate (intended for the first food item) has been placed on the table, and the robot is getting food from refrigerator.

(b) Two objects blocking the second food item have been placed in the warehouse on the right, and the second food item has been prepared and is now being cooked in pan obtained from shelf.

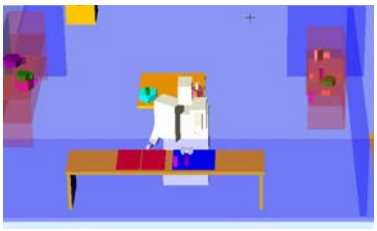
(c) The second food item has been cooked and placed on plate on the table, and the first food item is being cooked using the same burner; the previous pan is in the warehouse on the left.

(d) Starting to tidy up; both pans are in the sink.

(e) Both pans have been put away in the shelf.

(f) In order to enable picking up the pink cup in the right warehouse, the objects in the warehouse are rearranged.

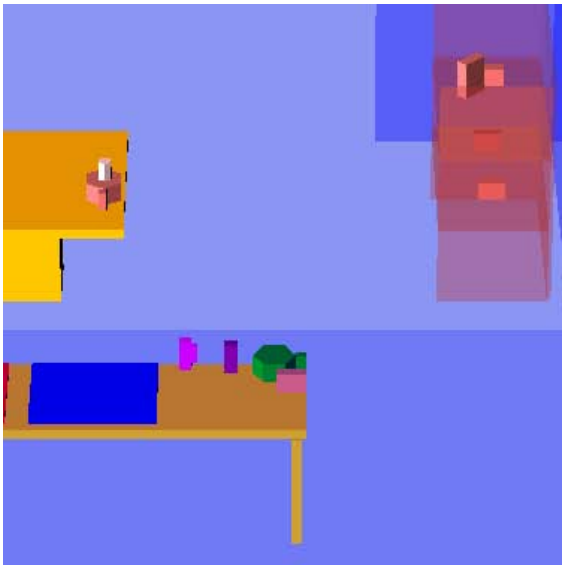
(g) Picking up the pink cup.



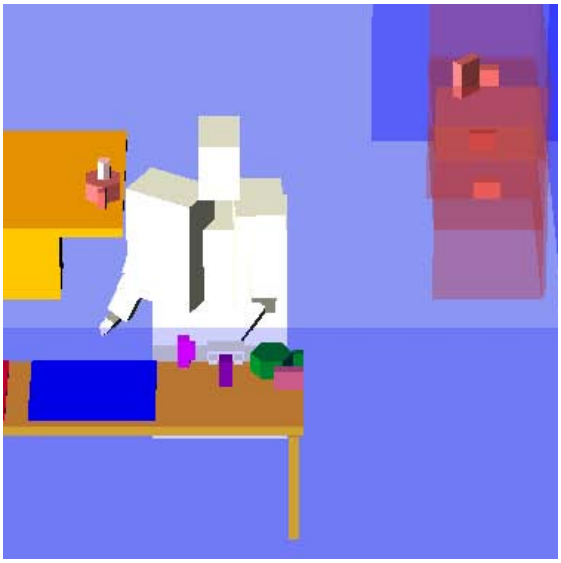
(h) All the objects in the right warehouse have been placed in refrigerator; two dirty cups left out on various tables are in the sink.

(i) The cups have been washed and placed on the shelf.

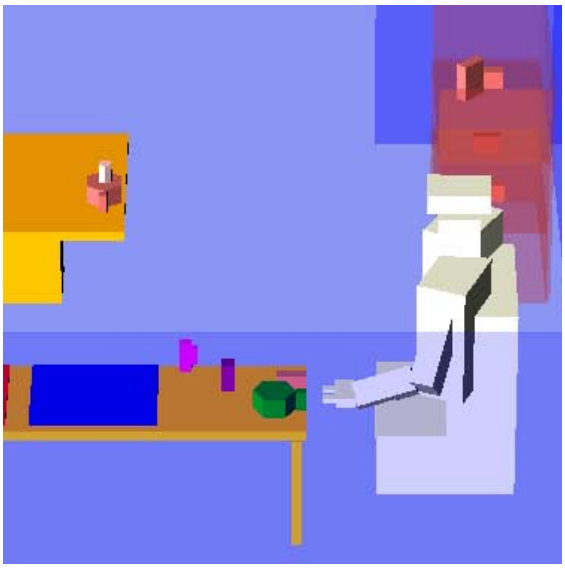
Figure 16: Snapshots for solution in cooking and tidying example.



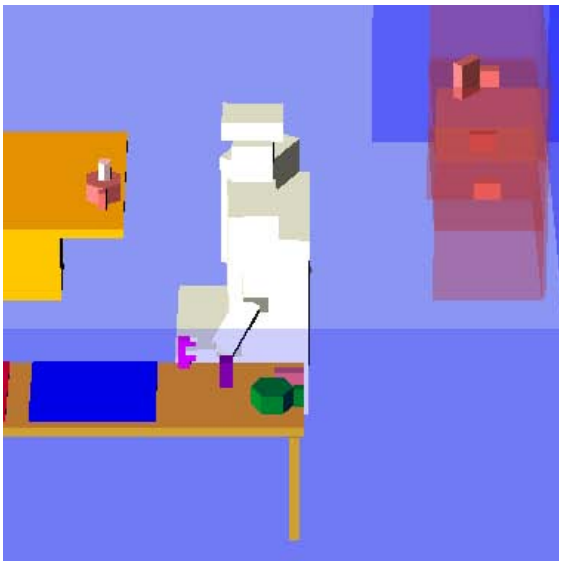
(a) Initial arrangement of objects in the warehouse.



(b) The purple box which is blocking access to the pink cup is moved forward (the chosen location is constrained by a taboo on the placement).



(c) The green pan is moved away from the access to the pink cup (the chosen location is constrained by a taboo on the placement).



(d) The pink cup is successfully grasped.

Figure 17: Moving objects in the warehouse to clear a path for grasp.

placing the pan, so the offending pan is moved to the nearest warehouse. The second pan could have been placed in the free burner or, alternatively, the first pan could have been removed from the burner and placed directly on the sink, since it is dirty and will ultimately need to be moved to the sink.

Figures 16(e) and 16(f) show an interesting sequence. Several items had to be removed from the refrigerator so as to reach the desired food items. These four items were placed in one of the warehouse areas. Now, it's time to put them back in the refrigerator. HPN picks a particular sequence in which the items are to be put back at a fairly high level of the plan, without reasoning about the geometry. The first item chosen is the pink cup, which can only be grasped by its narrow side (it is too wide for the gripper from the other side). Furthermore, the robot (which uses only its left arm) can only reach it from one side. But there are two other objects in the warehouse blocking access. So, HPN moves both objects out of the way, to other places in the warehouse, and then grasps the pink cup. However, it could have taken those objects directly to the refrigerator instead of moving them within the warehouse.

On the whole, these are relatively minor inefficiencies; we believe that a combination of post-processing the plans and learned strategies for detecting and avoiding goal interactions can ameliorate some of these issues. However, because of the worst case computational complexity, one cannot expect optimal solutions, and so some trade-off must be made.

The running time on this cooking/tidying example is on the order of 10 seconds per motion primitive (on a 2.3GHz Intel Core i7); about half of this time (an average of 5 seconds per primitive) goes into the RRT call for the primitive. The complete planning system is implemented in Python; some of the low-level geometric operations have been compiled via Cython, but there remain many opportunities for substantial optimization. However, even with this implementation, the total planning time is on the order of the actual motion time on the robot.

