Synthesis of Integrated Task and Motion Plans from Plan Outlines Using SMT Solvers

Rice University
Computer Science Department, MS 132
PO Box 1892
Houston, TX 77251-1892
Technical Report 15-01

Srinivas Nedunuri
nedunuri@rice.edu

Yue Wang
yw27@rice.edu

Sailesh Prabhu
snp3@rice.edu

Mark Moll
mmoll@rice.edu

Swarat Chaudhuri
swarat@rice.edu

Lydia E. Kavraki
kavraki@rice.edu

January 9, 2015
SYNTHESIS OF INTEGRATED TASK AND MOTION PLANS FROM PLAN OUTLINES USING SMT SOLVERS

Srinivas Nedunuri, Yue Wang, Sailesh Prabhhu, Mark Moll, Swarat Chaudhuri and Lydia E. Kavraki
Dept. of Computer Science, Rice University, Houston, TX 77251
nedunuri | yw27 | snp3 | mmoll | swarat | kavraki @rice.edu

Abstract. We present a new approach to integrated task and motion planning (ITMP) for robots performing mobile manipulation. In our approach, the user writes a high-level specification that captures partial knowledge about a mobile manipulation setting. In particular, this specification includes a plan outline that syntactically defines a space of plausible integrated plans, a set of logical requirements that the generated plan must satisfy, and a description of the physical space that the robot manipulates. A synthesis algorithm is now used to search for an integrated plan that falls within the space defined by the plan outline, and also satisfies all requirements.

Our synthesis algorithm complements continuous motion planning algorithms with calls to a Satisfiability Modulo Theories (SMT) solver. From the scene description, a motion planning algorithm is used to construct a placement graph, an abstraction of a manipulation graph, whose paths represent feasible, low-level motion plans. An SMT-solver is then used to symbolically explore the space of all integrated plans that use paths in the placement graph, and also satisfy the constraints demanded by the plan outline and the requirements.

Our approach is implemented in a system called Robosynth. We have evaluated Robosynth on a generalization of an ITMP problem investigated in prior work. The experiments demonstrate that our method is capable of generating integrated plans for a number of interesting variations on the problem.

1. Introduction

Integrated task and motion planning (ITMP) [1, 2, 3, 4] is a challenging class of planning problems that involve complex combinations of high-level task planning and low-level motion planning. In this paper, we present a new approach to ITMP, embodied in a system called Robosynth.

In the version of ITMP considered here, the task planning level is discrete and requires combinatorial exploration of the space of possible integrated plans, while the motion planning level is responsible for finding paths in continuous spaces. If m is the number of high-level actions (e.g., pick, place, move, etc.), the task planner has to explore a search space of \( O(m^k) \) plans to find a plan of length k. The continuous planning problem is PSPACE-complete in the degrees of freedom of the robot [5]. Moreover, the seamless integration of these two levels is difficult and eludes simple recipes [6, 7, 8, 3]. A strictly hierarchical approach where the task planner determines the task order and passes the solution to a continuous motion planner does not work. Fundamentally, this is because the task planner, given a choice of sequencing two tasks \( A \) and \( B \), may sequence \( A \) before \( B \), unaware that due to physical limitations, \( B \) must be carried out before \( A \). In the worst case, this may lead the task planner to send arbitrarily many plans to the motion planner before finding one is realizable at the physical level. Thus, for effective planning, some motion-level primitives must be exposed at the task level. At the same time, exposure of motion-level primitives at the task level is far from free as a large number of motion-level primitives must be introduced [9].

Our approach to solving the problem is based on two key ideas: (1) we request a limited amount of information from the programmer; and (2) we use a Satisfiability Modulo Theories (SMT) solver as a complement to continuous motion planning algorithms. The first idea utilizes the fact that while planning the actions of a robot, the human programmer usually has a partial picture of what an acceptable plan looks like. By letting the programmer specify this high-level knowledge in the form of a simple program, we can prevent the planning algorithm from searching through plans that are obviously unacceptable.

As for the latter idea, SMT-solvers [10, 11] are fully automatic, highly engineered satisfiability checkers for logical formulas in quantifier-free first-order theories. Over the last decade, these solvers have emerged as a core technology in many areas that demand analysis of large, discrete state spaces [12]. This is because large, even infinite, sets of system states can be compactly represented as formulas in first-order logic. Such
an approach allows us to solve problems of state space search using a small number of calls to an SMT-solver, leading to dramatic improvements in the scalability of state space analysis. The present paper uses an SMT-solver to efficiently explore the large combinatorial space of integrated plans. As far as we know, we are the first to explore the application of SMT-solvers in ITMP.

In more detail, the inputs to our approach (and the Robosynth system) include: (1) a scene description that specifies the robot and the physical workspace in which it operates; (2) a plan outline that describes high-level partial knowledge that the programmer has about plausible plans, and syntactically defines a space of integrated plans that we search; and (3) a set of logical, semantic requirements that the generated plan must satisfy.

To get a feel for what these inputs represent, consider the benchmark domain on which we have evaluated Robosynth, namely a simple Kitchen environment. The scene description specifies the layout of the kitchen, the locations of obstacles, the kinds of dishes that are available, the initial locations of dishes and the robot, and demarcates regions such as the food preparation areas, the Countertop, and the Dishwasher. The plan outline allows the programmer to supply information that is known to him/her while letting the tool determine details that the programmer does not care about or simply does not know, but can be inferred from the given information. For example, the programmer may know that larger items must be loaded before smaller ones in the dishwasher. What he or she probably would not know are the locations of dirty dishes or exactly where they should go in the dishwasher or the paths the robot should take; these are the plan unknowns. Finally, the logical requirements include goals like “All dirty dishes are to be cleaned and put away in storage” and constraints like “The path taken by the robot from the dishwasher to the storage area must avoid the food preparation area and all paths together must not total more than 100 units”. The plan outline and requirements are written as a C-like “program” where the objects that the robot moves around (for example, cups and dishes) are declared as symbolic variables, and actions that the robot performs (for example, moving and picking up an object) appear as function calls.

From the scene description, Robosynth first constructs a finite graph that we call a placement graph. The placement graph is an abstraction of a manipulation graph conventionally used in manipulation planning [13, 14]. A node in a manipulation graph represents a particular configuration of the robot and all the movable items in the workspace and an edge represents a physically allowable transition from one global configuration to another. For this reason manipulation graphs can grow very large, particularly when there are many objects. Our abstraction drastically reduces the number of nodes in the graph because the \( P_n^m \) permutations of \( m \) objects in \( n \) locations present in a manipulation graph are not represented, only the locations, leaving actual assignments to be designated later. The nodes representing robot positions are called b-points. Nodes representing object locations are called s-points. Edges in the graph connect b-points to other b-points and to s-points. Note that the placement graph is computed once per scene description and is common to all plan outlines in that scene description (Section 8 briefly discusses a more dynamic approach). Next, using program analysis techniques, Robosynth automatically computes a logical formula that represents the set of all integrated plans that follow the structure of the plan outline and satisfy the requirements, and are also consistent with paths in the placement graph. The problem of finding a plan that meets all the criteria now reduces to computing a satisfying solution of this formula. This is done using Z3 [10], a state-of-the-art SMT-solver.

We have evaluated Robosynth on a version of the ITMP problem that substantially generalizes prior work by Kaelbling and Lozano-Pérez [3]. This problem involves load/unloading a dishwasher and putting away the dishes in a kitchen using a PR2 robot. Our work extends earlier approaches such as [3] and Robosynth proves to be able to solve the given problem effectively under a rich variety of requirements and changes to the physical space.

This paper substantially extends our earlier work [15] in several ways. First, we have a new algorithm for constructing the placement graph; using this we can now represent multiple paths between two points in the workspace that pass through differently marked zones in the scene description. We have also automated its construction (the user need only identify the stable surfaces in the environment). Second, we have a discussion of the correctness of our approach, and its computational complexity. Third, we have expanded the Robosynth language and re-implemented the backend for efficiency. Finally, we describe how to represent some interesting variations on the problem originally described in our earlier work as well as an expanded set of experiments that benchmark the behavior of our tool.
The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces our benchmark example and uses it to illustrate the inputs and outputs of our approach. Sections 4 and 5 describe the internals of ROBOSYNTH. Section 6 demonstrates that our approach does indeed generate correct plans as well as finding a correct plan when one exists. Section 7 presents the results of several experiments we conducted with ROBOSYNTH and Section 8 concludes with a discussion of our approach and possible extensions.

2. Related Work

2.1. Program Synthesis. Our method is influenced by prior work in the programming language community on template-based program synthesis [16, 17, 18]. Just as our approach starts with a plan outline and a logical requirement, template-based program synthesis starts with a program template and a logical requirement. The goal in both cases is to complete the template/outline using solvers for automated reasoning. In all cases (including ours), a logical constraint of some sort is inferred from the program outline. The program semantics used differs in each case, but that is not an important distinction. Perhaps more significantly, template-based synthesis has so far been motivated by pure software applications. This means that satisfying the specified requirement is all that is needed. In contrast, because our application domain is robotics, our plans must also be realizable in the physical world. This additional requirement significantly increases the complexity of the synthesis problem. To the best of our knowledge, ours is the first application of such a synthesis approach to robotics. In that sense, our work could be considered a targeting of template-based synthesis towards robot programming. There has also been work on synthesizing fast domain-specific planners from formal specifications [19, 20] by utilizing declarative domain knowledge. Although the approach can lead to very fast planners, it requires more domain knowledge and is typically less automated.

2.2. Task Planning. Automated planners have a long history in AI [21]. Prominent among these are heuristic planners such as FF [22]. Such planners have traditionally been employed on problems in which there are a number of ways of achieving a given goal and therefore require combinatorial search [23]. A number of approaches to the ITMP make use of such automated planners in combination with motion planning [1, 9, 6, 8] and this is discussed further in the next subsection. AI planners that share our use of Boolean satisfiability solvers [24, 25] are known as SAT planners. A SAT planner constructs a Boolean formula that represents the existence of a plan of a given length. As in our case, satisfiability of the formula means that a plan exists, and the steps of the plan can be extracted from the model returned by the solver. Here we just briefly point out a significant difference between our work and that of automated planning in general is that our inputs include a plan outline containing programmer supplied partial information. This significantly decreases the space of possible plans that must be searched. Levesque also has a programming language in which to express “robot programs” [26]. While Levesque does handle unbounded loops he does not provide any correctness guarantees for the generated program. He also does not include motion planning, and there is no notion of incompletely specified actions with unknowns that are filled in by the solver as we have.

2.3. Integrated Task and Motion Planning. In recent years, several approaches have been proposed to integrate the motion planning level with task planning. Havir et al. [2] use a form of SAT planning to incorporate conflict-directed learning when an action fails at the motion planning level. Actual motion planning information is however not exposed. In contrast, we expose motion planning information to the SMT solver so information about why a particular solution fails is available and the same motion planning query need not be re-attempted. Both task level and motion level concepts can be uniformly described by extending the input language PDDL [27] used by heuristic planners such as FF [9]. Unfortunately, lifting motion level operators to the task planning level massively increases the search space (a motion level operator in Dornhege et al. [9] takes twenty one parameters). Garrett et al. [8] modify the construction of the relaxed planning graph used by FF to take into account motion level preconditions on actions. No change is required to the computation of the heuristic used by FF (which is still the length of a plan in the relaxed planning graph). To avoid the explosion in the search space entailed by directly planning over the configuration space, they use a pre-computed structure (Conditional Reachability Graph) which is similar in intent to our placement graphs.

