Open access • Proceedings Article • DOI:10.1109/ICRA.2014.6906924

SMT-based synthesis of integrated task and motion plans from plan outlines — Source link

Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri ...+1 more authors

Institutions: Rice University

Published on: 29 Sep 2014 - International Conference on Robotics and Automation

Topics: Motion planning, Graph (abstract data type), Satisfiability modulo theories, Plan (drawing) and Solver

To appear in the Proceedings of the 2014 IEEE Intl. Conf. on Robotics and Automation (ICRA2014)

SMT-Based Synthesis of Integrated Task and Motion Plans from Plan Outlines

Srinivas Nedunuri[2], Sailesh Prabhu[2], Mark Moll[1], Swarat Chaudhuri[2] and Lydia E. Kavraki[1]

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 now used to symbolically explore the space of all integrated plans that correspond to 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.

I. INTRODUCTION

Integrated task and motion planning (ITMP) [1]–[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—embodied in a system called ROBOSYNTH—to ITMP.

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. The task level planner has to search a space that is exponential in the number of actions required to achieve a goal, while the continuous planning problem is PSPACE-complete in the degrees of freedom of the robot [5]. Unsurprisingly, the seamless integration of these two levels is difficult. A strictly hierarchical approach where the task planner operates on an abstraction and passes the solution to a continuous motion planner does not always work: it either sacrifices completeness or requires extensive backtracking, which can be highly time-consuming. While we do not solve the above problem in its entirety, we have a minimal framework in place that enables us to demonstrate our approach.

Our approach to solving the problem is based on two key ideas: (1) we request a limited amount of specification from the user of our system; 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 through an intuitive interface, we can prevent the planning algorithm from searching through plans that are obviously unacceptable.

As for the latter idea, SMT-solvers [6]–[8] 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 [6], [9]. This is because large, even infinite, sets of system states can be compactly represented as formulas in first-order logic. This allows us to frame problems of state space search using a small number of calls to an SMT-solver. In practice, this formulation often leads to dramatic improvements in the scalability of state space analysis. In the present paper, we use 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. It also specifies stable locations for the robot and the objects including their grasps. Although this latter information is currently supplied manually, this is not intrinsic to our approach, as much of it can be automatically determined (see Sec. VI). The plan outline allows the programmer to

The authors are with the Dept. of Computer Science, Rice University, Houston, TX 77251. {nedunuri | snp3 | mmoll | swarat | kavraki}@rice.edu

[1]Work supported in part by NSF NRI 1317849 and NSF 1139011 grants
[2]Work supported in part by NSF Award 1162076 and NSF CAREER Award 1156059

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 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 be less than 10m long"*. 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 with nodes that correspond to the pre-defined configurations of the robot and possible locations for objects. Edges in the graph represent robot actions that are feasible between the given configurations and to and from the locations. We call this graph a *placement graph.* Note that the placement graph is computed *once per scene* and is common to all plan outlines in that scene (Sec. VI 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 resolves to computing a satisfying solution of this constraint. This is done using Z3 [7], a state-of-the-art SMT-solver.

We have evaluated ROBOSYNTH on a version of the ITMP problem that generalizes prior work by Kaelbling and Lozano-Pérez [1]. 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 [1] and ROBOSYNTH proves to be able to solve this problem effectively under a rich variety of requirements and changes to the physical space. Numerous extensions of our work can be considered, and these are left for future work.

The main contributions of this paper are as follows: (1) We apply SMT solvers to the problem of carrying out integrated planning in the presence of constraints. (2) To integrate the discrete and continouous levels, we introduce a novel abstraction of manipulation graphs we call placement graphs. (3) Finally, in order to control the search space we provide an intuitive programming interface in which the programmer can provide known control information. In our implementation of the above ideas, we have made some basic choices in order to demonstrate the concept. Several of these choices could be improved and we address some of these in Sec. VI.

The rest of the paper is organized as follows. Sec. II presents related work. In Sec. III, we introduce our benchmark example and use it to illustrate the inputs and outputs of our approach. In Sec. IV, we describe the internals of the ROBOSYNTH system. Sec. V presents our experimental results. We conclude with some discussion in Sec. VI.

II. RELATED WORK

A. Program Synthesis

Our method is influenced by prior work in the programming language community on *template-based program synthesis* [10], [11]. 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. The difference between the two settings is that 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.