5 Conclusions

We believe that autonomous robots must be able to reason about how to achieve their goals, that is, they must be able to plan. We do not believe that it is feasible to pre-program or to learn a fixed policy for acting in sufficiently complex domains. However, robot planning presents its own set of challenges, notably representing the world state and actions in complex domains, dealing with the computational complexity of planning and coping with the challenges of uncertainty. A number of sub-fields of artificial intelligence and robotics have emerged that address these challenges: symbolic task planning focuses on planning over a wide range of domains and has developed effective techniques for abstraction and factoring to deal with large domains; robot motion planning focuses on planning robot motions in complex geometric domains and has developed effective methods for planning motions for robot with many degrees of freedom; decision-theoretic planning focuses on planning under uncertainty, modeled probabilistically, and has developed representations and algorithms for optimal action. Unfortunately these sub-fields have developed somewhat independently and substantial gaps exist between the representations and algorithms in these separate areas.

The goal of our work is to develop an approach that bridges the gaps between these different styles of planning. In this paper, we have presented an approach to bridging between logic-based task planning and geometry-based motion planning; in [Kaelbling and Lozano-Pérez, 2012], we extend this approach to probability-based planning. The fundamental components of the approach presented in this paper are:

- A tight integration between the logical and geometric aspects of planning. We use a logical representation which includes entities that refer to poses, grasps, paths and regions, without the need for a-priori discretization. Given this representation and some simple mechanisms for geometric inference, we can characterize the pre-conditions and effects of robot actions in terms of these logical entities. We can then reason about the interaction of the geometric and non-geometric aspects of our domains using a uniform mechanism, namely, goal regression (pre-image backchaining).
- An aggressive mechanism for temporal hierarchical decomposition. We postpone the pre-conditions of actions to create an abstraction hierarchy that both limits the lengths of plans that need to be generated and limits the set of objects relevant to each plan.

Both components of our approach trade off some generality for efficiency; our representation of pre-images cannot capture the intricate chaining of geometric conditions required in some puzzle-like problems and the aggressive hierarchy often leads to non-optimal plans. We are comfortable making these trade-offs; although future work should endeavor to ameliorate these shortcomings, we cannot hope to have efficient, optimal solutions to large instances of these planning problems, all of which are fundamentally intractable in the worst-case.

We believe that both tightly integrated logical and geometric representations and hierarchical action decompositions are likely to play central roles in building autonomous robot systems. The detailed mechanisms developed in this paper aim to explore this premise and we believe they show substantial promise. However, the particular methods presented in this paper raise a number of issues that call for further work:

- We assume throughout that there are no irreversible actions, so that an unfortunate choice of action can always be undone. Can this assumption be weakened or removed?
- We endeavor to avoid generating inconsistent subgoals at any level so as to avoid backtracking across levels. What is the performance impact of backtracking across levels?
- We use generators that return one (or a small number) of carefully chosen values for the **choose** variables in the operators. What are the costs and benefits of extending this to a a "complete" sampling strategy?
- We write our operator definitions by hand. Can we learn some aspects of these automatically, for example, the costs and side-effects?

Acknowledgments

We thank all the members of the LIS research group for their help in this project, in particular: George Konidaris, Lawson Wong, Jennifer Barry and Jon Binney for comments on drafts; Ashwin Deshpande and Dylan Hadfield-Menell for help with hierarchical planning; Jennifer Barry for help in testing the planner and with PR2 software; and Caelan Reed Garrett for help with Cython.

This work was supported in part by the NSF under Grant No. 1117325. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also gratefully acknowledge support from ONR MURI grant N00014-09-1-1051, from AFOSR grant AOARD-104135 and from the Singapore Ministry of Education under a grant to the Singapore-MIT International Design Center. We thank Willow Garage for the use of the PR2 robot as part of the PR2 Beta Program.

References

Eyal Amir and Barbara Engelhardt. Factored planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 929–935. Morgan Kaufmann, 2003.

Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1993.

Christer Bäckström and Peter Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proceedings of the Internationl Joint Conference on Artificial Intelligence*, pages 1599–1605, 1995.

Jennifer Barry, Kaijen Hsiao, Leslie Kaelbling, and Tomás Lozano-Pérez. Manipulation with multiple action types. In *International Symposium on Experimental Robotics*, 2012.

Jennifer L. Barry, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. DetH*: Approximate hierarchical solution of large Markov decision processes. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2011.

Michael Beetz, Dominik Jain, Lorenz Mösenlechner, and Moritz Tenorth. Towards Performing Everyday Manipulation Activities. *Robotics and Autonomous Systems*, 58(9):1085–1095, 2010.

Ronen I. Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *In Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pages 809–814, 2006.

Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. The interactive museum tour-guide robot. In *AAAI/IAAI*, pages 11–18, 1998.

Stephane Cambon, Rachid Alami, and Fabien Gravot. A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research*, 28, 2009.

Rosen Diankov. *Automated Construction of Robotics Manipulation Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2010.

Mehmet Dogar and Siddhartha Srinivasa. A framework for push-grasping in clutter. In *Proceedings of Robotics: Science and Systems*, 2011.

C. Dornhege, M. Gissler, M. Teschner, and B. Nebel. Integrating symbolic and geometric planning for mobile manipulation. In *IEEE International Workshop on Safety, Security and Rescue Robotics*, November 2009.

Eric Fabre, Loïg Jezequel, Patrik Haslum, and Sylvie Thibaux. Cost-optimal factored planning: Promises and pitfalls. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, 2010.

Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds., Morgan Kaufmann, 1990.

Eugene Fink and Manuela Veloso. Formalizing the Prodigy planning algorithm. In Malik Ghallab and Alfredo Milani, editors, *New Directions in AI Planning*, pages 261–276. IOS Press, 1996.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning: Theory and practice*. Elsevier, 2004.

Robert P. Goldman. A semantics for HTN methods. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

Karen Zita Haigh and Manuela M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

Kris Hauser and Jean-Claude Latombe. Multi-modal motion planning in non-expansive spaces. *International Journal of Robotics Research*, 29:897–915, 2010.

Kris Hauser and Victor Ng-Thow-Hing. Randomized multi-modal motion planning for a humanoid robot manipulation task. *International Journal of Robotics Research*, 30(6):676–698, 2011.

Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *ICRA*, 2011a.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Pre-image backchaining in belief space for mobile manipulation. In *Proceedings of the International Symposium on Robotics Research*, 2011b.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. Submitted. Draft at `http://people.csail.mit.edu/lpk/papers/HPNBelDraft.pdf`, 2012.

Subbarao Kambhampati, Mark R. Cutkosky, Marty Tenenbaum, and Soo Hong Lee. Combining specialized reasoners and general purpose planners: A case study. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 199–205, 1991.

Thomas Keller, Patrick Eyerich, and Bernhard Nebel. Task planning for an autonomous service robot. In *Proceedings of the 33rd annual German conference on Advances in artificial intelligence*, KI'10, pages 358–365, Berlin, Heidelberg, 2010. Springer-Verlag.

Craig Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68: 243–302, 1994.

Ingo Kresse and Michael Beetz. Movement-aware action control—Integrating symbolic and control-theoretic action execution. In *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA, May 14–18 2012. Accepted for publication.

James J. Kuffner, Jr. and Steven M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

Tomás Lozano-Pérez and M. A. Wesley. An algorithm for planning collison-free paths among polyhedral obstacles. *Communications of the ACM*, 22:560–570, 1979.

Tomás Lozano-Pérez, Matthew Mason, and Russell H. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1), 1984.

Tomás Lozano-Pérez, Joseph L. Jones, Emmanual Mazer, Patrick A. O'Donnell, W. Eric L. Grimson, Pierre Tournassoud, and Alain Lanusse. Handey: a robot system that recognizes, plans and manipulates. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1987.

C.A. Malcolm. The SOMASS system: a hybrid symbolic and behaviour-based system to plan and execute assemblies by robot. In *Proceedings of AISB Conference*, 1995.

Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic semantics for high-level actions. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, 2007.

Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic hierarchical planning: Optimal and online algorithms. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 2008.

Drew McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2), 2000.

Neville Mehta, Prasad Tadepalli, and Alan Fern. Learning and planning with partial models. In *Proceedings of IJCAI Workshop on Learning Structural Knowledge from Observations*, 2009.

Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20: 379–404, 2003.

Nils J. Nilsson. Shakey the robot. Technical Report 323, Artificial Intelligence Center, SRI International, Menlo Park, California, 1984.

Illah Nourbakhsh. Using abstraction to interleave planning and execution. In *Proceedings of the Third Biannual World Automation Congress*, 1998.

Erion Plaku and Gregory Hager. Sampling-based motion planning with symbolic, geometric, and differential constraints. In *Proceedings of the IEEE Conference on Robotics and Automation*, 2010.

Erion Plaku, Lydia E. Kavraki, and Moshe Y. Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. *IEEE Transactions on Robotics*, 26(3):469–482, 2010.

Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 1974.

Thierry Siméon, Jean-Paul Laumond, Juan Cortés, and Anis Sahbani. Manipulation planning with probabilistic roadmaps. *International Journal of Robotics Research*, 23(7–8):729–746, 2004.

Siddharth Srinivasa, Dave Ferguson, Casey Helfrich, Dmitry Berenson, Alvaro Collet, Rosen Diankov, Garratt Gallagher, Geoffrey Hollinger, James Kuffner, and Michael Vande Weghe. HERB: A home exploring robotic butler. *Journal of Autonomous Robots*, 2009.

Mike Stilman and James J. Kuffner. Planning among movable obstacles with artificial constraints. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics (WAFR)*, 2006.

Mike Stilman, Jan-Ulrich Schamburek, James J. Kuffner, and Tamim Asfour. Manipulation planning among movable obstacles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2007.

Freek Stulp, Andreas Fedrizzi, Lorez Mösenlechner, and Michael Beetz. Learning and reasoning with action-related places for robust mobile manipulation. *Journal of Artificial Intelligence Research*, 43:1–42, 2012.

Gerald J. Sussman. *A Computational Model of Skill Acquisition*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1973.

Jur P. van den Berg, Mike Stilman, James Kuffner, Ming C. Lin, and Dinesh Manocha. Path planning among movable obstacles: A probabilistically complete approach. In *WAFR*, pages 599–614, 2008.

M. Westphal, C. Dornhege, S. Wölfl, M. Gissler, and B. Nebel. Guiding the generation of manipulation plans by qualitative spatial reasoning. *Spatial Cognition and Computation: An Interdisciplinary Journal*, 11(1):75–102, 2011.

Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling*, 2010.

Franziska Zacharias and Christoph Borst. Knowledge representations for high-level and low-level planning. In *Proceedings of the scheduling and planning applications workshop at the International Conference on Automated Planning and Scheduling*, 2011.

A Inference methods for one-dimensional kitchen environment

To characterize conditions under which fluents entail and contradict one another in the one-dimensional kitchen environment, we must define some terms. The function $v(o, l)$ takes an object o and a location l and returns a *volume*, which is the region of space occupied by o when it is placed at l. The function $f(o, r)$, where o is an object and r is a region made up of union of contiguous regions, returns TRUE if o *fits* in r: that is, if there is a location l such that $v(o, l) \subseteq r$. We will also use $f(\{o_1, o_2\}, r)$ to mean that objects o_1 and o_2 fit into r simultaneously and without overlapping: that is, that there exist locations l_1 and l_2 such that $v(o_1, l_1) \cap v(o_2, l_2) = \emptyset$, $v(o_1, l_1) \subseteq r$, and $v(o_2, l_2) \subseteq r$.

The following table includes the conditions under which a fluent f_1 entails fluent f_2. There are no extra entailment inferences to be made from *Cooked* and *Clean* fluents. This is not an exhaustive set: for instance, in a world with a known finite number of objects, a set of fluents encoding the poses of those objects entails an infinity of *ClearX* fluents. The entailment conditions shown below are sufficient for effective planning in this environment.

f_1	entails f_2		
	$ObjLoc(o_2, l_2)$	$In(o_2, r_2)$	$ClearX(r_2, x_2)$
$ObjLoc(o_1, l_1)$	$o_1 = o_2$ & $\|l_1 - l_2\| < \delta$	$o_1 = o_2$ & $v(o_1, l_1) \subseteq r_2$	
$In(o_1, r_1)$		$o_1 = o_2$ & $r_1 \subseteq r_2$	
$ClearX(r_1, x_1)$			$x_1 \subseteq x_2$ & $r_2 \subseteq r_1$

The following table includes the conditions under which a fluent f_1 *contradicts* fluent f_2. There are no extra contradiction inferences to be made from *Cooked* and *Clean* fluents. Because contradiction is symmetric, we only include the upper triangle of this table.

f_1	contradicts f_2		
	$ObjLoc(o_2, l_2)$	$In(o_2, r_2)$	$ClearX(r_2, x_2)$
$ObjLoc(o_1, l_1)$	$(o_1 = o_2$ & $\|l_1 - l_2\| > \delta))$ || $(o_1 \neq o_2$ & $v(o_1, l_1) \cap v(o_2, l_2) \neq \emptyset)$	$(o_1 = o_2$ & $v(o_1, l_1) \not\subseteq r_2)$ || $(o_1 \neq o_2$ & $\neg f(o_2, r_2 \setminus v(o_1, l_1)))$	$o_1 \notin x_2$ & $v(o_1, l_1) \cap r_2 \neq \emptyset$
$In(o_1, r_1)$	symmetric	$(o_1 = o_2$ & $\neg f(o_1, r_1 \cap r_2))$ || $(o_1 \neq o_2$ & $\neg f(\{o_1, o_2\}, r_1 \cup r_2))$	$o_1 \notin x_2$ & $\neg f(o_1, r_1 \setminus r_2)$
$ClearX(r_1, x_1)$	symmetric	symmetric	

We have the following fluent regression method, for the PickAndPlace operator. The use of this method during planning is described in appendix B.1.

MoveRegress(o, l, fl, s_{now}):
  if $fl = ClearX(r, x)$:
    if $o \in x$:
      return fl
    elseif $r \cap v(o, l) \neq \emptyset$:
      return False
    else
      return $ClearX(r, x \cup \{o\})$

It encodes the fact that, if we are placing an object o at location l, it might be that we are also either placing it in or removing it from the region r and therefore possibly affecting a *ClearX* fluent in the goal. If object o was already in the exception set x, then this action will not affect the truth value of that fluent. Otherwise, if the region of space occupied by o when placed at l overlaps r, then this action will violate the *ClearX* condition in the goal, so the regression is False, indicating a contradiction. If neither of these conditions holds, then we are removing object o from region r, and so it is permissible for o to be in the exception set x on the previous step; as a result, the regression is $ClearX(r, x \cup \{o\})$.

B Planning algorithms

B.1 Regression

The plan procedure takes as input the current world state s_{now}, a goal γ, and an abstraction level α. It reduces directly to a call to the A^* search algorithm. Recall that regression is a search in the space of subgoals, sg, which are sets of fluents that denote sets of world states, and the search

starts from the goal γ, also a set of fluents that denotes a set of world states. The arguments to A^* are as follows:

- *start*: the search starts with the goal γ;
- *goal test*: a function that maps a subgoal *sg* generated during the search to TRUE if the current world state s_{now} is contained in the set of world states denoted by that subgoal. The search terminates when this function returns TRUE;
- *successor function*: a function that maps a subgoal *sg* generated during the search into a list of successor subgoals to be considered by the search. Each new subgoal is the pre-image of the current subgoal under an applicable operation, where Ω is a set of operator descriptions; and
- *heuristic function*: a function that maps a subgoal *sg* generated during the search to an estimate of the cost of the least-cost path (a sequence of operations) from that subgoal to one that satisfies the goal test; in this case, we use a heuristic that counts the number of fluents in subgoal *sg* that are not true in s_{now}.