ASYMOV [1] partially ameliorates the graph explosion problem by introducing a separate graph for each object and robot, which is used for fast validation of motion plans. Instead of a heuristic planner, Wolfe et al.
[4] use a Hierarchical Task Network (HTN) planner [21]. In addition to a planning problem, HTN planning takes as input a schema for how to recursively decompose complex tasks, terminating in motion primitives at the lowest level. By using dynamic programming they are able to construct hierarchically optimal plans. Schemas can be viewed as grammars defining a language of legal plans. Thus HTN planners share with our approach the input of domain knowledge. They are also some of the fastest planners known. However, they require a level of domain expertise in order to correctly, completely, and efficiently codify the space of all possible plans. In contrast, our approach accepts partial user knowledge sufficient to solve the given problem, expressed as a plan outline. Feedback from the tool tells the programmer whether or not the plan outline can solve the given problem.

A different form of hierarchy is used in Hierarchical Planning in the Now (HPN) [3] which comes the closest to ours in scope and intent. In HPN, actions are represented at varying levels of abstraction, obtained by postponing different preconditions of an action. While HPN is very powerful (and has been extended to belief space planning [28]), it rests on the assumption that an action can always be reversed. This is not always true (e.g., if the robot exceeds the time limit for achieving some goal). Even if they can be reversed, some actions may prove rather expensive to reverse (e.g., running out of fuel or unloading a number of items from a truck only to discover there is insufficient space for the last item). Although our work bears resemblances to HPN, the two frameworks are different in design and have their respective advantages and disadvantages. Distinguishing characteristics of our work which also underline the differences with HPN [3] are the following:

- The Robosynth language allows a programmer to place constraints on the paths that are returned by the solver.
- When planning in the now, it is quite possible that the robot may have to (repeatedly) move just placed items to get them out of the way. As we will show, our plan outline language allows the programmer to write a simple loop which forces Robosynth to find an efficient solution if one exists, eliminating such unproductive actions.
- It is not necessary for the user to designate a specific “safe region” for where to place removed obstructions.
- HPN requires re-planning when a plan fails. Replanning may not always be possible, or may be prohibitively expensive in some situations.

Srivastava et al.[6, 7] take a different approach to the problem of interfacing a discrete task planner with the underlying continuous world by Skolemizing the continuous variables. While avoiding unnecessary discretization, that approach shares some of the drawbacks of HPN, particularly the possibility of the robot having to undo a lot of work in the real world if a particular sub-plan fails. The way in which the Skolem terms are interpreted at the motion level is also hard-coded for each type of action and does not have the same generality as pre and post-condition definitions of actions. In contrast, in Robosynth users can define their own domain-specific actions in the same way as the built-in actions.

Lagrioul et al. [29] take a different approach to the ITMP by verifying a task plan as it is being generated. Their goal is to eliminate a task plan that is infeasible at the motion level as early as possible. They do this by introducing a number of constraints on the motion level expressed over the motion level unknowns. After every action generated by the task planner, the bounds on each unknown are refined (reduced) by using a linear programming solver. A choice of a variable is then propagated reducing the variable ranges further. A related approach is that of Lozano-Perez and Kaelbling [30] who use a task planner to find a sequence of actions that achieves a goal at the task level, with the arguments left unspecified. A CSP solver is to find values for the discrete arguments. We, on the other hand, generate constraints that encompass both the discrete planning and the continuous motion planning.

Finally as an alternative to probabilistic approaches at the continuous level, Hung et al. [31] make use of an SMT solver to directly solve the path planning problem. However, the specific theory that they use for efficiency reasons (Difference Logic over Reals) limits them to rectangular obstacles and rectilinear paths.

While numerous extensions (such as reactivity and constraint propagation) could be considered, they are outside the scope of this paper, which is primarily focused on our use of automated constraint solvers for solving the ITMP.

2.4 **Controller Synthesis.** Another major research effort has been the automatic synthesis of both reactive [32, 33, 34] and non-reactive controllers [35] and plans [36] for robots from temporal logic specifications. Tools
such as LTL mop [32] go beyond what we do, in that they handle reactive robotics. The primary difference between these approaches and ours is that in our approach a space consisting of a vast number of plausible integrated plans can be represented concisely in the form of a (quantifier-free) first-order logical constraint, and a solver is invoked to perform operations on such constraints using a small number of calls. Such approaches to analysis of large combinatorial spaces can often be dramatically more scalable than methods relying on a propositional approaches, such as LTL, whether the space is represented by BDDs or propositional logic formulas [37]. We also rely on a programmer supplied plan outline to both reduce both the space of possible plans and the size of input specifications. An extension of our approach to reactive robotics is left for future work.

3. System overview and motivating example

In this section, we describe the inputs and outputs of the Robosynth approach using an illustrative example. The internals of the tool (see dashed box in Fig. 3.1) are described in Sections 4 and 5.

As outlined in the Introduction, the inputs required by Robosynth are (1) a scene description; (2) a plan outline; and (3) a set of logical requirements. We will now illustrate each of these pieces by means of our running example.

Example: Planning for a robot in kitchen. Consider a household robot operating in a kitchen environment. Kaelbling and Lozano-Pérez [3] take as an example the problem of planning for a robot to move a given item from its current location, place it in the Dishwasher, run the Dishwasher, remove the item and place it in Storage. We show how to describe in Robosynth a more complex class of problems in which:

- There is a set of dirty dishes, which can be located anywhere in the kitchen, to be moved to the dishwasher. By “anywhere” we mean that although in any particular problem input the dishes are in specific locations, the plan outline is agnostic as to what those specific locations are.
- In the Dishwasher, larger items such as plates can block access to smaller items such as cups. This information is automatically computed by the low-level planner (see 4).
- There are constraints on the paths the robot takes. For instance, they may be limited in length or be required to avoid or pass through certain regions.
- The number of dirty dishes exceeds the Dishwasher capacity. This means the dishes have to be loaded in batches. However, we prefer for the programmer to not have to determine the bounds on the various nested loops, something that would be required if programming in a conventional programming language.

Input 1. Scene Description. The scene description is provided in two files, called the domain and the scene. The domain represents information which does not vary across different instances of a given problem, such as immovable obstacles, layout, stable locations for the robot and objects. The domain used in the present example is shown in Fig. 3.2. The green square in the middle of the figure is an “island” that the robot must not collide with. Right of that is the food preparation area (FoodPrep). The scene provides information that can vary across different instances, in our case it provides the number of different cups, plates, etc. and their initial locations. It also allows the programmer to define collections of locations where objects can be placed, called regions. At the bottom of the figure is the Storage region; At the top of the
Figure 3.2. Example kitchen domain. Three alternative paths between Dishwasher and Storage are shown in the Overhead View.

figure are the Countertop and Dishwasher regions in that order. In this way, it is possible to define certain areas of Storage or the Countertop, for example, as inaccessible for a particular problem instance.

Input 2. Plan outline. The plan outline is a program written by the programmer that captures high-level, partial knowledge about what successful integrated plans look like. In our kitchen example, the plan outline captures the fact that in any reasonable plan, the robot would have to move from its current location and pick up a dish before it attempts to put it anywhere. While this ordering looks “obvious,” it nonetheless allows the SMT solver to prune out a large number of meaningless orderings of robot actions. In general, we view the plan outline as a syntactic, imperative definition of a large space of integrated plans. The goal of Robosynth is to search this space and come up with a plan that satisfies the requirements. Robosynth supplies the programmer with a C-like language in which to write plan outlines and requirements. Fig. 3.3 shows the essential parts of a plan outline for the example problem. Lines 1 and 2 import the domain and scene information respectively. Variables are declared where needed; these are the unknowns to which Robosynth will assign values. All variables used in this sketch are of type Path which is the type of robot paths, or Object, which is the type of items being manipulated. Lines 4-20 constitute the body of the main procedure. This code specifies the high-level knowledge that in any reasonable plan, the robot must repeatedly perform the following actions in sequence:

1. Repeatedly:
1. #import KitchenDomain
2. #import KitchenScene
3. void main()
4. { repeat
5. { repeat
6. { Path path1, path2;
7. Object dish;
8. fetch(?dish,?path1);
9. put(?dish,Dishwasher,?path2);
10. }
11. run(Dishwasher);
12. repeat
13. { Path path1, path2;
14. Object dish;
15. fetch(?dish,?path1);
16. put(?dish,Storage,?path2);
17. }
18. }
19. @goal: clean(DIRTY) & contains(Storage,DIRTY)
20. }
21. @constraint: ||?path|| <= 10 & ~crosses(?path,FoodPrep)
22. @pre-handler: fetch(obj,_,)
23. while (obstructs(?obst,obj))
24. { Path pathObst;
25. fetch(?obst,?pathObst);
26. put(?obst,?tempR,?pathObst);
27. }
28. }
29. }
30. }
31. @goal: clean(DIRTY) & contains(Storage,DIRTY)
32. @constraint: ||?path|| <= 10 & ~crosses(?path,FoodPrep)
33. @pre-handler: fetch(obj,_,)
34. while (obstructs(?obst,obj))
35. { Path pathObst;
36. fetch(?obst,?pathObst);
37. put(?obst,?tempR,?pathObst);
38. }
39. }
40. }
41. }
42. }
43.
44. Figure 3.3. Plan outline for the Dirty Dishes problem with multiple loads
45.
46. (a) Follow an (as yet unknown) path ?path1 from the robot’s current location to fetch an (unknown) dirty dish dish. The plan outline does not need to specify where the dish is located. The action fetch is part of the Robosynth language.
47. (b) Follow the unknown path ?path2 to the dishwasher and put the item in there. The action put is part of the Robosynth language.1
48. (2) When all remaining dirty items are loaded or the dishwasher is full, run the dishwasher.
49. (3) Unload the dishwasher and place the items in storage. The code for this is similar to the first loop.
50. As another major point of difference with conventional programming languages, note the use of the non-deterministic repeat construct provided by the Robosynth language. This construct (akin to Kleene ∗ in regular expressions) allows for an action to be repeated an arbitrary number of times, with the tool determining how many times (by default the least number of iterations). Without such a construct, the programmer would need to be aware of the dishwasher capacity, use for loops that keep track of the number of dishes loaded, and calculate how many loads were required.

Input 3. Requirements. Requirements come in the form of a goal for the planner, and constraints that must hold along the planned paths. Note in the example above that Robosynth is required to find bindings for the loop variable dish such that eventually the entire collection of DIRTY dishes is washed and placed in Storage. In order to do this, Robosynth needs to be told what the goal of the plan outline is. This is specified in Line 21 using the @goal construct. Examples of constraints are: health and safety regulations may require that paths should not cross, or that paths between certain regions should not come within a certain distance of each other; power limitations may limit paths to a certain length; protocols for clearing a building may require that paths must visit certain regions in a particular order, and so on. The constraints in our example are that: (1) the total length of any path is constrained to be under 10 units; and at no point

1The actions fetch and put were called pickup and place in our earlier work [15]. pickup and place no longer move the base, and instead fetch and put translate to a move action followed by a pickup and place resp.
must the robot go through the food preparation area. The specification of these properties uses functions like \( \|\| \) for path length, and predicates like \( \text{crosses}(\text{crosses}(p,z)) \) is true when \( p \) is a path crossing a zone \( z \) and the value of these properties is obtained from the placement graph (see Section 5 for details). It is the presence of linear arithmetic and functions in constraints that requires us to use an SMT solver rather than just a SAT solver.