B. Task Planning

Automated planners have a long history in AI [12]. Prominent among these are heuristic planners such as FF [13]. 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 [14]. A number of approaches to ITMP make use of such automated planners in combination with motion planning [3], [15], [16] and this is discussed further in the next subsection. Automated planners that rely on Boolean satisfiability solvers [17], [18] 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. 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.

C. Integrated Task and Motion Planning

In recent years, several approaches have been proposed to integrate the motion planning level with task planning. In [2] a form of SAT planning is used that incorporates 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. ASYMOV [3] 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, [4] uses a Hierarchical Task Network (HTN) planner [12]. 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. Schemas can be viewed as grammars defining a language of legal plans. Thus HTN planners share with our approach the input of domain knowledge. However, they require a level of domain expertise in order to correctly, completely, and efficiently codify the space of possible plans. In contrast, our approach accepts partial user knowledge sufficient to solve the given problem, expressed in a programming language format. Feedback from the tool tells the programmer whether or not the plan outline they provided can solve the given problem.

A different form of hierarchy is used in Hierarchical Planning in the Now (HPN) [1] 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 [19]), it rests on the assumption that an action can always be reversed. This is not always true. For example, an egg once fried, cannot be unfried, or if a robot is preparing meals in a restaurant, there may be several dishes that need to be ready to go at the same time. Even if actions can be reversed, reversing too many actions, although theoretically possible, may be practically infeasible. For example, if the robot runs on a battery the total time allowed for carrying out a plan may be limited.

Although our work tackles a similar problem 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 are the following:

- It is not necessary for the user to designate a specific "safe region" for where to place removed obstructions.
- The ROBOSYNTH language allows a programmer to place constraints on the paths that are returned by the solver.
- With HPN, 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.
- Because HPN relies on re-planning, it *requires* run-time sensing even when the environment does not change. We do not require run-time sensing, and have left reactivity to future work.

In [16] a different approach is taken to the problem of interfacing a discrete task planner with the underlying continuous world by Skolemizing the continuous variables. While avoiding unnecessary discretization, the 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, both the built-in and domain-specific

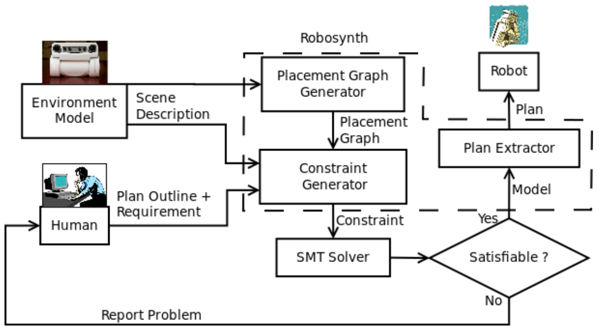

Fig. 1. Architecture of ROBOSYNTH

actions in ROBOSYNTH are defined in the same way using pre- and post- conditions.

[20] takes 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. An empty range indicates an unsatisfiable task plan. While this and other extensions (such as reactivity) could be incorporated, they are outside the scope of this paper, which is primarily focused on our use of automated constraint solvers for solving the ITMP.

D. Controller Synthesis

Another major research effort has been the automatic synthesis of both reactive [21], [22] and non-reactive controllers [23] and plans [24] for robots from temporal logic specifications. Tools such as LTLMOP go beyond what we do, in that they accept structured temporal specifications written in "natural language" style and handle reactive robotics. The primary difference between these approaches and ours is that we use a *symbolic* method, where a space consisting of a vast number of plausible integrated plans can be represented concisely in the form a logical constraint, and a solver is invoked to perform operations on such constraints using a small number of calls. Such symbolic approaches to analysis of large combinatorial spaces can be often dramatically more scalable than methods relying on explicit enumeration of the space, even when that space is compactly represented using, for example, BDDs. An extension of our approach to reactive robotics is left for future work.

III. 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 approach are described in Sec. IV.

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.