PLAN$(s_{now}, \gamma, \alpha)$:
  return A*SEARCH(γ,
    $\lambda sg.s_{now} \in sg$,
    $\lambda sg.$APPLICABLEOPS(sg, Ω, α)
    $\lambda sg.$NUMFLUENTSNOTSATISFIED(s_{now}, sg))

The following procedure computes successors of a goal γ in the search (which are pre-images of γ under various operator applications). It first considers each fluent f in γ, and each possible effect r of each operator $O \in \Omega$. If there is a way to bind the variables in f and r so that they match, then we let ω be a version of the operator with its variables bound so that it can achieve f. For each such binding of the effect variables, there may be several different ways the operator can achieve the effect: these are generated by calling $\omega.operatorInstances(\alpha)$, which yields a list of 5-tuples. Each tuple contains: a list of preconditions, a list of effects, a cost, a fluent regression method, and possibly some new bindings (we discuss in section 4.1.4 why new bindings may come about). Finally, we compute the regression of γ under these preconditions, effects, and regression function, and generate a new search node containing the operator, new bindings, cost, and subgoal.

APPLICABLEOPS(γ, Ω, α):
  $ops = []$
  for $f \in \gamma$
    for $O \in \Omega$
      for $r \in effects(O)$
        $\omega =$ APPLYBINDINGS$(O,$ UNIFY$(f, r))$
        for $(preConds, effects, cost, frf, newB) \in \omega.operatorInstances(\alpha)$
          $sg =$ REGRESS$(g, preConds, effects, frf)$
          if sg: $ops.append(Op(O, newB, cost, sg))$
  return *ops*

Here, the UNIFY procedure takes two expressions containing constants and variables and returns a binding of variables to constants that makes the two expressions equal, if one exists. In the simple discrete case with finite domains, the *regress* procedure is a straightforward set-theoretic operation:

$$\text{REGRESS}(g, \mathit{preConds}, \mathit{effects}) = (g \setminus \mathit{effects}) \cup \mathit{preConds} \quad .$$

The regression of a goal g, represented as a set of fluents, is computed by removing the effects of the operation (because the operation will make them true, they need not be true in the state before performing the operation) and then adding the preconditions (which must be true before the operation is executed).

Because we are operating in an infinite domain, we cannot list all of the actual effects on an operation. Instead we assert a minimal set of effects and extend the REGRESS procedure to do extra reasoning about entailment, contradiction, and regression of individual fluents, as follows:

REGRESS($g, \mathit{preConds}, \mathit{effects}, rf$):

```
sg = []
for f ∈ g
    if contradicts(effects, f): return FALSE
    elif entails(effects, f): pass
    else sg.conjoin(rf(f))
for f ∈ preConds
    sg.conjoin(f)
return sg
```

We start with an empty result, sg, and begin by going through each fluent f in the goal g that we are regressing. If f is in contradiction with the effects of the operation, then the regression result is FALSE: there are no previous states from which this operation would make the goal true. If f is entailed by the effects of the operation, then we don't need to add it to SG, because this operation makes it true, so it need not be true in the previous step. Otherwise, we conjoin the fluent f into SG. Finally, we conjoin each of the preconditions into SG and return it.

It remains to define the *conjoin* operation, which computes the conjunction (the "and" operation) on an existing conjunction of fluents, c, and a new fluent f. It would be technically correct to just add the fluent f to the collection of fluents c, but for efficiency reasons during planning, we want to detect as soon as possible whether f is in contradiction with c and/or whether c already entails f, which case it need not be added into the result.

We go through each fluent f' in c: if we find a fluent in c that entails f, then there is no need to add f to the conjunction, and we just return c unchanged; if f entails a fluent that already exists in c, then we can remove it; if f contradicts any fluent in c, then the resulting conjunction is false. Finally, we return the remaining conjuncts, plus f, as the result.

CONJOIN(c, f):

```
r = copy(c)
for f' ∈ c
    if entails(f', f): return c
    elif entails(f, f'): r = r \ {f'}
    elif contradicts(f, f'): return FALSE
return r ∪ {f}
```

B.2 Hierarchical planning

To support planning in the context of the HPN algorithm, we need to add a new procedure and change an existing one.

Inside HPN we must be able to increment the abstraction level α along the dimension described by operator ω, as follows:

```
NEXTLEVEL(α, ω)
    α' = copy(α)
    α'[ω] = α'[ω] + 1
    return α'
```

Because this procedure is only called by HPN when ω is not a primitive, the abstraction value for ω in α is guaranteed to be less than its maximum value, so it is safe to increment, and it is guaranteed that $\alpha \prec \alpha'$.

When the PLAN procedure is called by HPN with argument α for the abstraction level, it needs to call a modified version of APPLICABLEOPS, as follows:

```
APPLICABLEOPS(γ, Ω, α, consistent):
    ops = [ ]
    for f ∈ γ
        for O ∈ Ω
            for r ∈ effects(O)
                ω = APPLYBINDINGS(O, UNIFY(f, r))
                for (preConds, effects, cost, frf, newB) ∈ ω.operatorInstances(α)
                    absPreConds = [c for c in preConds if c.absLevel ≤ α[f]]
                    sg = REGRESS(g, absPreConds, effects, frf)
                    if sg and consistent(sg): ops.append(Op(O, newB, cost, sg))
    return ops
```

The two critical changes are a mechanism for only taking into account the preconditions that are appropriate for the abstraction level α and performing a consistency test on sg. The consistency test is part of the domain description; if a mechanism is used to backtrack across abstraction levels then it is not critical to test for consistency.

In some environments, it is most effective to make the hierarchies somewhat shallower by adding a mechanism whereby if a fluent with predicate p is used within a plan with abstraction value l, then any other operations to achieve that fluent, within the same call to PLAN, will be applied with an abstraction value no more abstract than l.

C Completeness of HPN

In the following, we will restrict our attention to finite domains, in which there is a finite universe of constants that can be used to instantiate the arguments of the operator schemas, which means there are finitely many ground operators, ground fluents, and world states. Including enough definitions to make this argument completely formal is outside the scope of this paper; we hope it is clear enough to be intuitive to the reader.

Following [Ghallab et al., 2004], we define a (classical) planning domain.

Definition 1. *A* finite planning domain *is a structure* $\langle \mathcal{L}, S, A, ns \rangle$*, where*

- $\mathcal{L}$ *is a first-order language with finitely many predicate symbols and constant symbols and no function symbols;*
- *the state space* S *is a (possibly proper) subset of all ground atoms of* $\mathcal{L}$*;*
- *the action space* A *is all ground instances of operators, each of which is characterized by a set of ground preconditions and a set of ground effects;*
- *the state transition function* ns *that characterizes the state* s' *that results from taking action* $a \in A$ *in state* $s \in S$*:* $s' = ns(s, a)$*.*

Definition 2. *A* finite hierarchy *for* HPN *is a structure* $\langle \mathcal{A}, \prec, \alpha_0, \alpha_p, \text{NEXTLEVEL}, \mathcal{M}, \text{ISPRIM} \rangle$*, where*

- $\mathcal{A}$ *is a finite set of abstraction levels;*
- $\prec$ *is a partial order on abstraction levels, so that if* $\alpha \prec \beta$*, we will say that* α *is less concrete than* β*;*
- $\alpha_0 \preceq \mathcal{A} \preceq \alpha_p$ *for all* $\alpha \in \mathcal{A}$*;*
- NEXTLEVEL *is a procedure that maps an abstraction level and an operator into a new (more concrete) abstraction level:*

  $$\alpha \prec \text{NEXTLEVEL}(\alpha, o) \ ,$$

  for all abstraction levels α *and operators* o*;*
- $\mathcal{M}$ *is a set of finite planning domains indexed by* $\mathcal{A}$*, so that* $\mathcal{M}(a)$ *for* $a \in \mathcal{A}$ *is a finite planning domain; these domains all share the same language* $\mathcal{L}$*;*
- ISPRIM *is a procedure mapping all actions* a *in any planning domain to* TRUE *or* FALSE*; for all* a_p *in the action set of* $\mathcal{M}(\alpha_p)$*,* $\text{ISPRIM}(a) = \text{TRUE}$*;*
- *Whenever* $\text{ISPRIM}(a) = \text{TRUE}$ *for any* a *in any planning domain in* $\mathcal{M}$*, the preconditions and effects of* a *constitute a correct and complete formalization of the primitive operation in the domain.*