**Events.** Robosynth also offers a special syntax for specifying exceptional scenarios that prevent the straightforward execution of a robot action. For example, there may be items obstructing the target items, and these must first be moved into some safe place that is out of the way. In many GUI oriented languages such as Java, it is common to define event handlers for GUI events like a mouse button being clicked, a window being moved, etc. The event handler is a piece of code that is executed upon an occurrence of the corresponding event. For example, the event “left mouse button clicked” might be handled by an event handler which determines how the interface responds to the left mouse button being clicked. Likewise, in Robosynth, conceptually a before event and after event is “fired” before and after each action. The programmer can set up event handlers to carry out whatever corrective action is needed. An example event handler for the fetch action is shown in lines 25 to 30. The code shown moves any obstacles out of the way to an unknown safe region (\(?\text{tempR}\)) that Robosynth determines automatically.

3.1. **Plan Synthesis.** The inputs described above, namely the scene description, plan outline, and requirements, are fed to Robosynth as shown in Fig. 3.1. Robosynth computes the placement graph and then uses that to determine values for the unknown variables in the plan outline (for example, \(?\text{path}\)) such that the requirements are satisfied. If there is no plan that follows the plan outline and meets the requirements, Robosynth will indicate that no solution exists. The placement graph is the connection between the discrete solver and the underlying motion planning, and its construction is discussed in Section 4. Note that once the unknowns are resolved in our system, the plan outline can be instantiated and the corresponding sequence of paths can be extracted from the placement graph. The result is a deterministic sequence of instructions for the robot, i.e., a complete plan. For example, Fig. 3.2 shows some sample paths that are found for the present problem when the constraints are varied. More details on the plans computed by Robosynth on our current example are presented in Section 7.

Finally, we wish to draw the reader’s attention to the following in the plan outline:

1. Obstructing items may be blocking not just the target items but also each other. Since the programmer does not know which obstructions are not themselves blocked, a feasible order in which to remove them is left for Robosynth to determine, indicated by the non-deterministic choice of obstruction (\(?\text{obst}\)) in the while loop of the event handler (Lines 26-30).

2. Recall there are physical constraints on the order in which items can be placed in the dishwasher, with plates blocking cups. By virtue of the fact that the plan outline only permits each item to be picked up and placed once (lines 9,10 and 16,17), Robosynth is forced to find an efficient way of moving the dishes (cups then plates).

3. The plan generated by Robosynth contains no references to external events, that is, it does not require run-time sensing. This is possible because we syntactically process and remove the event handlers (Appendix 1).

3.2. **Variations.** To illustrate the richness of the language of Robosynth, we will give some examples of interesting variations on the kitchen problem which can be expressed in and solved by Robosynth.

**Just-in-time exception handling.** There may be situations in which it may be more efficient to deal with a problem at the destination even after a robot has picked up an item, for example if the destination is some distance away from the current item location. Suppose there is a door on the Storage unit that needs to be opened. For this, a handler for the event that is fired before the put action is executed (line 17) can be used:

```plaintext
@pre-handler: put(obj,Storage,): if (closed?(Storage.door)) { place(obj,?tempR); open(Storage.door); pickup(obj,?tempR); place(obj,Storage); }
```
Note the use of the pickup and place actions to require Robosynth to find a place to put the item (in tempR) without moving the base. If no such space is available, the synthesis will fail.

**Layout Experimentation.** Our approach also supports experimenting with the assignment to regions. Suppose in the plan outline of Fig. 3.3 the exact amount of Storage space necessary had not been determined. Then Storage on line 14 could be replaced with a variable, say storage

14. put(o, ?storage, ?path2)

Now Robosynth would return an assignment for the Storage region. Indeed by calling the solver multiple times, different regions can be obtained.

Having covered the black box behavior of Robosynth, the next two sections cover the internals.

### 4. The Placement Graph Generator

This sections explains how placement graphs differ from standard manipulation graphs, justifies the need for them, and describes how they are generated.

#### 4.1. Background: Manipulation Graphs.

The usual way of representing manipulation planning problems is with a manipulation graph [13, 14, 38, 1]. A node in a manipulation graph represents a particular configuration of the robot and all the movable items in the workspace and an edge represents a physically allowable transition from one global configuration to another.

The main drawback of manipulation graphs is their size - every stable configuration of all the objects and every robot pose in which the robot is grasping an object becomes a node in the graph. Additionally, a problem unique to our approach is the difficulty of making path information available to the programmer. Consider the unknown path1 in the plan outline example of Fig 3.3. The actual path found by the solver must be valid not only for several different kinds of dishes such as cups and plates, but also for the many possible locations of the same dish on the Countertop. Clearly no single such value can exist given the different grasps needed for different objects and the different places from which locations are accessible. A natural solution to both these problems is to decouple the motion of the base from the motion of the arms, and this is something we exploit in our work. An additional benefit of this solution is that the base movements can be planned in a low dimensional space. The result is a variant of the manipulation graph we call a placement graph, described next.

#### 4.2. Placement Graphs.

The vertices of the placement graph are divided into so-called base points (b-points) and stable points (s-points). The b-points correspond to robot configurations that allow the robot to reach many possible object locations without moving the base. The s-points correspond to configurations of the robot potentially holding an object resting on a support surface in a stable grasp; thus each s-point is only specific to a kind of object (Cup, Plate, etc.), not to an individual object. The b-point from which an s-point is accessible to the robot is called the parent b-point of that s-point. The construction of b-points and s-points given a collection of stable surfaces has been automated (see Section 4.4). In future the graph itself could be computed adaptively and lazily [39, 40].

An edge between a b-point and another b-point or s-point corresponds to a collision-free (physical) path between the corresponding robot configurations. We define that an s-point i blocks an s-point j iff an object in s-point i would collide with the robot or object along any (physical) path from s-point j to its parent b-point. This is indicated by a directed edge from the blocked s-point to the blocking s-point. A point-to-point sampling-based planner (KPIECE [41]) is used to find a feasible path between a b-point and an s-point while the subgraph of paths between b-points is computed by a PRM planner [14]. The blocking edges are determined by performing collision checks on the edges connecting b-points to child s-points.

A portion of an example placement graph for the kitchen domain is shown in Fig. 4.1. The b-point and s-point names begin with “B_” and “S_”, respectively. Solid lines depict bidirectional edges between b-points, and between b-points and s-points, and dotted arrows depict blocking edges. The graph is constructed by reading in the scene description which is why it appears as one of the inputs to the Constraint Generator in Fig. 3.1. The annotations on the edges are described next.
4.3. Representing properties on a placement graph. The requirements input to Robosynth often places constraints on the paths through the workspace (such as \(||?path|| \leq 10\) & \(\neg\)crosses(?path,FoodPrep)). Other possibilities, not yet implemented, are clearance of the path, level of difficulty, energy consumption, etc. Such properties can be represented straightforwardly by annotating edges in the placement graph. Edges are also annotated with labels referring to zones and kinds. Zones are areas of interest in a workspace that the robot may have to pass through or avoid, and are indicated in the scene description (e.g., the FoodPrep zone is shown in Fig. 3.2). Note that zones need not necessarily be workspace related. For example, they might partition the nodes according to which specific dimensions of the overall configuration space they belong in, that is nodes that represent the use of a particular joint or robot arm. Kinds denote the kind of object which is placed at a location. Examples of kinds in a kitchen environment are Cup, Plate, etc. For an object to be placed at a location, its kind must match that of the location it is being placed at and the corresponding grasp must be used by the robot. Kinds are edge annotations and not location annotations because they denote a relationship between a particular base node and a location node. The preceding discussion motivates the following definition of placement graphs.

**Definition 4.1.** Placement Graph

A placement graph is a tuple \((V, E, Z, K)\) where \(V = V_B \cup V_S\) is a set of nodes. Node in \(V_B\) are called b-points, nodes in \(V_S\) s-points. \(Z\) and \(K\) are sets of labels denoting zones and kinds resp. \((V_B \cap V_S \cap Z \cap K = \emptyset)\). \(E = E_B \cup E_L \cup E_S\) is a set of directed edges where \(E_B \subseteq V_B \times V_B \times [Z] \times N\), \(E_L \subseteq V_B \times V_S \times K\) and \(E_S \subseteq V_S \times V_S\). (Square brackets denote a sequence). Edges in \(E_B\) are base edges, each edge has an associated sequence of labels and a length. Edges in \(E_L\) are local edges in which each edge is labeled with a kind. Edges in \(E_S\) are blocking edges, that is an edge \((s, t)\) in \(E_S\) means that \(s\) is blocked by \(t\).

4.4. Constructing a placement graph.

4.4.1. Constructing a PRM. We have developed an algorithm (Alg. 1) for automatically populating the initial PRM. The algorithm takes a user supplied surface in the workspace where objects can be placed. The surface is gridded into a set of stable locations \(L\). Each iteration of the algorithm adds a b-point and connects it to the s-points corresponding to those locations which the robot can reach from the base configuration corresponding to the added b-point. Once an iteration fails to connect any further b-points, the entire algorithm terminates. Each iteration executes four steps:

1. **Sampling for a Base Configuration** The algorithm samples for base configurations with a combination of uniform and obstacle-based sampling [42]. The obstacle-based sampler allows the b-point’s children to grasp more object configurations.

2. **Generating Child s-points** For each b-point sampled, the algorithm generates several child s-points. The algorithm generates the s-point by positioning the robot’s base at the b-point and, then, performing inverse kinematics to discover an arm configuration to grasp the stable configurations in \(L\). A table \((BPt2Locs)\) records the stable object configurations that the children of a b-point grasp.

3. **Selecting and Improving a b-point** After step 2 generates children s-points, the algorithm iterates through the b-points in \(BPt2Locs\) and picks the b-point whose children grasp the most object...
Algorithm 1 Automating the b-point and local edge construction in the placement graph

**Input:** A surface S

L := grid(S);
BPt2Locs := emptyTable;

repeat
(1) \( B := \) sample for base configurations in the robot base configuration space
(2) for every configuration \( b \) in \( B \):
   (a) \( R := \) those locations in \( L \) that the robot can reach from \( b \)
   (b) Enter \((b, R)\) in a table BPt2Locs
(3) Select an entry \((\tilde{b}, \tilde{R})\) in BPt2Locs such that for every entry \((b, R)\) in BPt2Locs, \(|\tilde{R}| \geq |R'|\). Adjust \( \tilde{b} \) using Gaussian sampling.
(4) if \( \tilde{R} \neq \emptyset \) then remove \( \tilde{R} \) from \( L \) and update the entry \( R \) for every key \( b \neq \tilde{b} \) in BPt2Locs to \( R - \tilde{R} \)
until \( R = \emptyset \)

**Output:** BPt2Locs /*provides the local edge from each b-point to its set of accessible s-points*/

configurations in \( L \). To increase the set of objects reached by the b-point, the algorithm samples around the selected b-point with a Gaussian sampler [43] and picks the base configuration that reaches the most object configurations.