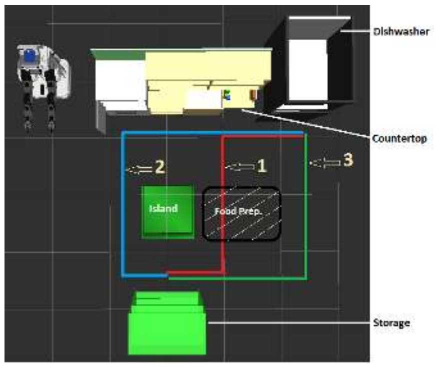

Fig. 2. Example kitchen domain (overhead view). Three alternative paths between Dishwasher and Storage are shown

```
1. #import KitchenDomain
2. #import KitchenScene
3.
4. Path path, path1, path2, pathR;
5. Region tempR, somewhere; Location loc1;
6.
7. void main()
8. { for o in! DIRTY do
9..  { findPlace(?loc1,Dishwasher);
10.     pickup(o,?somewhere,?path1);
11.     place(o,?loc1,?path2);
12.   }
13.   run(Diswasher);
14.   for o in DIRTY do
15.   ... //move dishes from Dishwasher to Storage
16.}
17.@goal: clean(DIRTY) & contains(Storage,DIRTY)
18.@invariant (||?path|| <= 10) &
19.              ~crosses(?path,FoodPrep))
```

Fig. 3. Plan outline for the Dirty Dishes problem

A. Example: Planning for a robot in a kitchen

Consider a household robot operating in a kitchen environment. Kaelbling and Lozano-Perez [1] 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. (Information about what kinds of objects in which locations can block access to other locations is input as part of the process of creating the scene description).
- 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.

Input 1*. Scene Description:* The scene description is provided in two files, called the *domain* and the *scene*. The domain represents that 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. 2. At the bottom of the figure is the Storage area; At the top of the figure are the Countertop and Dishwasher areas in that order. 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. 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.

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 *pick up* a dish (assume this is done by an action `pickup`) before it attempts to `place` 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 shows the essential parts of a plan outline for the example problem. Lines 1 and 3 import the domain and scene information respectively. Lines 4 and 5 declare a few variables; these are the unknowns to which ROBOSYNTH will assign values. The types of these variables are `Path`, `Location`, and `Region`, corresponding respectively to the types of robot paths, locations of objects, and sets of possible object locations. Lines 8-16 constitute the body of the `main` procedure. This code specifies the high-level knowledge that in any reasonable plan, the robot must perform the following actions in sequence:

1) Repeatedly carry out the following steps:
   a) Locate a place `?loc1` in the dishwasher for the object through an action `findPlace`. As the location is unknown to the programmer, the variable for the place has a "?". The action `findPlace`, along with actions `pickup` and `place` for picking up and placing objects, are predefined in the language.
   b) Follow an (as yet unknown) path `?path1` from the robot's current location to an (unknown) area `?somewhere` and `pickup` an (unknown) dirty item `?o` located there.
   c) Follow the unknown path `?path2` to the dishwasher and place the item in there.
2) When all the dirty items are loaded, `run` the dishwasher.
3) Unload the dishwasher and place all the items in storage. The code for this is similar to the first for-loop.

Input 3*. Requirements :* Requirements come in the form of a *goal* for the planner, and *invariants* that must hold along the planned paths. The goal (line 17) states that all the dirty dishes defined in the scene (contained in the set `DIRTY`) must be clean and put away in Storage. Examples of invariants 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 invariants in our example are that: (1) the total length of any path is constrained to be under 10 units; and at no point must the robot go through the food preparation area. The specification of these properties uses functions like $\|\|$ for path length, and predicates like `crosses` (`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. We use an SMT solver rather than just a SAT solver because of the presence of linear arithmetic and functions in the invariant.

Events: We could enrich the above requirements and the plan outline further. In particular, we offer 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. The natural way to handle such "exceptions" is to have an "event handler" fired before and after each action. The programmer can set up such event handlers to carry out whatever corrective action is needed. In Robosynth, code for these event handlers is written as follows:

```
20.@pre-handler:  pickup(obj,rgn,_):
21.     while (obstructs(?obst,obj))
22.     { pickup(?obst,rgn,?pathR);
23.       place(?obst,?tempR,?pathR);
24.     }
```

This code, which defines the pre-event handler for the pickup action, moves any obstacles out of the way to an unknown safe region (`?tempR`) that Robosynth determines automatically.

B. Plan Synthesis

The inputs described above, namely the scene description, plan outline, and requirements, are fed to Robosynth as shown in Fig. 1. Robosynth computes the placement graph and then uses that to compute values for the unknown variables in the plan outline (for example, `?loc1` and `?path1`) 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 IV-A. 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. 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 Sec. V.

