International Conference on Automated Planning and Scheduling (ICAPS) 2016

Task and Motion Policy Synthesis as Liveness Games

Yue Wang and **Neil T. Dantam** and **Swarat Chaudhuri** and **Lydia E. Kavraki**
Department of Computer Science, Rice University, Houston, TX, 77005 USA
{yw27, ntd, swarat, kavraki}@rice.edu

Abstract

We present a novel and scalable policy synthesis approach for robots. Rather than producing single-path plans for a static environment, we consider changing environments with uncontrollable agents, where the robot needs a *policy* to respond correctly over the infinite-horizon interaction with the environment. Our approach operates on task and motion domains, and combines actions over discrete states with continuous, collision-free paths. We synthesize a task and motion policy by iteratively generating a candidate policy and verifying its correctness. For efficient policy generation, we use grammars for potential policies to limit the search space and apply domain-specific heuristics to generalize verification failures, providing stricter constraints on policy candidates. For efficient policy verification, we construct compact, symbolic constraints for valid policies and employ a Satisfiability Modulo Theories (SMT) solver to check the validity of these constraints. Furthermore, the SMT solver enables quantitative specifications such as energy limits. The results show that our approach offers better scalability compared to a state-of-the-art policy synthesis tool in the tested benchmarks and demonstrate an order-of-magnitude speedup from our heuristics for the tested mobile manipulation domain.

Introduction

Recently, there has been a growing interest in the integration of Task and Motion Planning (TMP) (Dornhege et al. 2009; Wolfe, Marthi, and Russell 2010; Bhatia et al. 2011; Kaelbling and Lozano-Pérez 2011; Erdem et al. 2011; Srivastava et al. 2014; Cirillo et al. 2014). These previous TMP approaches assume the environment is static. However, in domains involving uncontrollable agents, such as the scenario shown in Fig. 1, the robot needs a *policy*, rather than a pre-computed linear plan, to react to changes *online*, ensure safety, and accomplish desired tasks.

This paper presents a novel approach for the generation, or *synthesis*, of robot task and motion policies that is scalable and enables *quantitative specifications* through the use of symbolic constraints and an SMT solver (De Moura and Bjørner 2008). We extend our previous deterministic TMP system called ROBOSYNTH (Nedunuri et al. 2014a) with additional modeling and policy synthesis components to handle non-deterministic environments.

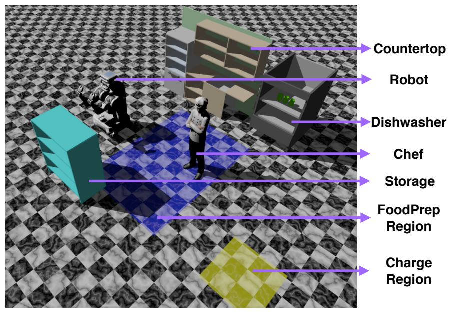

Figure 1: The robot must navigate through the kitchen and pick a cleaned dish from the dishwasher, while avoiding collisions with the chef. Since the chef's movement is uncontrollable, the robot needs a *policy* to accomplish the task no matter how the chef moves.

We formulate synthesis as a concurrent two-player game (de Alfaro and Henzinger 2000; Alur, Henzinger, and Kupferman 2002) over a discrete abstraction of the task and motion domain. In each round of the game, the robot and the environment choose their moves independently and simultaneously. Our method synthesizes a winning policy for the robot by iteratively generating a candidate and verifying its correctness (Solar-Lezama et al. 2006). We efficiently generate candidates by limiting the search space through a grammar of potential policies (Alur et al. 2013). To verify a generated policy candidate, we extend the proof rules of (Beyene et al. 2014), which provide compact, symbolic constraints that characterize the correctness of policies. We apply an SMT solver to check the validity of the constraints, and if they are valid, the policy is correct. Otherwise, we generalize verification failures by finding similar states using domain-specific heuristics. We then use these counterexamples to guide the subsequent policy candidate generation. This iterative policy synthesis procedure converges either to a winning policy or a proof that no such policy exists.