(4) **Updating L and BPt2Locs** Once the algorithm selects a b-point to add to the placement graph, it removes the object configurations reached by its children from \( L \), and it updates BPt2Locs so that it only maps a b-point to object configurations in \( L \).

Once the algorithm has run, KPIECE [41] is used to connect b-points with their child s-points, and a PRM planner is invoked to connect the b-points.

4.4.2. **Transforming the PRM to a Placement Graph.** The PRM constructed by the previous step will typically contain multiple paths between b-points, not all of which are interesting from the programmer’s point of view (e.g., they may be physically close to each other and pass through the same zones). We have developed an algorithm for constructing placement graphs which, given a PRM in which states are labeled with such properties, constructs a placement graph \( G \) in which each path in the PRM between a pair of nodes is represented by an edge, and two edges exist between a pair of nodes only if those edges are labeled with distinct sets of properties. The algorithm is described in Appendix 3.

There are a couple of drawbacks of our approach of abstracting away some of the paths in the underlying PRM. One is that there is no guarantee that the shortest among a set of indistinguishable paths will be returned. Therefore a path length constraint may fail to be satisfied even though such a path does exist. The other is that distances are over-estimated. However, this is not usually expected to be a problem unless the distance represents some metric that is being maximized.

Note that b-points and s-points are essentially *internal* to ROBOSYNTH. They are not exposed to the programmer who can work with more user-friendly concepts such as regions and zones. Also, for complex maneuvers, the robot may need to place an object at a stable position in order to change grasps or base position. This requires any path between the starting b-point and the ending one go via another b-point. For this reason, the placement graph is not necessarily a complete graph.

The placement graph is fed to the Formula Generator, described next.

5. **The Formula Generator**

This section describes internals of the Formula Generator (box “Formula generator” in Fig. 3.1), which is the heart of ROBOSYNTH.

5.1. **Overview.** The function of the constraint generator in a nutshell is as follows: construct a logical formula that represents the meaning of the plan outline and any required initial conditions required to achieve the stated goal. This formula contains variables corresponding to the plan unknowns (namely the variables representing paths the robot follows such as \( \text{path}_1 \), \( \text{path}_2 \), locations of objects, variables such as \( \text{dish} \) in the loop). It also contains predicates that are constructed from the placement graph, as we will
shortly explain. Such a logical formula can be automatically constructed by utilizing a style of program semantics called *weakest precondition semantics* [44, 45] (w.p. semantics), which is described in the next subsection.

Finding a satisfactory assignment to the unknowns in this formula amounts to checking the *satisfiability* of the formula. Formally, a formula is satisfiable if there is a set of assignments to each of the unknowns, called a *model*, which makes the formula true. For example the formula \( p \land x > 2 \) is satisfiable by a model \( M = [p \leftrightarrow true, x \leftrightarrow 3] \), written as \( M \models p \land x > 2 \). On the other hand, the formula \( x > 2 \land \neg(x > 1) \) is not satisfiable (under the standard interpretation of \( \neg \)). Satisfiability of quantifier-free formulas can be determined automatically by an SMT (Satisfiability Modulo Theories) solver which returns a model in case the formula is satisfiable. The SMT solver we use is Z3 [10]. If Z3 finds the formula satisfiable it returns values for the unknowns. These values can be used to instantiate the plan outline to obtain a plan.

5.2. **Weakest Preconditions.** Weakest preconditions were introduced by Dijkstra [45] to serve as a practical way of applying Hoare logic [46] to program derivation. The idea behind weakest preconditions is to push or regress a given goal back through a program to determine what needs to hold at the beginning of the program in order for the goal to be achieved by the program\(^2\).

To understand what this means in our setting, suppose we are given a plan outline body \( P \) and a requirement \( G \) that must hold at the end of the plan. Define a state \( s \) to be an assignment of appropriately-typed values to the known and unknown variables of \( P \). Now note that a plan is nothing but an instantiation of the variables of \( P \) with values from an assignment \( s \), written \( P[s] \). Each action of \( P[s] \) effectively updates the state it starts in (because robot actions change the locations of objects), resulting in a final state when all the steps have executed. The integrated plan is correct for a goal \( G \) if executing it leads to a final state in which \( G \) holds. Then the *weakest precondition* of \( P \) with respect to \( G \), written \( wp(P,G) \), corresponds to the largest set of states \( \{s\} \) for which the plan \( P[s] \) is correct. In Robosynth, this weakest precondition is represented as a (quantifier-free) first-order formula.

To see how \( wp(P,G) \) is determined, consider first the simplest example where \( P \) is a single action, e.g.

\[
\text{pickup(Cup1,Cooking,?path1)};
\]

and \( G \) is \( \text{holding(Cup1)} \) \& \( \text{loc(Cup2)=s2} \) ("Robot is holding Cup1 and Cup2 is at s2"). Now to determine \( wp(P,G) \), two things are needed:

- A formal definition of the action (here the pickup action) (Sec. 5.3), and
- A way of pushing \( G \) back through \( P \) (Sec. 5.4)

5.3. **Formal definition of actions.** Note that an action carried out by a robot achieves a certain effect (e.g. block on a table, dish in hand, etc.). However, many actions also have specific conditions that must be satisfied before they can be carried out. Thus actions can be viewed as defining a *contract*. If the required conditions hold before executing the action, then the effects of the action are guaranteed to hold after executing the action. For this reason, it is convenient to specify actions in the form of a triple (*action name, precondition, postcondition*). The pickup action, for example, is specified in the following formal definition:

<table>
<thead>
<tr>
<th>action</th>
<th>pickup(o:Obj,b:Bpt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition</td>
<td>( \text{loc}(o) \in \text{reach}(b,\text{kind}(o)) \land \exists o' . \text{obstr}(\text{loc}(o'),\text{loc}(o)) \land \exists o' . \text{holding}(o') )</td>
</tr>
<tr>
<td>postcondition</td>
<td>( \text{holding}(o) \land \text{CURR}=b )</td>
</tr>
</tbody>
</table>

The action definition makes use of several pre-defined *fluents* (a fluent is a predicate or function whose value can change over time). The fluents we use are \( \text{loc}(o) \), \( \text{holding}(o) \) and \( \text{CURR} \), which return resp. the current location of an object, whether the robot is currently holding the object, and the b-point the robot is currently at. Fluents are evaluated w.r.t. the current state. Robosynth internally keeps track of where each object is at any stage of the plan and when objects should be picked up or released thus providing the interpretation of the fluents. We use the state-variable representation [21] of a state, namely as a

\(^2\) A similar idea called *goal regression* was developed around the same time in AI Planning by Waldinger [47]. We use the weakest precondition approach because it is was developed specifically for programs.
<table>
<thead>
<tr>
<th>Function &amp; Params</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in(l,r)</code></td>
<td>is S-pt <code>l</code> contained in Region <code>r</code>?</td>
</tr>
<tr>
<td><code>crosses(u,z)</code></td>
<td>does Path <code>u</code> cross Zone <code>z</code>?</td>
</tr>
<tr>
<td><code>path(u,b1,b2)</code></td>
<td>does the Path <code>u</code> exist between B-pts <code>b1</code> and <code>b2</code>?</td>
</tr>
<tr>
<td><code>obstr(l1,l2)</code></td>
<td>would an object placed at the first S-pt block access to the second?</td>
</tr>
<tr>
<td><code>reach(b,k)</code></td>
<td>the set of S-pts of Kind <code>k</code> which are accessible to the robot when at the B-pt <code>b</code></td>
</tr>
<tr>
<td><code>grasp(b)</code></td>
<td>returns the grasp Kind associated with B-pt <code>b</code></td>
</tr>
</tbody>
</table>

**Table 1.** Meaning of literals generated from the placement graph (S-pt, B-pt, Path, Zone, Kind are variable types)

finite map from Terms to Values. An example state is `{loc(Cup1) = s3, loc(Cup2) = s5, holding(Cup1) = true, CURR = b1}`.

To support queries such as `obstr` and `path` the Formula Generator generates logical assertions, called literals, that represent the information in the placement graph. For example each edge between b-points `B_i` and `B_j` is represented by a literal `isEdge(B_i, B_j)`. The complete list of such generated literals is shown in Table 1. Currently these are all statically generated literals, but in future we expect to implement an API to support dynamic querying of a lazily constructed placement graph. The terms B-pt, S-pt, Rgn, Zone, Kind, and Path refer to built-in types in Robosynth. The formal definition of the remaining actions is in Appendix 1. Next we show how to regress a goal back through such an action specification.

5.4. **Regressing a goal back through an action.** To understand how goal regression works, consider again the simple one line plan outline and goal (`holding(Cup1) ∧ loc(Cup2) = s2`) given earlier. Intuitively, one can see that anything established by the postcondition of the action (such as `holding(Cup1)`) should be removed from the goal, and anything else that is not (such as `loc(Cup2) = s2`) should be propagated. The presence of variables (such as the loop variable `o` representing an unknown object) complicates things a bit, but nevertheless this basic idea can be formalized and automated. Details are in Appendix 1). By following such a rule, the following `wp(P,G)` can be automatically computed:

\[
\text{loc}(\text{Cup1}) \in \text{reach}(\text{bpt}, \text{CupKind}) \\
\land \nexists o' \cdot \text{blocks}(\text{loc}(o'), \text{loc}(\text{Cup1})) \land \nexists o' \cdot \text{holding}(o')
\]

Informally, this says that in order for the robot to be holding the cup after the pickup action, there must be a b-point `bpt` from which the robot can access that location, which must not be blocked by some other object `o'`, and the robot is not currently holding anything.

5.5. **Regressing goals back through entire plans.** The regression of goals through more complex plan outlines than single actions can be done by composing the regression through actions. The rules for doing this are with respect to the language syntax so we first provide an abstract syntax of the main imperative part of the Robosynth language:

\[
\text{Stmt ::= Assgt | Action | Stmt-Seq | Conditional} \\
\text{Stmt-Seq ::= Stmt ; Stmt-Seq | }\epsilon\text{ } \\
\text{Conditional ::= if (BooleanExpr) Stmt [else Stmt]} \\
\text{Asst ::= Var ::= Expr} \\
\text{Action ::= pickup(Obj,BPt) | place(Obj,Rgn,Bpt) | ...}
\]
The following rules for regressing a goal through constructs in the above syntax were introduced by Dijkstra (apart from “Action” which was considered earlier). Their utility stems from the fact that they can be automatically applied to any program made up of the given constructs to determine the **wp** for the entire program above

\[
\begin{align*}
wp(S; T, Q) &= wp(S, wp(T, Q)) \\
wp(\text{if } B \text{ then } S \text{ else } T; Q) &= (B \Rightarrow wp(S, Q)) \wedge (\neg B \Rightarrow wp(T, Q)) \\
wp(x=a; Q) &= Q[x \mapsto a] \\
wp(\text{action}, Q) &= (Q \triangleright a.\text{post}) \wedge a.\text{pre}
\end{align*}
\]

where \(Q[x \mapsto a]\) is the result of replacing all free occurrences of \(x\) in \(Q\) with \(a\) and \(Q \triangleright \). The first rule says that the weakest pre-condition required to achieve \(Q\) after a sequence of statements \(S\) and \(T\) is the weakest pre-condition that must hold before \(S\) in order to establish the weakest pre-condition required by \(T\) alone to establish \(Q\). The second rule says that the weakest pre-condition required to achieve \(Q\) after a conditional statement is the weakest pre-condition required to establish \(Q\) after the “if” branch in the case that \(B\) is \(true\) and the “else” branch in the case that \(B\) is \(false\). The third rule is for the base case of an assignment statement. It says that in order to establish \(Q\) after an assignment \(x := E\), where \(E\) is some expression, \(Q\) with \(E\) substituted for \(x\) must hold before the assignment. Dijkstra also introduced a rule for calculating the weakest pre-condition of a while loop. However, the rule is not completely mechanical as it requires the programmer to supply a loop invariant. We instead unfold while loops a bounded number of times. The details are in Appendix 1.