Finally, we wish to draw the reader's attention to the following in the plan outline for the example: (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 (`?obst`) in the while loop of the event handler (Lines 21-24). (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 10,11), Robosynth is forced to find an efficient way of moving the dishes (cups then plates).

IV. Internals of Robosynth

In this section, we describe the internals of Robosynth. We start by discussing the low level (box labeled "Motion planner" in Fig. 1), which requires the use of a motion planning algorithm. Next we describe the discrete level (box "Constraint generator" in Fig. 1), which operates on a discrete abstraction of the scene, and makes use of an SMT-solver.

A. Low-level Planning

The usual way of representing manipulation planning problems is with a manipulation graph [25]. 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. To facilitate integrated planning for a mobile manipulator we will introduce a variant of the manipulation graph called a *placement graph*. 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 an object resting on a support surface and the robot potentially holding that object in a stable grasp. The s-points are defined for each kind of object (cups, plates, etc.). Edges in the placement graph are allowed among b-points and between b-point and s-points. 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 a path from s-point j to its parent b-point. A sampling-based planner is used to find a feasible path between j and its parent b-point. The information required to compute b-points and s-points, such as object location, robot location, and grasp is currently manually identified in the scene description. In future much of this information will be computed automatically and possibly adaptively while the graph will be computed lazily [26], [27]. An example placement graph for the kitchen domain is shown in Fig. 4. The b-point and s-point names begin with "b_" and "s_", respectively. Solid lines depict edges between b-points, dashed lines depict edges between b-points and s-points, and dotted arrows show which s-points are blocked by which other s-points. The graph is constructed by reading in the

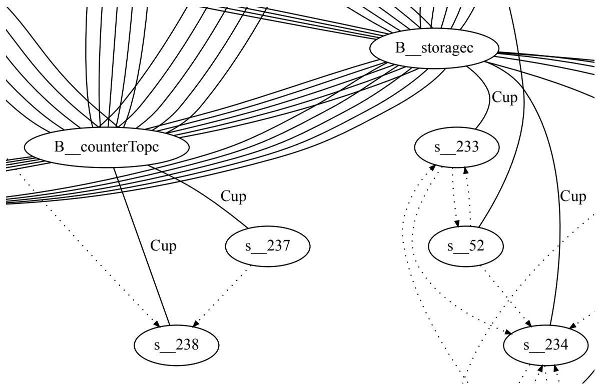

Fig. 4. A section of the placement graph for the kitchen scene description

scene description which is why it appears as one of the inputs to the Constraint Generator in Fig. 1

The primitives *moveTo*, *pickup*, and *place* that are used in plan outlines specify locations in terms of *regions* (e.g., Dishwasher, Cooking) rather than specific configurations. The placement graph is used by ROBOSYNTH to find feasible b-points and s-points within these regions. As will be described in more detail below, the SMT solver will automatically compute all feasible paths in the placement graph that correspond to collision-free paths in the continuous space and that satisfy the plan outline. ROBOSYNTH internally keeps track of where each object is at any stage of the plan and when objects should be picked up or released.

B. The discrete level

The task at the discrete level of ROBOSYNTH is as follows: find an assignment to the unknowns in the given plan outline P such that by executing the integrated plan obtained by replacing the unknowns in P with this assignment, from the given initial scene, the requirements in the plan outline are satisfied.

To achieve this, it is necessary to (a) have a clear meaning or semantics for the plan outline and (b) given a plan outline determine the required initial conditions. Both these requirements are met by utilizing a style of program semantics called *weakest precondition semantics* [28][1].

Now we briefly discuss what weakest preconditions mean in our setting; for a general understanding of weakest preconditions, see [30]. Suppose we are given a plan outline P and a requirement g that must hold at the end of the integrated plans in which we are interested (such requirements are *goals* in the terminology of Sec. III — the notion can be generalized in a standard way to invariants). 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 most program analysis settings and also in ROBOSYNTH, this weakest precondition is represented as a (quantifier-free) first-order formula.

[1]A similar idea called *goal regression* was developed around the same time in AI Planning by Waldinger [29]. We use the weakest precondition approach because it is was developed specifically for programs.