Definition 3. *A finite planning component* (M, s_0)*, where* M *is a finite planning domain and* s_0 *is a state, is* reversible *iff for all* $s \in M.S$ *and* $\gamma \in 2^{M.S}$*, if* γ *is reachable in* M *from* s_0 *and* s *is reachable in* M *from* s_0*, then* γ *is reachable in* M *from* s*.*

Definition 4. *A finite hierarchy is* complete *iff for any starting state* s_0 *and goal* g*, if there exists a solution for planning problem* $(\mathcal{M}(\alpha_p), s, g)$ *then there exists a solution for* $(\mathcal{M}(\alpha), s_0, g)$ *for all* $\alpha \in \mathcal{A}$*.*

Definition 5. *A finite hierarchy is* sound *iff, for any sound and complete planning algorithm* PLAN*, start state* s_0*, and goal* γ*, if*

$$((-, g_0), (\omega_1, g_1), ..., (\omega_n, g_n)) = \text{PLAN}(s_0, \gamma, \mathcal{M}(\alpha)) \ ,$$

then for all i*,* g_i *is reachable in* $\mathcal{M}(\alpha_p)$ *from* s_0*. That is, none of the subgoals in any plan at any level are infeasible.*

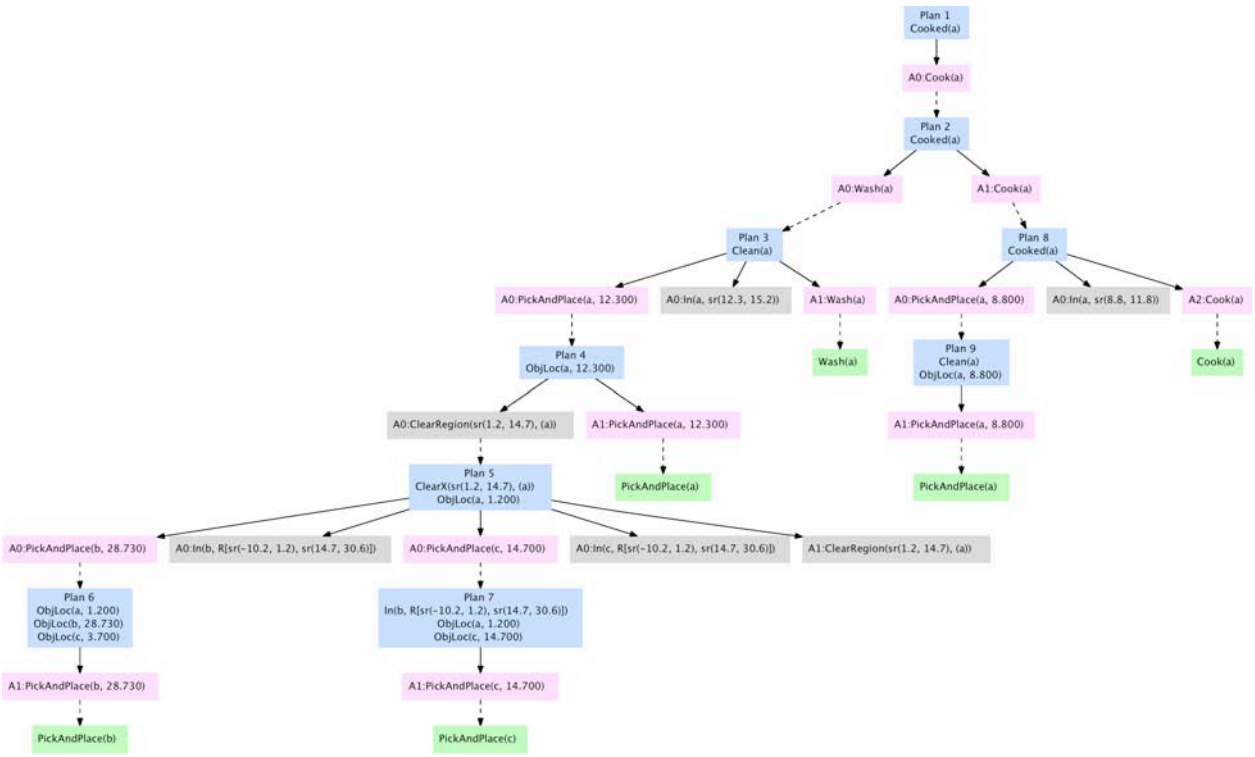


Figure 18: Deeper hierarchical planning and execution trees for cooking object a. This is a solution to the same problem as the flat plan shown in figure 6.

Theorem 1. *If finite hierarchy H and* PLAN *are sound and complete, $(\mathcal{M}(\alpha_p), s_0)$ is reversible, and some state $s_\gamma \in \gamma$ is reachable from s_0, then* HPN(s_0, γ, H) *will terminate with the world in some state $s \in \gamma$.*

Proof. By induction on α.

Base case: $\alpha = \alpha_p$. By the soundness and completeness of PLAN and the last clause in definition 2, which ensures that $\mathcal{M}(\alpha_p)$ is a correct formalization of the domain, executing the sequence of primitive ω_i in the plan p will drive the world into some state $s_g \in \gamma$.

Inductive case: Assume the result holds for all $\alpha \prec \alpha_i$. Let

$$p = \text{PLAN}(s_0, \gamma, \alpha_i) = ((-, g_0), (\omega_1, g_1), ..., (\omega_n, \gamma)) \ .$$

For every step j in p, either ω_j is a primitive or not. During the execution of p, let s_j be the world state in g_j that actually holds. If ω_j is a primitive, then by the soundness of PLAN and the primitive models, if the world was in a state $s_{j-1} \in g_{j-1}$ before executing ω_j it will be in a state $s_j \in g_j$ after executing ω_g. If ω_j is not a primitive, then by the inductive hypothesis, and because NEXTLEVEL$(\alpha_i, \omega_j) \prec \alpha_i$, after a call to HPN$(s_{j-1}, g_j,$ NEXTLEVEL$(\alpha_i, \omega_j), \textit{world})$ the resulting world state s_j will be in g_j. So, after all steps in p, the world state s_n will be in γ. □