Finally, note that weakest precondition is derived completely independently of a specific initial state of the physical space (the initial scene). Let the formula \(I\) capture constraints on the variables of \(P\) that describe this initial scene. Then the formula \(I \wedge wp(P, G)\) defines the largest set of valuations of the unknowns of \(P\) such that the plans corresponding to these valuations: (a) respect the initial conditions defined by the scene description; and also (b) meet the goal. As mentioned earlier, this formula is fed to an SMT solver (Z3) which returns values for the unknowns if the formula is satisfiable.

### 6. Correctness of Robosynth

We would like to know both that instantiating the plan outline with values for the unknowns returned by Robosynth results in a correct plan w.r.t. a given placement graph (soundness) and that if there is an instantiation of the plan that can achieve the goal, Robosynth will find it (completeness). We will do this by borrowing a standard technique for solving a similar problem in programming language semantics [48]. There, in order to show the correctness of a w.p. definition of a language, it is shown equivalent to a suitably abstracted interpreter (called a denotational definition) for that language. Although we utilize the same basic approach, our situation is slightly complicated by the fact that we produce a model for the w.p. of the plan outline which is then used by the interpreter. Thus we want to show that a model or state is returned by our solver iff the interpreter running the plan outline instantiated with that state and starting from that state reaches a final state that satisfies the goal. For us the interpreter corresponds to a plan execution engine.

An interpreter is abstracted as a function \(\mathcal{E}\) which takes a plan (instantiated plan outline) and an initial state and returns the state that holds from executing that plan from the initial state. Since the interpreter operates over a global placement graph \(R\) we write it as \(\mathcal{E}^R\) rather than making the graph a separate argument. Note that the argument to the interpreter is assumed to be grounded syntax, i.e. instantiated with some state assignment. As an example, consider the meaning of a sequence of plans \(S\) and \(T\) defined as follows (a fragment of syntax is enclosed in double square brackets):

\[
\mathcal{E}^R[S; T](s) = \mathcal{E}^R[T](\mathcal{E}^R[S](s))
\]

This says that running the interpreter on the sequence \(S; T\) starting from a state \(s\) is equivalent to running the interpreter on \(S\) from state \(s\) and using the state that results from that subplan as the starting state for the interpreter on the second subplan \(T\). Similar rules exist for conditionals as well as for actions. The complete denotational definition is in an associated technical report [49].

Now the following definitions are required.
Definition 6.1. A planning domain \( D \) is a pair \((R, A)\) where \( R \) is a placement graph and \( A \) is a set of pre/post condition action definitions.

Definition 6.2. A plan \( \pi \) is correct for state \( s_i \), constraint \( k \), initial condition \( i \), goal \( g \), and planning domain \( D \) if \( s_i \models_D i \land k \) and \( E[D][\pi](s_i) \models g \).

Finally, the following theorem expresses the correctness of our plan synthesis approach: Note that a plan outline is always instantiated into a plan before being passed to the interpreter.

Theorem 6.3. Given an initial condition \( i \), a goal \( g \), a plan outline \( p \), constraint \( k \), a planning domain \( D \), and any state assignment \( s \), \( p_s \) is correct for \( s, i, g \), \( k \) and \( D \) is \( s \models_D i \land k \land wp(p, g) \).

Proof. See Appendix 6 □

7. Results

7.1. Implementation and Setup. The Constraint Generator (Fig. 3.1) was implemented in F# and utilizes Z3 4.3.10 [10] as the SMT solver. The scenes are based on CAD files imported into the MoveIt [50] manipulation planning library, and depicted in RViz [51] for a simulated PR2. Individual motion plans were computed using OMPL [43]. The plan output by the Plan Extractor (see Fig. 3.1) is fed to a simple C++ interpreter which also makes calls on MoveIt. All experiments were carried out in Ubuntu Linux on a 16 core 2.8 GHz machine with 30 GB of memory.

In all the experiments below, the placement graph was first generated from the scene description shown in Fig. 3.2. Apart from variations such as the number of s-points and property labels on edges, the same basic structure was retained for all experiments.

7.2. Qualitative Evaluation. We have shown how to solve a more general problem than the one in [3] by moving several dishes, and efficiently handling a limited dishwasher capacity. But perhaps more importantly we allow the programmer to constrain the solution returned by Robosynth at the motion planning level, specifically the values returned for path, location, or region unknowns. We illustrate the power of our approach by offering some examples of how the programmer can use the constraint mechanism to obtain paths that satisfy different user-level requirements.

Experiment 1. Consider a very simple plan outline in which the robot is asked to move from the Dishwasher to the Storage to pickup a cup there, and a placement graph in which there are several paths between the Dishwasher b-point and the Storage b-point, corresponding to different paths in the kitchen workspace, numbered 1 through 3 in Fig. 3.2. In the complete absence of any constraints, the solver returns a path in the placement graph which corresponds to path 1 (also shown in red). In the presence of a safety requirement, this path may be considered undesirable because it passes through the food preparation area FoodPrep (shaded in Fig. 3.2). A zone constraint \( \text{crosses(path, FoodPrep)} \) forces Robosynth to find an alternative path, for example one that goes around the Island (path 2 shown in blue). But such a path might be considered too long if there are power or time limitations on the robot. A stronger constraint that also adds \( ||\text{path}|| \leq 10 \) (where distance is measured in workspace units) ensures Robosynth returns path 3, shown in green.

Experiment 2. Another aspect of our plan outline language is that it allows a degree of latitude to the programmer in how much information can be omitted without paying an undue penalty in plan synthesis time. Consider a simpler version of the example problem in Section 3 in which there are no other dishes obstructing the DIRTY dishes to be moved nor are any of the DIRTY dishes obstructing each other. Suppose DIRTY consisted of the collection Cup1, Cup2, Cup3. Then the following straight-line plan outline that moves each DIRTY dish in order is perfectly fine:

```plaintext
fetch(Cup1, Countertop, ?pathDC);
put(Cup1, Dishwasher, ?pathDC);
fetch(Cup2, Countertop, ?pathDC);
put(Cup2, Dishwasher, ?pathDC);
fetch(Cup3, Countertop, ?pathDC);
put(Cup3, Dishwasher, ?pathDC);
```
From a traditional task planning perspective, there is no actual planning required here as the actions are all sequenced. Robosynth takes 5 seconds to find a value for $\text{pathDC}$, and return a plan for this problem. If however, some DIRTY dishes may be obstructing each other, then it is not clear exactly which order the cups should be moved, and in that case a non-sequential for loop can be used (Appendix 1)

\begin{verbatim}
for dish in! DIRTY
{
  fetch(dish, Countertop, $\text{pathDC}$);
  put(dish, Dishwasher, $\text{pathDC}$);
}
\end{verbatim}

in which the exact item to be moved at each iteration is represented by the loop variable dish because it is unknown. This requires slightly more effort from Robosynth because it has to determine the correct order in which to remove the dishes. However, Robosynth still takes 5.4 seconds to return a plan for this problem. We can go one step further. By replacing the location Countertop with $\text{somewhere}$, this allows DIRTY dishes to be located anywhere in the kitchen. Robosynth still takes only 5.4 seconds to return a plan for this problem.

Of course, in the most general case, if the programmer supplies nothing other than a goal and an unordered bunch of action statements with unknowns, then the behavior devolves into that of a SAT planner [24, 25]. But if the programmer supplies at least the desired actions and the order, then as can be seen, we can do much better.

7.3. Quantitative Evaluation.

Experiment 3. In this experiment, the example plan outline in Fig. 3.3 was run on a varying number $n \leq 9$ of dirty dishes (number of items in the set DIRTY) and a varying number $l \leq 60$ of locations (s-points) where they may be placed, for different Dishwasher capacity values (3, 6, and 9). The experiment was carried out with a random selection of 10 of the possible assignments of the additional $l - m$ locations to the regions in the scene description (Countertop, Dishwasher, and Storage) and 10 of the possible initial locations of objects to locations, and the median value taken.

Fig. 7.1 shows the time taken to construct the initial PRM (Sec. 4.4.1). The observed behavior is quite reasonable, given the possible worst-case quadratic behavior of Alg. 1.

Fig. 7.2 shows that the time required by the synthesis engine scales well for $n \leq 6$. We observe that the solver struggles the most with a dishwasher capacity of 6. We suspect this is an artifact of how the SMT solver goes about determining the number of iterations of the inner and outer loops (since we have no notion of optimality regarding the combined number of such iterations). We also observe that when the dishwasher capacity is increased, the average synthesis time for a given number of objects also increases. We hypothesize that due to the nature of SMT solvers, having too many available locations in the wrong region can also slow down the solver as it has to examine many alternatives that lead nowhere.

We also believe that when space is tight, either due to a larger number of objects or due to a small number of locations, the solver becomes sensitive to the exact region in which the additional locations are available (i.e. Dishwasher, Countertop, or Storage). For example, if space is not tight in one region the inclusion of additional locations in that region may not make very much of a difference to the solution time. Thus the distribution of locations can make a huge difference to the solving time, as the large variances for 20 and 30 placement locations (top-right plot in Fig. 7.2) and for 9 objects demonstrates (middle-right and bottom-right plots in Fig. 7.2). We surmise that in some cases the SMT solver can quickly find a solution and in others will have to search extensively to find few available locations.

Finally, Fig. 7.3 shows how the synthesis time relates to the ratio of number of locations to number of objects. We observe that once that ratio gets high enough (no more than 17), the synthesis time falls drastically, and remains essentially flat thereafter. Although this observation is limited to the lower numbers of objects (i.e. 3 or under) because our experiments do not allow us to conclude anything beyond this, we conjecture that such behavior may be related to the phase transition from hard to easy problems observed in SAT solvers [52].

In general, we wish to point out that many of the observed variations in times across similar looking inputs is a consequence of the underlying solver we are using (Z3). As such solvers are solving an NP-complete problem, somewhat idiosyncratic behavior is to be expected. However, such solvers are also currently the
focus of a massive research and engineering effort aimed at boosting their performance, and we expect to be able to continue to leverage their growing power.

Although these results are encouraging, further experimentation is needed to draw any definitive conclusions about trends beyond this data.

Experiment 4. This experiment was used to show how synthesis time was affected by the size of the constraint. The original constraint in the example plan outline in Fig. 3.3 was successively augmented with a constraint on path2, path3, etc.