Space does not permit a complete exposition of how $wp(P, g)$ is determined, but to get a feel for what such a condition looks like, consider the simplest example where P is a single action, e.g.,

```
    pickup(Cup1,Cooking,?path1);
```

and g is $holding(Cup1)$ ("Robot is holding Cup1"). Then $wp(P, g)$ is the set of states captured by the following condition:

$$\exists bpt \cdot loc(Cup1) \in Cooking \wedge loc(Cup1) \in reachOf(bpt) \wedge \nexists o' \cdot blocks(loc(o'), loc(Cup1)) \wedge path(path, CURR, bpt) \wedge \nexists o' \cdot holding(o').$$

Informally, this says that the cup $Cup1$ must be located in the $Cooking$ region, 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', there is a path $path$ in the placement graph between the current b-point $CURR$ and bpt, and the robot is not currently holding anything. Note that the precondition definition makes use of auxiliary functions and predicates loc and $holding$ which are computed from the input scene description and $reachOf$, $\in$, $path$, and $blocks$ which are answered by querying the placement graph. The semantics of the other actions *place*, *moveTo*, etc. as well as any domain-specific actions can be defined in a similar way.

The weakest precondition semantics gives rules by which the weakest precondition can be calculated for a sequence of statements, for an *if-then-else* statement and a *while* statement. By following these rules, a constraint representing the weakest precondition for the entire plan outline is automatically calculated by the Constraint Generator of ROBOSYNTH (Fig. 1).

Note that the weakest precondition is derived completely independently of the 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. Our objective of finding a satisfactory assignment to the unknowns in P thus amounts to checking the *satisfiability* of this formula.

Recall that the satisfiability of a formula means there is a set of assignments to each of the unknowns, called a *model*, which makes the formula true. For example the formula $p \wedge x > 2$ is satisfiable by a model $M = [p \mapsto true, x \mapsto 3]$, written as $M \models p \wedge x > 2$. On the other hand, the formula $x > 2 \wedge \neg(x > 1)$ is not satisfiable (under the standard interpretation of >). Satisfiability of quantifier-free formulas can be determined automatically by the SMT solver which returns a model in case the formula is satisfiable. The SMT solver we use is Z3 [7]. If Z3 finds the formula satisfiable it returns values for the unknowns.

V. Results

A. Experimental Setup

The Constraint Generator (Fig. 1) was implemented in F# and utilizes Z3 4.3 [7] as the SMT solver. The scenes are based on CAD files imported into the MoveIt! [31] manipulation planning library, and depicted in RViz [32] for a simulated PR2. Individual motion plans were computed using OMPL [33]. The plan output by the Plan Extractor is fed to an interpreter which also makes calls on MoveIt!. All experiments were carried out on Ubuntu Linux on a 3.1 GHz machine with 4 GB of memory.

B. Qualitative Evaluation

We have discussed how to solve a more general problem than the one in [1] not only by moving several dishes, but more importantly by allowing 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. 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. 2. In the complete absence of any constraints, the solver might return 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. 2). A zone constraint `~crosses(?path1,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 `& ||path|| <= 10` ensures ROBOSYNTH returns path 3, shown in green.

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 Sec. III in which all the DIRTY dishes are on the Countertop, there are no other dishes obstructing the DIRTY dishes to be moved nor are any of the DIRTY dishes obstructing each other. Then the following straight-line plan outline that moves each DIRTY dish in order is perfectly fine:

```
findPlace(?loc1,Dishwasher);
pickup(Cup1,Countertop,?pathDC);
place(Cup1,?loc1,?pathDC);
findPlace(?loc2,Dishwasher);
pickup(Cup2,Countertop,?pathDC);
place(Cup2,?loc2,?pathDC);
...
```

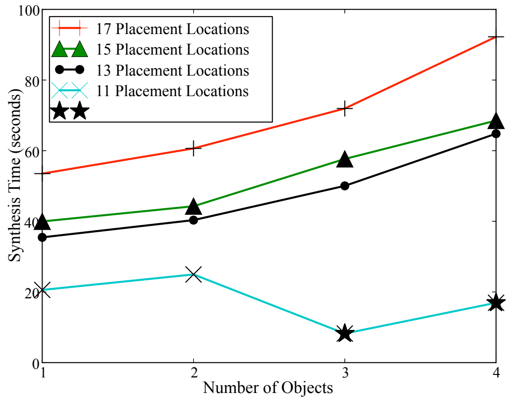

Fig. 5. Synthesis time with varying number of objects and locations. In two cases, the SMT solver returned UNSAT because there was no assignment to the unknowns that would allow the plan outline to fulfill the goal condition. These points are denoted with a star.