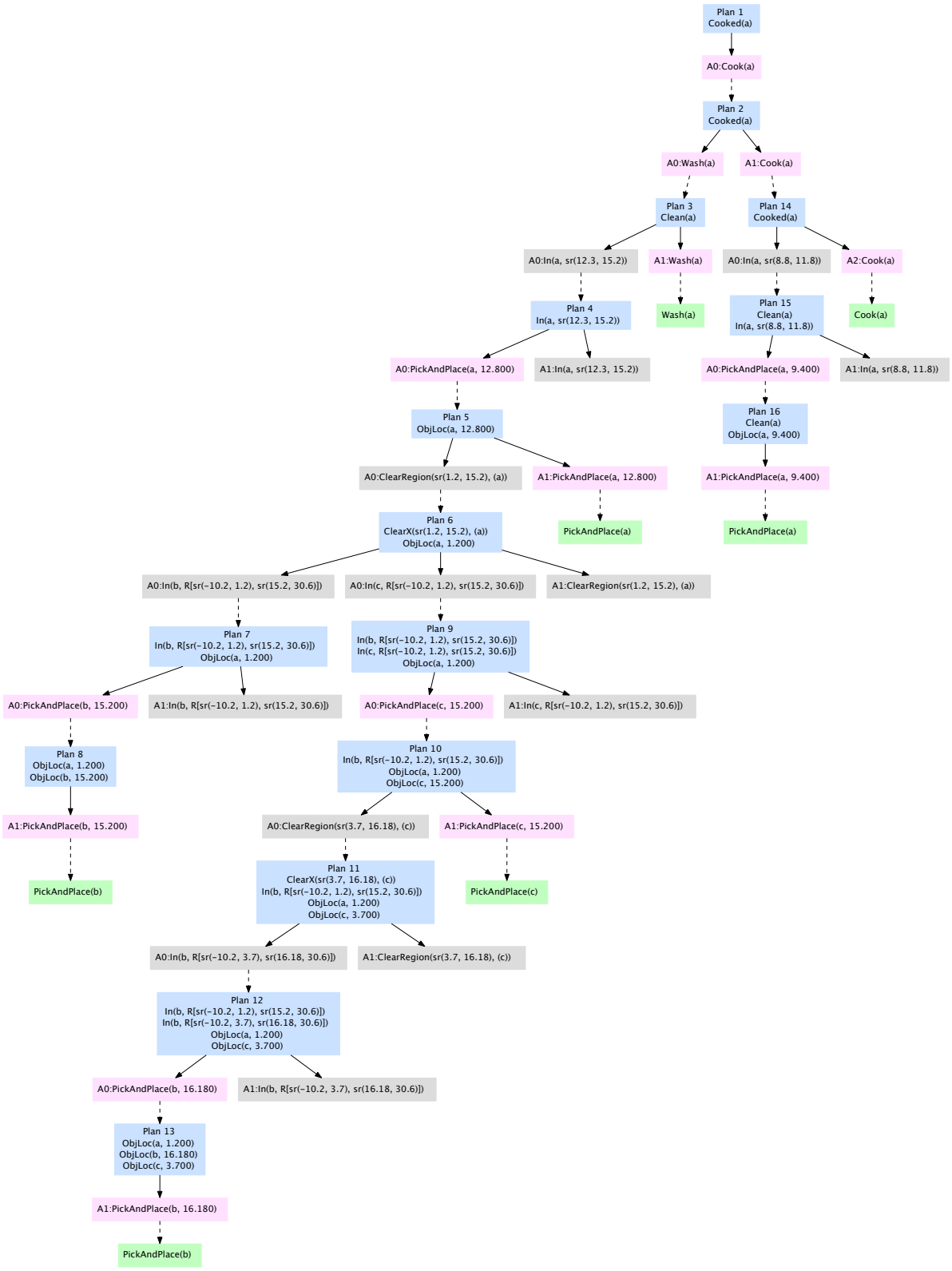


Figure 19: Deepest hierarchical plan for cooking object A. This is a solution to the same problem as the flat plan shown in figure 6

.

D Additional results in hierarchical kitchen environment

D.1 Increasing abstraction

If we assign different abstraction values to the preconditions, shown below, we get the execution tree shown in figure 18 which has 9 planning problems with one solution of length 5, two of length 3, two of length 2, and four of length 1.

PickPlace$((o, l_t), s_{now}, \gamma)$:
effect: $ObjLoc(o, l_t)$
choose: $l_s \in \{s_{now}[o].pose,$ generateLocsInRegion$((o, warehouse), s_{now}, \gamma),$
generateLocsInRegion$((o, stove), s_{now}, \gamma),$
generateLocsInRegion$((o, sink), s_{now}, \gamma)\}$
pre:
$ObjLoc(o, l_s)$ **[0]**
$ClearX(sweptVol(o, l_s, l_t), \{o\})$ **[1]**

The difference is that we have postponed the *ClearX* precondition on the swept volume for the PickPlace operator.

If we postpone even more preconditions, we get the execution tree shown in figure 19 which has 14 planning problems, the longest of which is length 3. Although this structure is larger, the planning process is much faster than in the formulations with less hierarchy.

In(o, r, s_{now}, γ):
effect: $In(o, r)$
choose: $l \in$ generateLocsInRegion$((o, r), s_{now}, \gamma)$
pre: $ObjLoc(o, l)$ **[1]**

We have now postponed the *ObjLoc* precondition for making *In* true, which means the planner does not select the particular location where an object will be located within a region until a more concrete level of planning. The execution tree shows that this degree of hierarchicalization leads to suboptimality in the plan: plan 6 decides to put objects b and c in the same region of space, either between -10.2 and 1.2 or between 15.2 and 30.6. Object b is considered first, and is placed at location 15.2. Next, object c is considered, and location 15.2 is chosen for it, as well. So, in the process of placing c, it is necessary to move b again, resulting in an extra step. This problem could easily have been avoided by looking at s_{now} when selecting a placement for c. But as we consider larger and larger environments, these minor inefficiencies will be inevitable.

D.2 Five objects

Now consider a one-dimensional kitchen example, as in Section 3.3, but with 5 objects, each of which must be cooked. The high-level decisions about ordering are uninformed—it is an area of current research to estimate costs of different subtask orderings. The planning and execution tree is shown in figure 21; the execution visualization is shown in figure 20. This version uses only a small amount of abstraction. Even so, it solves 24 plans: two of size 7, two of size 5, and 20 of sizes 2–4; there are 31 primitives among the resulting plans. The corresponding flat plan would have required on the order of 30 steps, which would have been intractable in our planner