@invariant(||?path1|| <= 10) &
~crosses(?path1, FoodPrep)) &
(||?path2|| <= 10) &
~crosses(?path2, FoodPrep))
...

The results are shown in Table 2. Again we observe good performance.

8. Discussion

We have described a tool Robosynth for expressing and solving problems in integrated task and motion planning that arise with mobile manipulators, and have evaluated our tool on a benchmark problem (the Kitchen domain). Our main contributions are a novel way of solving the ITMP by transforming the problem into suitable input for an SMT solver, a language for plan outlines allowing the programmer to supply known plan information, and. a novel abstraction of the manipulation graphs we call placement graphs.

Some key difference between our work and other work are that first we leverage the growing power of automated solvers. Although we currently use an SMT solver, we are not tied to any particular solving technique. Once the constraints have been generated they could, for example, be fed to a CSP solver which uses specialized techniques such as arc-consistency propagation and backjumping [53] to efficiently solve the constraints. Secondly, unlike other most other approaches, we accept programmer input. As the problem we are solving is NP-complete (see Appendix 2), it may appear that there is no advantage to having the
programmer supply a plan outline given that a standard SAT planner supplied with a placement graph could probably also solve the problem with the same complexity. However it should be borne in mind that this is simply a worst case. In practice, many of the arguments to the actions are known to the programmer as is the ordering of the actions. This in turn relieves the solver of the burden of having to determine action ordering and many arguments. Additionally, the ordering information in the plan outline imposes constraints on the solver which ensure a more efficient solution as highlighted in Section 3.1 earlier. Finally, our abstraction of manipulation graph avoids both the combinatorial explosion that arises in some other approaches as well as indiscriminate discretization.
We are currently refining the language to allow greater expressibility as well as investigating several improvements. The placement graph can be computed lazily to avoid computing low-level paths that are never considered by the task level planner. Finally, the placement graph can be grown adaptively: Using constraint propagation techniques [29, 54] it is possible to significantly reduce the number of local paths that are generated. We are also investigating the use of feedback from the motion planning level to the solver to help guide its search.

Numerical quantities other than path length, including real-values, such as fuel consumption or time taken are possible though we have not currently implemented them.

One drawback of approaches (including ours) that compute solutions ahead of time, as opposed to approaches that re-plan, is that if an unexpected event occurs, the robot is stuck. To address this problem,
Wong et al. [55] modify the synthesis process for LTL specifications to generate alternative actions. In addition, they incorporate runtime learning so that when the alternative actions fail to achieve the goal, the additional assumptions on the environment are incorporated and a new controller synthesized. Similar ideas could be applied to our work.

An issue we plan to address is the lack of run-time sensing. If the robot cannot determine the global state based on local state information, it will need to sense in order to decide how to proceed. For example, if, in running a Kitchen example, the dirty dishes (DIRTY) were not known ahead of time, but a sensor indicated whether or not a dish was dirty, our current approach of unfolding a loop a bounded number of times would not work and we would need to consider true unbounded while loops, which will require inferring invariants, a currently active research area.

REFERENCES

Appendix 1: Overview of the Input Language of Robosynth

The core language of Robosynth consists of actions which are composed using the basic operations of sequencing and conditionals. However, for usability, a richer language is provided which provides constructs such as for loops, while loops, repeat loops, and events. This appendix describes how the richer constructs in the language of Robosynth are transformed into simpler ones and also the informal descriptions of the base actions.
8.1. **Preprocessing.** The more complex constructs which are allowed in the Robosynth language, such as *for* loops, *while* loops, and event handlers, are transformed into the simpler constructs given in the previous section by a pre-processor. A simple rewrite engine traverses the AST produced by the parser, applying each transform in turn.

**For Loops.** For loops in Robosynth are bounded so they are simply expanded by instantiating the loop body for each value of the iteration variable. That is

\[
\text{for } i \text{ in } [i_1,i_2,\ldots,i_n] \text{ do } S
\]

becomes

\[
S[i_1 \mapsto i_1]; S[i_2 \mapsto i_2]; \ldots S[i_n \mapsto i_n];
\]

E.g., the *for* loop

\[
\text{for dish in } \text{DIRTY\_DISHES do}
\]

\[
\text{wash(dish)};
\]

given \(\text{DIRTY\_DISHES} = \{\text{Cup1,Plate1,Cup2}\}\) (defined in the scene file) is replaced by the following sequence

\[
\{\text{wash}(\text{Cup1}); \text{wash}(\text{Plate1}); \text{wash}(\text{Cup2})\}
\]

Robosynth also supports *non-sequential for* loops. Such loops are used when the programmer does not know or want to specify the order in which each item in the collection is processed. A non-sequential *for* loop is identical to a sequential one except for the use of ‘!in’ instead of ‘in’. The unfolding creates a new unknown for each iteration and an extra assertion (command) is added at the end. E.g.,

\[
\text{for dish in! } \text{DIRTY\_DISHES do}
\]

\[
\text{wash(dish)};
\]

becomes

\[
\{\text{wash}(\text{dish1}); \text{wash}(\text{dish2}); \text{wash}(\text{dish3})\}
\]

\[
\text{assert}(\text{dish1,dish2,dish3}\Rightarrow \text{DIRTY\_DISHES})
\]

The solver finds values for \(\text{dish1, dish2, and dish3}\).

**While Loops.** While loops are also syntactically expanded using the equivalence “\(\text{while } B \text{ do } S\) ≡ “if \(B\) then \(S\); while \(B\) do \(S\)” As this is a non-terminating expansion, this is only carried out some fixed number of times (which can be set by the programmer, and currently defaults to 5). Sketching [16] takes a similar approach. Clearly this approach is not correct when the loop condition is unbounded. This is discussed in Sec. 8.

**Repeat Loops.** Repeat loops are also syntactically expanded using the equivalence “\(\text{repeat } S\) ≡ “if ?\(B\) then \(S\); repeat \(S\)” where \(?\(B\)\) is a fresh boolean variable. As this is a non-terminating expansion, this is only carried out some fixed number of times (which can be set by the programmer, and currently defaults to 5).

**Event Handlers.** Finally, event-handlers are syntactically pre-processed. Given a pre-event handler

\[
A(\text{param}_1,\ldots,\text{param}_n) : S
\]

the instantiated body \(S[\text{param}_i \mapsto \text{arg}_i]\) (for every \(i\)) is inserted before every matching action \(A(\text{arg}_1,\ldots,\text{arg}_n)\) in the plan outline. E.g. given the rule

\[
\text{@pre: pickup(o,r,?):}
\]

\[
\text{if obstructs(obst,o) then pickup(obst,r,u).}
\]

the following action in the body of the plan outline

\[
\text{pickup(Cup1,?rgn,?path)}
\]

is expanded to

\[
\text{if obstructs(obst,Cup1) then pickup(obst,rgn,u); pickup(Cup1,?rgn,?path)}
\]
8.2. **Actions.** The base construct in Robosynth is the action. An action specifies a primitive step that, once its arguments have been instantiated, can directly be translated into a sequence of calls that a robot can carry out. These actions have already been introduced in the plan outlines given earlier.

\( \text{moveTo}(rgn, path) / \text{moveTo}(bpt, path) \). If there is a path \( path \) from the current robot base position to a base position \( b \) from which the robot can access some location in the region \( rgn \) then when the action is executed, the new base position of the robot is \( b \).

\( \text{pickup}(obj, bpt) \). If the robot at b-point \( bpt \) can access the current location of the object \( obj \) and that location is not blocked by some other object, and the robot is not currently holding anything, then when the action is executed, the robot is holding \( obj \).

\( \text{place}(obj, rgn, bpt) \). If the robot can access the object \( obj \) located in region \( rgn \) from b-point \( bpt \) and the object location is not blocked or occupied by some other object, and the robot is holding \( obj \), and the grasp at the current location and at \( b \) match, then when the action is executed, the new base position of the robot is \( b \) and the robot is not holding anything.

Finally there is an action \( \text{assert(booleanExpr)} \) which has the effect of asserting the given boolean expression.

The actions \( \text{fetch}(obj, bpt, path) \) and \( \text{put}(obj, rgn, bpt, path) \) translate as \( \text{moveTo}(bpt, path); \text{pickup}(obj, bpt) \) and \( \text{moveTo}(bpt, path); \text{place}(obj, rgn, bpt) \) resp.

### Appendix 2: Complexity Analysis

Our approach relies on transforming the problem of finding values for the unknowns in a plan outline into one of Boolean satisfiability, which is known to be NP-Complete, and unlikely to have polynomial time algorithms. It is therefore reasonable to ask whether using an exponential-time algorithm for the problem is justified. In this section, we answer the question in the affirmative. We show that using just the imperative subset of Robosynth, ie., no constraints, we can encode the requirement for solving a well-known NP-Complete problem called the Hamiltonian Path Problem. The Hamiltonian Path Problem is the problem of deciding whether there is a path in a given graph which visits every vertex in the graph only once. We first define what we mean by a planning input and problem.

**Definition 8.1.** An ITMP input is a 5-tuple \( (D, p, k, i, g) \) where \( D \) is a planning domain, \( p \) is a plan outline, \( k \) is an invariant, \( i \) is an initial condition, and \( g \) is a goal condition.

(A planning domain was defined in Def. 6.1).

**Definition 8.2.** The planning problem given a planning input \( (D, p, k, i, g) \) is the problem of deciding whether there is a state assignment \( S \) to the unknowns of \( p \) such that \( p_S \) is correct for \( S, D, k, i, g \).

**Theorem 8.3.** The ITMP problem is NP-Complete

**Proof.** Given a plan outline and a state assignment \( s \) it is obvious that \( P[s'] \) can be verified in time linear in the length of \( P \) so the problem is in NP. To show the problem is NP-hard, we show a reduction from the Hamiltonian Path Problem. Given a graph \( (V, E) \) let every vertex \( v \in V \) be mapped to a b-point \( B_v \) and every edge in \( E \) be mapped to a edge between the corresponding b-points. (Thus the reduction is polynomial time). Let \( OBJS \) denotes a collection of \( n \) objects \( o_1, \ldots, o_n \) of kind \( K \). Associate a unique s-point \( s_i \) with reach of each \( B_i \) of kind \( K \), i.e. \( s_i \in \text{reach}(B_i, K) \) (see Table 1) and let the initial location of each object \( o_i \) be \( s_i \). Let there be one additional b-point separate from \( B_1, \ldots, B_n \) called the holding base point, \( B_H \) and associate \( n \) s-points within reach of \( B_H \) also of kind \( K \). Let this collection of \( n \) s-points be \( H \). Connect each \( B_i \) to \( B_H \) with a directed edge, and vice versa. There are no other nodes or edges in the placement graph. The original graph and corresponding placement graph are shown in Fig 8.1.