From a task planning perspective, there is no actual planning required here. ROBOSYNTH takes 37 seconds to find values for `?loc1`, `?loc2`, and `?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-deterministic `for` loop can be used (just as in lines 9 through 17 of Fig. 3 except with `Countertop` in place of `?somewhere`), in which the exact item to be moved at each iteration is represented by the loop variable `o` because it is unknown. This requires more effort from ROBOSYNTH because it has to determine the correct order in which to remove the dishes. However, ROBOSYNTH still takes 39 seconds to return a plan for this problem. We can go one step further. By making the outline read exactly like the `for` loop of Fig. 3 this allows DIRTY dishes to be located *anywhere* in the kitchen. ROBOSYNTH still takes only 41 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 [17], [18]. But if the programmer supplies at least the desired actions and the order, then as can be seen from the results, we can do much better.

C. Quantitative Evaluation

The `main` body of the example plan outline in Fig. 3 was run on a varying number of dirty dishes (number of items in the set DIRTY) and a varying number of locations where they may be placed. Fig. 5 shows that the time required by the synthesis engine does not seem unreasonable given the complexity of the problem. Fig. 6 shows the time required to compute the low-level paths for the placement graph as a function of the number of s-points (i.e., the number of possible object locations). Although these results are encouraging, further experimentation is needed to draw any definitive conclusions about trends beyond this data.

VI. 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

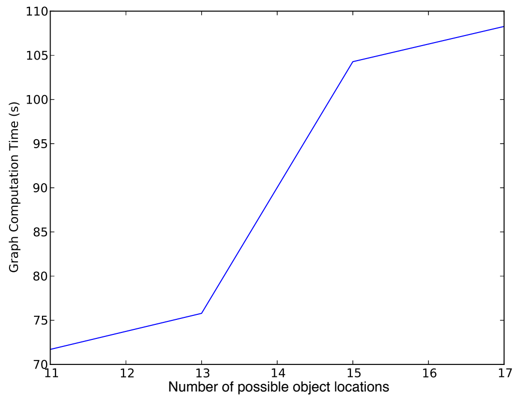

Fig. 6. Computation time of placement graph.

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.

We are currently refining the language to allow greater expressibility as well as investigating a tighter integration between the motion and task planning levels in order to provide better scalability. We will further reduce the human input that is currently required by automatically determining grasps and parent b-points given possible locations for the robot and objects. 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 [20], [34] 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.

References

[1] L. Kaelbling and T. Lozano-Pérez, “Hierarchical task and motion planning in the now,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2011, pp. 1470–1477.

[2] G. Havir, K. Haspalamutgil, C. Palaz, E. Erdem, and V. Patoglu, “A case study on the Tower of Hanoi Challenge: Representation, reasoning and execution,” in *IEEE Intl. Conf. on Robotics and Automation*, 2013.

[3] S. Cambon, F. Gravot, and R. Alami, “Overview of aSyMov: Integrating motion, manipulation and task planning,” in *Intl. Conf. on Automated Planning and Scheduling Doctoral Consortium*, 2003.

[4] J. Wolfe, B. Marthi, and S. Russell, “Combined task and motion planning for mobile manipulation,” in *Intl. Conf. on Automated Planning and Scheduling*, Toronto, Canada, 05/2010 2010.

[5] J. Canny, *The Complexity of Robot Motion Planning*. Cambridge, MA: MIT Press, 1988.

[6] L. De Moura and N. Bjørner, “Satisfiability modulo theories: introduction and applications,” *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[7] ——, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[8] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “The Barcelogic SMT solver,” in *Computer Aided Verification*. Springer, 2008, pp. 294–298.

[9] J. Rushby, “Harnessing disruptive innovation in formal verification,” in *IEEE Intl. Conf. on Software Engineering and Formal Methods, 2006. SEFM 2006*. IEEE, 2006, pp. 21–30.

[10] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, “Combinatorial sketching for finite programs,” in *Proc. of the 12th Intl. Conf. on Architectural support for programming languages and operating systems*, ser. ASPLOS XII, 2006, pp. 404–415.