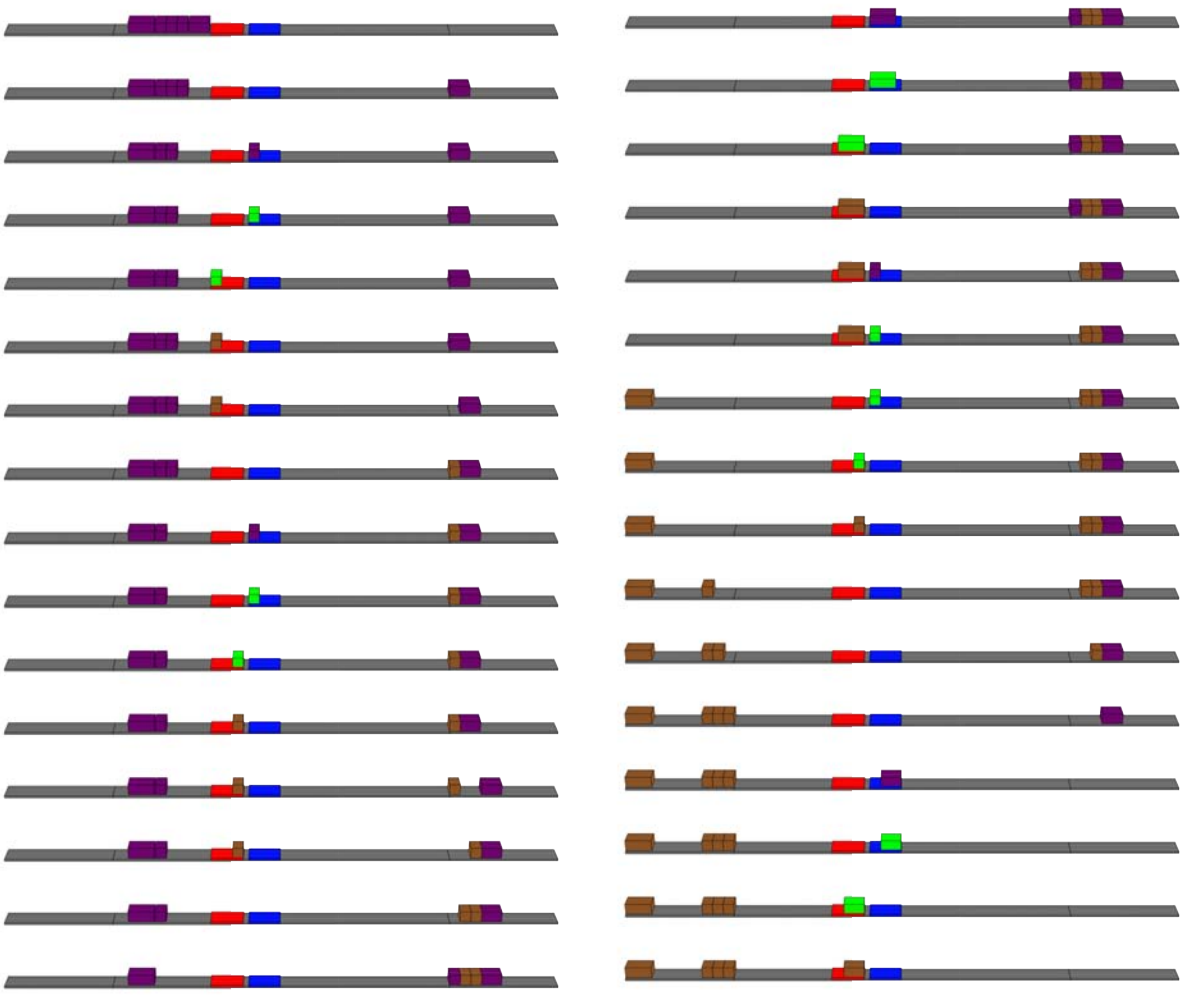


Figure 20: One-dimensional kitchen environment: goal is to cook all of the objects. Purple objects are dirty; green objects are clean but not cooked; brown objects are cooked; the red region is the *stove* and the blue region the *sink*. The initial state is at the top of the left column, time progress down the left column, then down the right column; the bottom right state satisfies the goal that all objects be cooked.

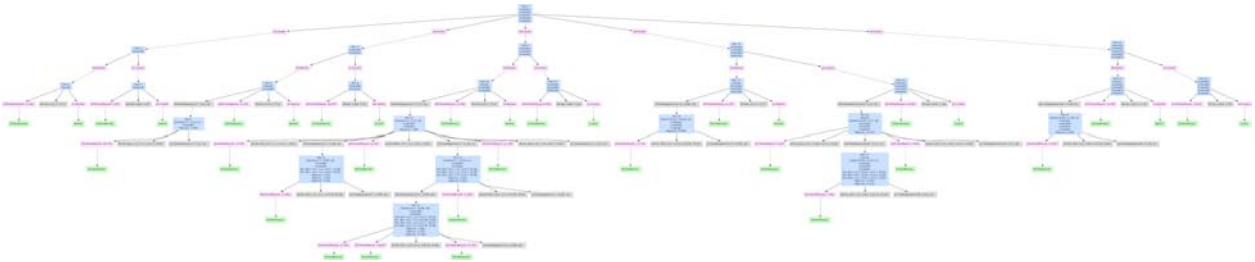


Figure 21: Plan for cooking all 5 objects, with a small amount of hierarchicalization.

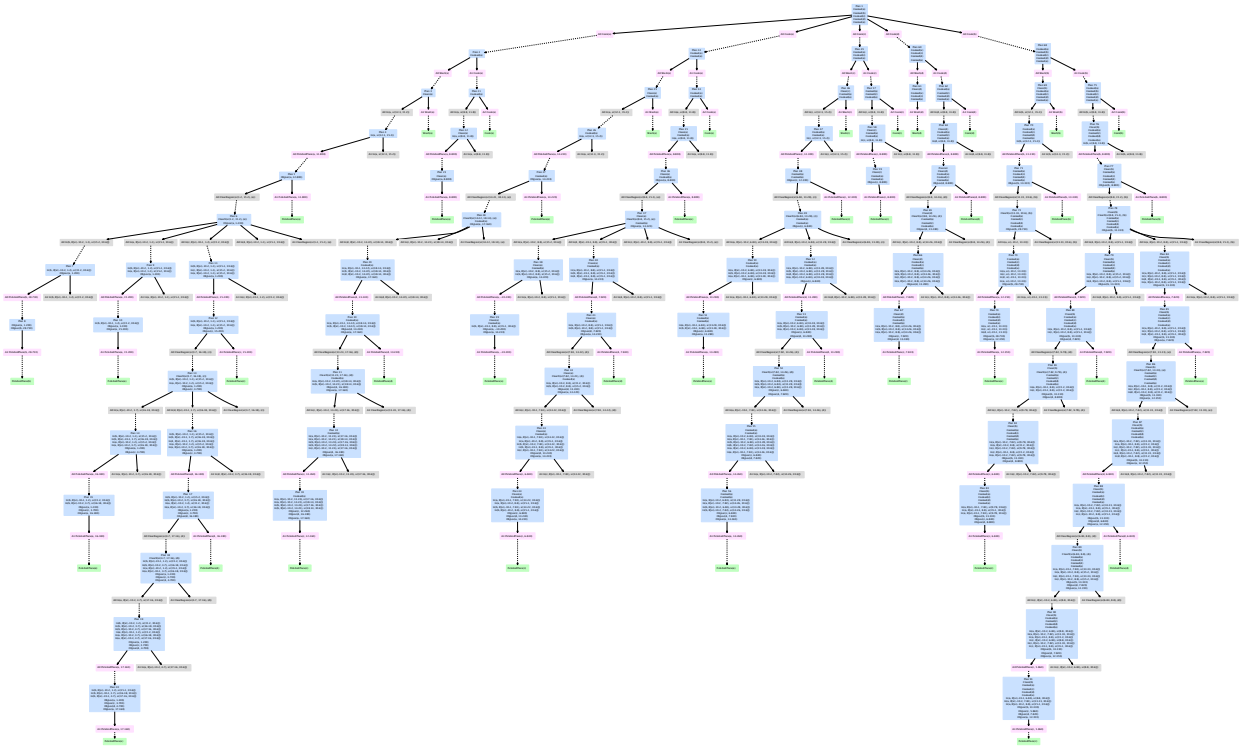


Figure 22: Plan for cooking all 5 objects with more hierarchicalization.

implementation. Figure 22 shows a plan with more hierarchicalization; the resulting plan uses 40 primitives. In this formulation, HPN solves 91 planning problems, the longest of which is length 5. This last formulation has taken the hierarchicalization too far: for a small amount of computational speed-up we now have execution sequences that are substantially longer than optimal.

E Mobile manipulation environment details

E.1 Entailment and contradiction

Entailment We specify conditions under which one fluent entails another. These conditions are not exhaustive, but they are sufficient for our implementation to work effectively, by avoiding generating inconsistent subgoals or goals that contain redundant fluents that entail one another.

- $PoseAt(o_a, p_a)$ entails $PoseAt(o_b, p_b)$ if the objects are the same and the poses are very close to one another:

$$o_a = o_b \;\&\; |p_a - p_b| \leq \delta \quad ,$$

  where $|p_a - p_b|$ is a vector of absolute differences in the components of the pose and δ is a vector of tolerances.

- $ConfAt(c_a)$ entails $ConfAt(c_b)$ if the base poses, hand poses, and gripper openings in c_a and c_b are all very close to one another:

$$|base(c_a) - base(c_b)| \leq \delta \;\&\; |hand(c_a) - hand(c_b)| \leq \delta \;\&\; |grip(c_a) - grip(c_b)| \leq \delta_{grip} \quad .$$

- $In(o_a, r_a)$ entails $In(o_b, r_b)$ if the objects are the same and region r_a is contained in region r_b:

$$o_a = o_b \;\&\; r_a \subseteq r_b \quad .$$

- $ClearX(r_a, x_a)$ entails $ClearX(r_b, x_b)$ if the first region contains the second and the first set of exceptions is contained in the second:

$$x_a \subseteq x_b \;\&\; r_b \subseteq r_a \quad .$$

- $PoseAt(o_a, p_a)$ entails $In(o_b, r_b)$ if the objects are equal and the volume taken by o_a at pose p_a is contained in r_b:

$$o_a = o_b \;\&\; v(o_a, p_a) \subseteq r_b \quad .$$

Contradiction These are the conditions under which one fluent contradicts another. Note that they are symmetric (if f contradicts g, then g contradicts g.) This is also not a complete set of contradictions, but it is sufficient for the implementation to be effective. The danger in having an incomplete set of contradictions is that we could generate an infeasible subgoal whose infeasibility is not detected until a more concrete level in the hierarchy.