Now the original graph has a Hamiltonian path iff the following plan outline has a solution. Broadly speaking the plan outline is required to move each object (in some order to be determined) \( o_i \) from its location next to its corresponding b-point to a location next to \( B_H \) then return to the b-point \( B_i \) and from there move to the next b-point.
Suppose the original graph has a Hamiltonian path. W.l.o.g. let it be \( v_1, v_2, \ldots, v_n \). Now let a robot execute an instantiation of the above plan as follows: move to \( B_1 \) from which it can fetch \( o_1 \) located at \( s_1 \), put it at some location in \( H \) and return to \( B_1 \). Repeat this for b-points \( B_2, \ldots B_n \). Clearly when this plan finishes, each object is in the holding area. By Theorem 6.3, the solver returns a model for the given plan outline.

\[ \Rightarrow: \]

Suppose the solver returns a model. We need to show that the original graph possesses a Hamiltonian path. By Theorem 6.3 there is a corresponding feasible plan. The only such plan is one in which each object is picked up from its initial location accessible from some b-point \( B_i \), dropped off at the holding area, with the robot returning to \( B_i \), then moving to the b-point from which it accesses the next object. Each such b-point must be distinct since each object is accessible from only one b-point. Consider the path only between the vertex b-points, i.e. ignoring the path from each \( B_i \) to the holding area. W.l.o.g. let this be \( B_1 \rightarrow B_2, B_2 \rightarrow B_3, \ldots B_{n-1} \rightarrow B_n \). As the maximum path length (number of hops) is set to one, each path \( B_i \rightarrow B_{i+1} \) can be only one edge long so cannot be via any other b-point. Therefore each \( B_i \) is visited only once and the corresponding path \( v_1, v_2, \ldots, v_n \) in the original graph is Hamiltonian.

\[ \Leftarrow: \]

Suppose the solver returns a model. We need to show that the original graph possesses a Hamiltonian path. By Theorem 6.3 there is a corresponding feasible plan. The only such plan is one in which each object is picked up from its initial location accessible from some b-point \( B_i \), dropped off at the holding area, with the robot returning to \( B_i \), then moving to the b-point from which it accesses the next object. Each such b-point must be distinct since each object is accessible from only one b-point. Consider the path only between the vertex b-points, i.e. ignoring the path from each \( B_i \) to the holding area. W.l.o.g. let this be \( B_1 \rightarrow B_2, B_2 \rightarrow B_3, \ldots B_{n-1} \rightarrow B_n \). As the maximum path length (number of hops) is set to one, each path \( B_i \rightarrow B_{i+1} \) can be only one edge long so cannot be via any other b-point. Therefore each \( B_i \) is visited only once and the corresponding path \( v_1, v_2, \ldots, v_n \) in the original graph is Hamiltonian.

\[ \square \]

**Appendix 3: Transforming the PRM into a Placement Graph**

This Appendix describes how an initial PRM constructed as described in Sec. 4.4 is transformed into a placement graph. As an example, Fig. 8.2 (a) shows a PRM in which the properties of interest are zones, \( L, M, \) and \( N \), shown as the dotted regions in the figure and also by the shading of the nodes. Therefore each node is labeled with one of these labels (indicated by the pattern on the node: shaded, hatched, and dotted resp.). The algorithm shown in Alg. 2 can be used to construct the placement graph.
Algorithm 2: Constructing a placement graph

**Input:** A labeled PRM

1. for each label \( l \): /* e.g. labels \( L, M, N \) in Fig. 8.2 */
2. \( G_l := \) subgraph of the PRM such that \( G_l \) nodes are labeled \( l \);
3. \( l_1, l_2, \ldots, l_n := \) strongly connected components (SCCs) of \( G_l \); /* Fig. 8.2(b) */
4. for every pair \( (l_i, m_j) \) of SCCs where \( l \) and \( m \) are labels
5. if exists some node \( a \in l_i \), some node \( b \in m_j \) s.t. connected\((a, b)\)
6. then
7. path := \([a, p_1, q_1, \ldots, r, b]\) where \((a, p_1), (p_1, q_1), \ldots, (r, b) \in \text{edges}(G_i)\);
8. /*If such a path exists, this means that there is a path between every node in \( l_i \) and every node in \( m_j \) (because the components are strongly connected)*/
9. \( l_p, l_q, \ldots, l_r := \) SCCs through which \( p, q, \ldots, r \) pass;
10. \( E := \emptyset \);
11. for every edge \((p, q)\) in path
12. \( E += (p, q, d) \) where \( d \) is the distance between the two nodes \( p, q \) in the PRM; /* Fig. 8.2(c) */
13. for every pair \((b, b')\) of b-points in the PRM
14. paths := all paths between \( b \) and \( b' \) traversing edges in \( E \);
15. for every path \([b, p_1, d_1], (p_1, q_1), \ldots, (r, b'), d_k]\) in paths
16. \( G.edges += \) labeled base edge \((b, b', [l_b, l_p_1, l_q_1, \ldots, l_r, l_b'], \Sigma_{i=1}^{k} d_i)\) where \( l_x \) is the label of the SCC to which node \( x \) belongs;
17. /*If multiple edges exist with the same label sequence retain only the shortest. See Fig. 8.2(d)*/
18. \( G.edges += \) local edges and blocking edges discovered by the local planner;

**Output:** A placement graph \( G \).

---

**Figure 8.2.** Construction of the placement graph (see Alg. 2 for details)
Paths in the PRM between two b-points that have the same sequence of labels (minus stuttering) on their nodes are called indistinguishable. Our algorithm ensures that only one path among a set of indistinguishable ones is actually returned in the placement graph. For example, path $p_1 \rightarrow p_3 \rightarrow p_6 \rightarrow p_5 \rightarrow B_2$ in Fig 8.2(a) is indistinguishable from path $B_1 \rightarrow p_4 \rightarrow p_5 \rightarrow B_2$ in that they both have the same sequence of labels ($L, M, N, L$) on their nodes, so the latter does not appear in the placement graph. However, both $B_1 \rightarrow p_4 \rightarrow B_2$ and $B_1 \rightarrow p_4 \rightarrow p_5 \rightarrow B_2$ are retained because the former does not include the label $N$.

**Appendix 4: Weakest Preconditions for Actions**

8.3. **Formal definitions of actions in Robosynth.**

8.3.1. **Domain Independent Actions.**

`moveTo(b:BPt, u:Path)`

- **pre:** $\text{path?}(u, \text{CURR}, b)$
- **post:** $\text{CURR} = b$

`pickup(o:Obj, b:BPt)`

- **pre:** $\text{loc}(o) \in \text{reach}(b, \text{kind}(o))$
  $\land \exists o' \cdot \text{obstr?}(\text{loc}(o'), \text{loc}(o))$
- **post:** $\text{holding?}(o)$

`place(o:Obj, b:BPt, s:SPt)`

- **pre:** $\text{loc}(o) \in \text{reach}(b, \text{kind}(o)) \land \exists o' \cdot \text{obstr?}(\text{loc}(o'), s)$
  $\land \exists o' \cdot \text{loc}(o') = s \land \text{holding?}(o)$
- **post:** $\text{loc}(o) = s \land \neg \text{holding?}(o)$

`assert(bexp:BExpt)`

- **pre:**
- **post:** $bexp$

Note that the action definitions make use of the Frame Axiom [21] which states that any property that is not explicitly changed by an action continues to hold its previous value.

8.3.2. **Domain Specific Actions.** Domain-specific actions (e.g. “run(Dishwasher)”) are defined the same way as the built-in actions, by use of domain-specific fluents (e.g. `clean?`) provided by the domain definer. For example, an action to run a dishwasher has the following definition:

`run(dishwasher)`

- **pre:**
- **post:** $\forall o \in \text{dishwasher} \cdot \text{clean?}(o)$

8.4. **Calculating weakest preconditions.** Note that a quantifier-free first order formula can always be expressed as a disjunction of conjunctions of literals, called disjunctive normal form (DNF). If the formula contains quantifiers it is converted into another normal form called prenex normal form in which all the quantifiers occur up front, and then the body is converted to DNF. In the problems we deal with, as in classical planning, all actions are deterministic. Therefore $A.post$ is always a conjunction, which can be represented as a set of literals.

For postconditions other than simple conjunctions (which could arise either from an indefinite goal or as the weakest precondition of a conditional statement) assume the postcondition $Q$ is in DNF. Let $Q = \bigvee_i Q_i$ where each $Q_i$ is conjunction. Then since all statements are deterministic, for any program body $S$

$$wp(S, \bigvee_i Q_i) = \bigvee_i wp(S, Q_i)$$

Each disjunct on the right hand side is converted into DNF. Therefore there is never a need to determine $wp$ w.r.t. to more than a collection of literals. In a simple programming language, where the base case is just the assignment operator, the weakest pre-condition is simply the post-condition with a subsituation applied to it. Here more complex relationship exists between a postcondition and the corresponding wp. Specifically, the weakest precondition of $A$ w.r.t. to some postcondition $Q = \{L_1, L_2, \ldots, L_n\}$, $wp(A, \{L_1, L_2, \ldots, L_n\})$ is

$$(\{L_1, L_2, \ldots, L_n\} \Rightarrow A.post) \land A.pre$$
This is saying that \( A.post \) first logically simplifies the set of literals, removing any conditions that are subsumed by \( A.post \). This is essential to ensuring that the condition that results is indeed the weakest precondition. Then the additional conditions required by \( A \) are conjoined. The simplification is carried out by the \( \gg \) operator, described next.

### 8.4.1. The \( \gg \) operator

Each literal \( L \) is either an atom (e.g. \( \text{holding}(\text{Cup1}) \)) which can be written as \( \text{holding}(\text{Cup1}) = \text{true} \) or already an equality (e.g. \( \text{loc}(\text{Cup1}) = s2 \)). Suppose \( A.post \) establishes \( \{\text{holding}(\text{Cup1}), \text{CURR} = B2\} \) and \( Q \) is \( \{\text{holding}(\text{Cup1}), \text{path}?(B1, B2, \text{path}4)\} \). Then since \( \text{holding}(\text{Cup1}) \) is established by \( A.post \) it can be removed from \( Q \). That is \( Q \gg A.post \) is \( \{\text{path}?(B1, B2, \text{path}4)\} \). But when the object being moved is an unknown (for example in the for loop of Fig. 3.3) \( A.post \) will be \( \{\text{holding}(o), \text{CURR} = B2\} \). Now whether or not the holding literal can be removed from \( Q \) depends on whether \( o \) gets bound to \( \text{Cup}1 \) or not. Since this is not determinable at VC generation time, a conditional literal is generated. A conditional literal takes the form “if \( B \) then \( P \) else \( Q \)” where \( P,Q \) are (possibly conditional) literals. This is just \( B \rightarrow P \land \neg B \rightarrow Q \), which we sometimes abbreviate to \( B \gg P : Q \).

The update operator \( \gg_L \) between a pair of literals is recursively defined as follows

\[
l \gg_L l' = \begin{cases} 
  t = t' \land v = v' : l & \text{if } (l, l') \text{ are of form } \langle f(t) = v, f(t') = v' \rangle \\
  B? (m \gg_L l') : (n \gg_L l') & \text{if } l \text{ is of form } B? m : n \\
  t = t'? p(v) : l & \text{if } (l, l') \text{ are of form } \langle p(t), t' = v \rangle \\
  l & \text{otherwise}
\end{cases}
\]

The explanation of the cases is as follows:

1. A literal \( f(t) = v \) can only be removed if there is another literal \( f(t') = v' \) with the same function symbol operating over the same argument. In such a case the values of the right hand sides must also match otherwise the result is an error. If there is no such other literal, the literal continues through into the precondition. Eg. if \( l \) is \( \text{holding}(o) = \text{true} \) and \( l' \) is \( \text{holding}(\text{Cup1}) = \text{true} \) then \( l \) can be removed (is satisfied) if there is a model in which \( o \) is assigned \( \text{Cup1} \). On the other hand if \( l' \) were \( \text{holding}(\text{Cup1}) = \text{false} \) the resulting condition would be \( \text{false} = \text{true} \) which is \( \text{false} \) leading to an unsatisfiable precondition. This is because the postcondition of some action (represented by the literal \( l' \)) is contradicting the current goal being regressed.