[11] S. Srivastava, S. Gulwani, and J. S. Foster, “From program verification to program synthesis,” in *Proc. 37th ACM Symp. Principles of Prog. Lang. (POPL)*, 2010, pp. 313–326.

[12] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

[13] J. Hoffmann and B. Nebel, “The FF planning system: fast plan generation through heuristic search,” *J. Artif. Int. Res.*, vol. 14, no. 1, pp. 253–302, May 2001.

[14] “Ipc 2011 deterministic domains,” http://www.plg.inf.uc3m.es/ipc2011-deterministic/DomainsSequential.html.

[15] C. Dornhege, M. Gissler, M. Teschner, and B. Nebel, “Integrating symbolic and geometric planning for mobile manipulation,” in *IEEE Intl. Workshop on Safety, Security Rescue Robotics*, 2009, pp. 1–6.

[16] S. Srivastava, L. Riano, S. Russell, and P. Abbeel, “Using classical planners for tasks with continuous operators in robotics,” in *Intl. Conf. on Automated Planning and Scheduling*, 2013.

[17] H. Kautz and B. Selman, “Planning as satisfiability,” in *Proc. 10th European Conf. on Artificial Intelligence*, ser. European Conf. on Artificial Intelligence ’92. New York, NY, USA: John Wiley & Sons, Inc., 1992, pp. 359–363.

[18] J. Rintanen, “Engineering efficient planners with SAT,” in *European Conf. on Artificial Intelligence*, ser. Frontiers in Artificial Intelligence and Applications, L. D. Raedt, Ed., vol. 242, 2012, pp. 684–689.

[19] L. P. Kaelbling and T. Lozano-Pérez, “Planning in the know: Hierarchical belief-space task and motion planning,” in *Workshop on Mobile Manipulation, IEEE Intl. Conf. on Robotics and Automation*, 2011.

[20] F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson, “Constraint propagation on interval bounds for dealing with geometric backtracking.” in *IROS*, 2012, pp. 957–964.

[21] H. Kress-Gazit, T. Wongpiromsarn, and U. Topcu, “Correct, reactive, high-level robot control,” *Robotics Automation Magazine, IEEE*, vol. 18, no. 3, pp. 65–74, 2011.

[22] T. Wongpiromsarn, U. Topcu, and R. Murray, “Receding horizon control for temporal logic specifications,” in *Proc. of the 13th ACM Intl. Conf. on Hybrid systems: computation and control*, 2010, pp. 101–110.

[23] M. Guo, K. Johansson, and D. Dimarogonas, “Motion and action planning under LTL specifications using navigation functions and action description language,” in *Proc. of the Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2013.

[24] A. Bhatia, M. Maly, L. Kavraki, and M. Vardi, “Motion planning with complex goals,” *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 55 –64, Sep. 2011.

[25] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006. [Online]. Available: http://planning.cs.uiuc.edu/

[26] C. L. Nielsen and L. E. Kavraki, “A two level fuzzy PRM for manipulation planning,” in *In Proceedings of the International Conference on Intelligent Robots and Systems*, 2000, pp. 1716–1722.

[27] R. Bohlin and L. Kavraki, “Path planning using lazy PRM,” in *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 1, 2000, pp. 521–528.

[28] E. W. Dijkstra, *A Discipline of Programming*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1976.

[29] R. Waldinger, “Achieving several goals simultaneously,” in *Machine Intelligence 8: Machine Representations of Knowledge*, E. Elcock and D. Michie, Eds. Wiley, 1977.

[30] A. R. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*. Springer, 2007.

[31] S. Chitta, I. Şucan, and S. Cousins, “MoveIt!” *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 18–19, Mar. 2012. [Online]. Available: http://moveit.ros.org

[32] “Rviz: A 3d visualization tool for ros,” http://wiki.ros.org/rviz.

[33] I. Sucan, M. Moll, and L. Kavraki, “The Open Motion Planning Library,” *IEEE Robotics Automation Magazine*, vol. 19, no. 4, pp. 72–82, Dec 2012.

[34] A. K. Pandey, J.-P. Saut, D. Sidobre, and R. Alami, “Towards planning human-robot interactive manipulation tasks: Task dependent and human oriented autonomous selection of grasp and placement,” in *IEEE Intl. Conf. on Biomedical Robotics and Biomechatronics*, 2012.