Other recent work has focused on reactive synthesis for robots and hybrid systems (Decastro and Kress-Gazit 2015; Wongpiromsarn, Topcu, and Murray 2010; Dantam and Stilman 2013; Ulusoy, Marrazzo, and Belta 2013). These approaches consider the differential dynamics of hybrid sys-

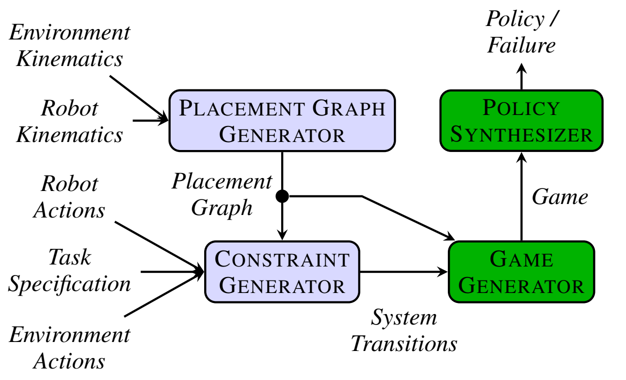

Figure 2: Overall structure of our policy synthesis approach. The *placement graph generator* and *constraint generator* (blue-colored components) function as in ROBOSYNTH (Nedunuri et al. 2014a; 2014b). We add two components (green-colored components): *game generator* and *policy synthesizer*, to handle uncontrollable environment behaviors.

tems but do not incorporate efficient path planning. In contrast, we focus on the mobile manipulation domain where high dimensionality makes efficient, collision-free path planning crucial, and we therefore apply fast, sampling-based motion planning methods (Kavraki et al. 1996). Furthermore, these previous works often perform a combinatorial search over large state spaces. In contrast, we avoid expensive combinatorial search using SMT-based symbolic methods, improving the scalability of policy synthesis.

We validate our approach in a mobile manipulation domain with human-robot interaction. Our results show that for the tested benchmarks, our method scales better than an alternate synthesizer (Piterman, Pnueli, and Saar 2006).

Problem Formulation

In this work, we consider task and motion policy synthesis for a controllable robot operating in an environment with uncontrollable agents such as humans. The robot and the environment have both continuous, kinematic state and purely discrete state. The robot can select actions to modify its own state, but it cannot control the state of the adversarial environment. Finally, we specify the desired task as a set of valid state sequences. Our goal is to synthesize a *policy* that guides the robot during execution to accomplish the task.

Fig. 2 shows the overall structure of our policy synthesis method, which extends our previous ROBOSYNTH system (Nedunuri et al. 2014a). The *placement graph generator* abstracts the continuous, kinematic state to a discrete *placement graph* (Nedunuri et al. 2014a), representing a subset of the continuous, collision-free configuration space. The robot actions are defined based on abstract *Planning Domain Definition Language* (PDDL) (Edelkamp and Hoffmann 2004) actions, such as `pickup(?object)`, the placement graph nodes (representing locations) and edges (representing reachability), and the specific objects in the scene. We use a *plan outline* (Nedunuri et al. 2014a), i.e., a parameterized sequence of robot actions with unknown parameters, to over-approximate the task specification set. A *valid* policy will determine the appropriate values for these unknown parameters such that resulting concrete plans achieve the task. Then the *constraint generator* transforms the placement graph, environment actions, robot actions, and task specification into a logical formula representing valid transitions of the system.

We formulate task and motion policy synthesis as a discrete, concurrent game between the robot and the environment:

Definition 1 (Concurrent Game)**.**
A concurrent game is a tuple $G = (\Sigma, \theta, \Gamma_e, \Gamma_r, \delta, \varphi)$:

- Σ is a state space of the game.
- $\theta \subseteq \Sigma$ is the set of initial game states.
- Γ_e, Γ_r are valid-move functions. For each state s, $\Gamma_e(s)$ and $\Gamma_r(s)$ represents the set of valid *moves* for the environment and the robot at state s, respectively.
- δ is the transition function of the game. For every state $s \in \Sigma$, *move* $a_e \in \Gamma_e(s)$ and *move* $a_r \in \Gamma_r(s)$, $\delta(s, a_e, a_r) \in \Sigma$ is the corresponding *successor* state.
  A *play* σ is an infinite sequence of states: $s_0, s_1, \ldots$, such that $s_0 \in \theta$ is an initial state and for $i \geq 0$, s_{i+1} is a successor state of s_i as defined by δ.
- φ is a set of *winning* plays for the robot.