2. When \( l \) is a conditional literal, which branch the update is applied to depends on whether \( B \) evaluates true or not. Eg. if \( l \) is \( o = \text{Cup1}? \text{loc}(o) = s1 : \text{loc}(o) = s2 \) and \( l' \) is \( \text{loc}(\text{Cup1}) = s3 \) then if \( o \) is assigned \( \text{Cup1} \) in some model, then the value of the expression is what results from evaluating \( \text{loc}(o) = s1 \gg \text{loc}(\text{Cup1}) = s3 \), otherwise \( \text{loc}(o) = s2 \gg \text{loc}(\text{Cup1}) = s3 \). The entire expression is thus the conditional literal \( o = \text{Cup1}? (\text{loc}(o) = s1 \gg \text{loc}(\text{Cup1}) = s3) : (\text{loc}(o) = s2 \gg \text{loc}(\text{Cup1}) = s3) \).

3. When \( l' \) is a literal of the form \( t' = v \) and \( l \) is a literal containing some term \( t \) then \( t \) may be substituted in \( l \) by \( v \) if \( t = t' \). E.g. if \( l \) is \( v, \text{loc}(o), \text{Dishwasher} \) and \( l' \) is \( \text{loc}(\text{Cup1}) = s \) then the result is \( v, s, \text{Dishwasher} \) in a model in which \( \text{loc}(o) = \text{loc}(\text{Cup1}) \) and \( \text{in}(\text{loc}(o), \text{Dishwasher}) \) otherwise.

4. The final case simply passes \( l \) on without change.

The update operator can now be lifted to conjunctions \( \varphi, \varphi' \) by defining \( \varphi \gg \varphi' \) as the cross product of every literal in \( \varphi \) with every literal in \( \varphi' \) using \( \gg_L \), ie \( \bigcup_{l \in \varphi, l' \in \varphi'} \{l \gg_L l'\} \)

**Example 8.4.** Let \( \phi \) be \( \{\text{loc}(\text{Cup1}) = s1, \text{holding}(\text{Cup1}) = \text{true}\} \) and \( \phi' \) be \( \{\text{obstr}?(s, \text{loc}(c)), \text{holding}(c') = \text{true}\} \) then \( \phi \gg \phi' = \{c = \text{Cup1}?\text{obstr}(s, \text{loc}(\text{Cup1})): \text{obstr}(s, \text{loc}(c)), c' = \text{Cup2}?\text{true} : \text{holding}(c') = \text{true}\} \)
Appendix 5: A Denotational Definition

\[
E^R[a](s) = \begin{cases} 
   s \oplus a.post & \text{if } s \models a.pre \\
   \emptyset & \text{otherwise}
\end{cases}
\]

\[
E^R[S;T](s) = \begin{cases} 
   E^R[T](E^R[S](s)) & \text{if } E^R[B](s) \\
   E^R[T](s) & \text{otherwise}
\end{cases}
\]

where the \( \oplus \) operator requires that its second argument is interpreted as a unique state. Since \( a.post \) is a conjunction this is always possible. First define the operator \( \oplus_A \) between a pair of state assignments \( a, a' \)

\[
a \oplus_A a' = \begin{cases} 
   a' & \text{if } \langle a, a' \rangle \text{ are of form } (f(t) = v, f(t) = v') \\
   \{ t \mapsto t' \} & \text{if } \langle a, a' \rangle \text{ are of form } (p(t), t = t') \\
   a & \text{otherwise}
\end{cases}
\]

As before \( \oplus_A \) can be lifted to state arguments by defining \( s \oplus s' = \bigcup_{a \in A, a' \in A'} \{ a \oplus a' \} \) similar to before. The method calls on \( R \) to refer to the predicates and functions supported by the placement graph (Table 1). Boolean (and, or, not), relational \( (<, >, =) \), and arithmetic \( (+, -, \|) \) expressions have their usual declarative semantics so we do not list them here.

Appendix 6: Proof of Theorem 6.4

**Lemma 8.5.** A plan \( \pi_1: \pi_2 \) is correct for state \( s_i \), initial condition \( i \), goal \( g \), invariant \( K \) and planning domain \( D \) iff \( \pi_1 \) is correct for \( s_i, i, \text{goal } g' \), \( K \) and \( \pi_2 \) is correct for \( E[\pi_1](S), true, G, K, \) and \( D \) where \( g' \) is some predicate that holds in state \( E[\pi_1](S) \)

**Proof.** From the denotational definitions for sequences, and the definition of plan correctness.

The expression \( s \models_R p \) states that state \( s \) is a model for \( p \) modulo the theory generated from the placement graph \( R \). Also \( p_s \) for a plan outline \( p \) is the result of instantiating \( p \) with the the state assignments in \( s \). Note that a plan outline is always grounded into a plan before being passed to the interpreter \( E \).

**Theorem 6.4**

Given an initial condition \( i \), a goal \( g \), a plan outline \( p \) with invariant \( K \), a placement graph \( R \), and any state assignment \( s, p_s \) is correct for \( s, i, g, K, \) and \( R \) iff \( E \models_R i \land K \land wp(p,g) \)

**Proof.**

\[\leq\]

Suppose \( s \models i \land K \land wp(p,g) \) for some \( s, g, p \). We proceed by structural induction on \( p \). The proof refers to the rules for the formal semantics and weakest preconditions of actions (Appendix 4) and the denotational semantics (Appendix 5).

**case P=\text{P1;P2}:** \( wp(P1;P2,G) = wp(P1,wp(P2,G)) \) so \( S \models wp(P1,wp(P2,G)) \). By the IH, the theorem holds for \( P1 \). Therefore \( E[P1](S) = wp(P2,G) \land K \) . Let \( S' \) denote \( E[P1](S) \) . Again by the IH, this time for \( P2 \), \( E[P2](S') \land G \land K \) . As \( E[P1;P2](s) = E[P2](E[P1](s)) \) this proves the result.

**case P=\text{if B then P1 else P2}:** From the w.p. definitions, \( wp(\text{if } B \text{ then } P1 \text{ else } P2, Q) = (B \rightarrow wp(P1,Q)) \land (\neg B \rightarrow wp(P2,Q)) \). Consider the case where \( B \) holds in the interpretation \( S \), that is \( B_S \) is true. Therefore \( E[P1;\text{else } P2](S) \) takes the value \( E[P1](S) \) . By the IH, the theorem holds for \( P1 \) and the result is proved. The case where \( B \) is false in \( S \) is identical.

**case P=\text{a (for some action a)}:** \( S \models i \land K \land (G \gg a.post) \land a.pre \). We need to show \( S \models i \land K \) and \( E[a](S) \models G \land K \) . Consider the various cases for the definition of the operator \( \gg_L \) on each pair of literals from the cross product of \( G \) and \( a.post \).
(1) if \( \langle l, l' \rangle \) are of the form \( \langle f(t) = v, f(t') = v' \rangle \) then \( S \models t = t' \land v = v' \): let which is equivalent to \( S \models (t = t' \rightarrow v = v') \) and \( S \models (t \neq t' \rightarrow l) \). Since \( S \models a.pre \) by assumption above, the value of \( E[a_S](S) \) is \( S \oplus a.post \) which from the definition consists of the union of all the pairs of assignments \( a \oplus a' \) where \( a' \) is drawn from the state (model) for \( a.post \). Consider the case where \( a' \) is \( l' \). In the case \( t \neq t' \) then regardless of whether \( f(t) = v \in S \) the matching case in the definition of \( \oplus \) is the 3rd one, the result is just \( S \), which establishes the result as \( S \models G \) by assumption above. If \( S \models t = t' \) then \( S \models v = v' \). Now if \( f(t) = v \notin S \) then the matching case for \( S \oplus a.post \) is the 3rd case, establishing \( G \) as before. Otherwise the matching case is the first one, i.e. the value is \( f(t) = v' \) but as \( v = v' \), this is again just \( S \) which establishes \( G \).

(2) if \( l \) is of the form \( B? m : n \) then \( S \models B? (m \gg l') : (n \gg l'). \) It is sufficient to show that \( S \oplus a.pre \models l \). The proof is by induction on the structure of the conditional literals. Suppose \( S \models B \) then \( S \models I \land K \land (m \gg l') \land a.pre \) which by the IH leads to a state \( S' \) which satisfies \( m \). But since \( B \) also holds in \( S' \), this means \( S' \models l \) from the definition of a conditional literal. Similarly for \( S \models \neg B \).

(3) if \( \langle l, l' \rangle \) are of the form \( \langle p(t), t' = v \rangle \) then \( S \models t = t'? p(v) : l \). Consider the case where \( S \models t = t' \). Now the matching case in the definition of \( S \oplus a.post \) is the 2nd one, i.e. \( p(t \mapsto t') \) which is just \( p(t) \) so \( S \) is unchanged and therefore entails \( G \). If \( S \models t \neq t' \) then \( S \oplus a.post \models S \) by definition and therefore again entails \( G \).

(4) otherwise, \( l \) is passed through and \( G \) is not changed. Therefore \( S \models G \) as required.

\[ \Rightarrow \]

Suppose \( S \models I \land K \) and \( E[\pi](S) \models G \land K \) for some \( S \).

**case** \( P = P1; P2 \): By the denotational rule for sequences, \( E[P1; P2](s) = E[P2_S](E[P1_S](s)) \). Let \( S' = E[P1_S](S) \) i.e. \( E[P1; P2](s) = E[P2_S](S') \). By Lemma 8.5, \( P2 \) is correct for \( S', true, G \).

By the IH, the theorem holds for \( P2 \). Therefore \( S' \models wp(P2, G) \land K \). By Lemma 8.5, \( P1 \) is correct for initial state, initial condition \( S, I \) and the goal \( wp(P2, G) \). By the IH on \( P1 \), therefore, \( S \models wp(P1, wp(P2, G)) \). Since \( wp(P1; P2, G) = wp(P1, wp(P2, G)) \), the result follows.

**case** \( P = if \ B \ then \ P1 \ else \ P2 \): By the denotational rule for evaluating conditionals, \( E[if \ B \ then \ P1 \ else \ P2 \](s) \) is \( E[P1_S](s) \) if \( E[B_S](S) \), and \( E[P2_S](S) \) otherwise. Consider the first case. As \( P \) is correct for \( S, I, G \), so must also \( P1_S \). By the IH therefore \( S \models I \land K \land wp(P, G) \). Also, \( E[B_S](S) \) iff \( B \).

Therefore \( B \) holds under the interpretation \( S \). Therefore \( B \rightarrow wp(P1, G) \) is true under \( S \). The other case is similar.

**case** \( P = a \) (for some action \( a \)): As above. □