- $PoseAt(o_a, p_a)$ contradicts $PoseAt(o_b, p_b)$ if the objects are the same but the poses are not nearly equal or if the objects are different but their volumes overlap:

$$(o_a = o_b \;\&\; |p_a - p_b| > \delta) \;||\; (o_a \neq o_b \;\&\; v(o_a, p_a) \cap v(o_b, p_b) \neq \emptyset) \quad .$$

- $PoseAt(o_a, p_a)$ contradicts $In(o_b, r_b)$ if the objects are the same and the object, if placed at pose p_a, is not contained in r_b; or, if the objects are not equal and o_b does not fit in the r_b when o_a is placed at p_a:

$$(o_a = o_b \;\&\; v(o_a, p_a) \not\subseteq r_b) \;||\; (o_a \neq o_b \;\&\; \neg f(o_b, r_b \setminus v(o_a, p_a))) \quad .$$

- $ConfAt(c_a)$ contradicts $ConfAt(c_b)$ if the configurations differ significantly in any dimension:

$$|base(c_a) - base(c_b)| > \delta \;||\; |hand(c_a) - hand(c_b)| > \delta \;||\; |grip(c_a) - grip(c_b)| > \delta \quad .$$

- $Holding(o_a, g_a)$ contradicts $Holding(o_b, g_b)$ if the objects are different or the grasps are significantly different.

$$o_a \neq o_b \;||\; |g_a - g_b| < \delta \quad .$$

- $Holding(o_a, g_a)$ contradicts $In(o_b, r_b)$ if the objects are equal, because the In fluent implicitly asserts that the object is not in the robot's hand:

$$o_a = o_b \quad .$$

- $In(o_a, r_a)$ contradicts $In(o_b, r_b)$ if the objects are the same but it does not fit in the intersection of the two regions[1]:

$$o_a = o_b \;\&\; \neg f(o_a, r_a \cap r_b) \quad .$$

[1]There should also be a test for the case in which the objects are not the same and they cannot both be in their respective regions at the same time; however that is a difficult test which is not currently implemented.

- $In(o_a, r_a)$ contradicts $ClearX(r_b, x_b)$ if the object is not in the set of exceptions and it does not fit in the part of r_a that is not also in r_b:

$$o_a \notin x_b \;\&\; \neg f(o_a, r_a \setminus r_b) \;\; .$$

- $ClearX(r_a, x_a)$ contradicts $PoseAt(o_b, p_b)$ if the object is not in the set of exceptions and o_b, when placed at pose p_b, overlaps r_a:

$$o_b \notin x_a \;\&\; v(o_b, p_b) \cap r_a \neq \emptyset \;\; .$$

F Apartment environment details

This appendix describes the environment from Section 4.2 in more detail.

F.1 State representation

To support this environment, we augment the world model to represent, for each food object, the amount it has been cooked, as well as the values of two binary attributes: *clean* (relevant for cups, plates, and pans) and *served* (relevant for food).

We extend the model to include support relationships so that, for example, when an object is put onto a pan, and the pan is moved, then the object moves along with it. For now, we define a *regionAbove* function that maps a container to a region above it, such that if an object is at the bottom of that region, it will move along with the container.

In addition, for the purpose of tidying the house, we store a "tidy" location for each object, where the object should be put away. These locations are not poses; just general regions, such as a particular shelf or closet. The function *pansFor* maps a food item into a set of pans that can be used to cook that food. The function *cookingTime* maps a food item into a number of steps it must be cooked until it is done.

F.2 Primitives

The added primitives in the apartment environment are non-geometric, and depend directly on attributes in the world model.

- CookPrimitive(*food*, *pan*, *burner*) causes the *cooked* attribute for the object *food* to be set to True. There must be no other pans on the same burner.
- CleanPrimitive(*obj*) causes *obj* to be clean, if *obj* is contained in the sink region.
- ServePrimitive(*food*, *plate*) causes *food* to be *served*, if it is on the plate and cooked.
- PreparePrimitive(*food*) causes *food* to be *prepared*, if it is on the kitchen table.

F.3 Fluents

Here are the new fluents and associated tests necessary to support the apartment environment.

- *Prepared*(*food*): true if the food has been prepared:

$$\tau_{Prepared}((food), s) := food \in s.prepared \ .$$

- *Clean*(*obj*): true if *obj* is clean:

$$\tau_{Clean}((obj), s) := obj \in s.clean \ .$$

- *Served*(*food*, *plate*): true if *food* is served on *plate*:

$$\tau_{Served}((food, plate), s) := v(food, s) \subseteq regionAbove(plate) \ \& \ food \in s.prepared \ .$$

- *Cooked*(*food*, *time*): true if *food* has been cooked for at least *time* steps:

$$\tau_{Cooked}((food, time), s) := s.cooked(food) \geq time \ .$$

- *Tidy*(*type*, *exceptions*): true if all objects of type *type*, except those in *exceptions* have been put into their tidy regions; in addition, non-food items must be clean:

$$\tau_{Tidy}((type, exceptions), s) :=$$
$$\forall o.\ o \in (s.universe(type) \setminus exceptions) \rightarrow v(o, s) \subseteq s.tidyRegion(o) \ \& \ s[o].clean \ .$$

There are no special contradictions or entailments among these fluents.

F.4 Operator descriptions

To formalize the cooking process, we use two operators. The StartCook operator begins the cooking. It requires the food to be prepared and in a clean pan, and the pan to be on a burner. It results in the food being cooked one unit and the pan being dirty. The Cook operator is nearly the same, except it does not require that the pan be clean, and it advances the amount of cooking by one unit from what it was before. There was no real inefficiency in planning with these operators so the conditions are all have abstraction value 0.

StartCook(*food*, s_{now}, γ):
effect: *Cooked*(*food*, 1)
pre:
choose: $pan \in pansFor(food, s_{now})$
choose: $burner \in burners(s_{now})$
In(*food*, *regionAbove*(*pan*)) **[0]**
Clean(*pan*) **[0]**
Prepared(*food*) **[0]**
In(*pan*, *regionAbove*(*burner*)) **[0]**
sideEffects:
Clean(*pan*) = False **[0]**
prim: CookPrimitive(*food*)

Cook($food, s_{now}, \gamma$):

effect: $Cooked(food, n)$

pre:

$n > 1$

choose: $pan \in pansFor(food, s_{now})$

choose: $burner \in burners(s_{now})$

$In(food, regionAbove(pan))$ **[0]**

$In(pan, regionAbove(burner))$ **[0]**

$Cooked(food, n-1)$ **[0]**

prim: CookPrimitive($food$)

For food to be served, it has to be cooked and put on a clean plate

Serve($food, plate, s_{now}, \gamma$):

effect: $Served(food, plate)$

pre:

$In(food, regionAbove(plate))$ **[1]**

$Cooked(food, cookingTime(food))$ **[1]**

$Clean(plate)$ **[1]**

sideEffects:

$Tidy(X, Y) = None$

$Clean(plate) =$ False **[0]**

prim: ServePrimitive($food$) **[0, 1]**

Preparing is just an extra step before cooking that needs to be performed with the food on the kitchen table.

Prepare($food, s_{now}, \gamma$):

effect: $Prepared(food)$

pre: $In(food, regionAbove(kitchenTable))$

prim: PreparePrimitive($food$)

Washing is used to make plates and pans clean.

Wash($item, s_{now}, \gamma$):

effect: $Clean(item)$

pre: $In(item, regionAbove(sink))$

prim: WashPrimitive($item$)

Tidying all the objects of some type, except for the exceptions, requires each to be put in its tidy region and, if it is not food, to be clean.

Tidy($type, exceptions, s_{now}, \gamma$):

effect: $Tidy(type)$

pre:

for: $o \in s_{now}.universe(type) \setminus exceptions$:

$In(o, s.tidyRegion(o))$ **[1]**

if $type(o) \neq food$: $Clean(o)$ **[1]**