The *game generator* formulates our domain as a concurrent game. The game state space Σ is the set of discrete states in the placement graph. The set of initial game states θ combines the robot's initial state set and the environment's initial state set. The environment valid-move function $\Gamma_e(s)$ and the robot valid-move function $\Gamma_r(s)$ are constructed from the robot actions and environment actions, respectively. The game transition function is constructed from the system transitions generated by the *constraint generator*. Finally, the winning condition φ is the set of valid state sequences in the *task specification*.

Now, we have reduced task and motion policy synthesis to the problem of finding a winning policy for the corresponding concurrent game. We define a concurrent game policy as follows:

Definition 2 (Concurrent Game Policy)**.**
A concurrent game *policy* for the robot is a function $p(s) \in \Gamma_r(s)$ that selects a next robot move for every state s.

Winning policies must satisfy the game's winning condition. The combination of concurrent game G and policy p defines a set of plays ψ, which are the state sequences that occur due to the robot actions selected by p and the possible environment actions in G. Policy p is *winning* for game G if every play in set ψ satisfies G's *winning* condition φ, i.e., $\psi \subseteq \varphi$.

In this work, we solve concurrent games with liveness winning conditions:

Definition 3 (Liveness Games)**.**
In a liveness concurrent game G, there is a set dst of goal states. A play σ is a *winning* play $\sigma \in \varphi$ if there exists a state s in the play σ that is a goal sate $s \in dst$. That is, a *winning* play should eventually visit a state $s \in dst$.

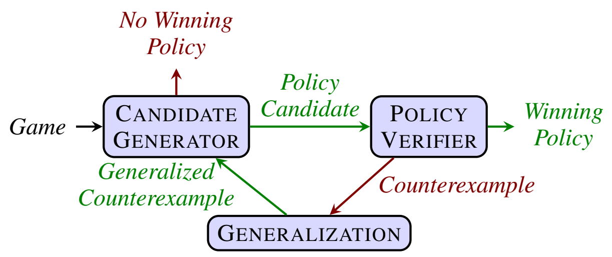

Figure 3: The core steps of the synthesis algorithm used in the *policy synthesizer* component in Fig. 2

Policy Synthesis

The policy synthesizer iteratively constructs a winning policy for a robot in a liveness game (see Fig. 3). First, the synthesizer generates a policy candidate based on a given grammar. Then, the synthesizer verifies the correctness of this candidate using an SMT solver to check symbolic constraints that characterize winning policies. Finally, we generalize verification failures via domain-specific heuristics to help subsequent policy candidate generation, reducing the number of necessary iterations and improving efficiency. This iterative policy synthesis procedure converges either to a winning policy or a proof that no such policy exists.

Candidate Generation We generate policy candidates using a grammar for potential policies. Specifically, our grammar for policies defines conditional expressions that choose actions based on the current state. This grammar is,

$$\begin{aligned} p(s) \rightarrow\ & \Gamma_r(s)[i] \\ & \mid \textbf{if } \text{test}(s) \textbf{ then } p(s) \textbf{ else } p(s) \end{aligned}$$

where $\Gamma_r(s)[i]$ is the ith valid *move* of the robot at state s, i.e., the ith element of valid-move set $\Gamma_r(s)$, and $\text{test}(s)$ is a first-order formula that defines a set of game states. This grammar limits the search space for policy candidates and improves performance (Alur et al. 2013).

We generate policy candidates from this grammar using an adaptation of the enumeration algorithm of (Udupa et al. 2013). At each iteration, the *candidate generator* systematically enumerates candidate policies from the grammar that are consistent with the generated counterexamples, in increasing order of expression size. If no such candidate exists, it reports that the game has no winning policy. Otherwise, it passes the candidate policy p to the *proof verifier*.

Policy Verification: Liveness Games The next step is to check if the candidate p is a *winning* policy. The *policy verifier* constructs symbolic constraints based on a new extension of the proof rules in (Beyene et al. 2014), and applies an SMT solver to verify the validity of these constraints.

In this work, we consider verifying policies for liveness games, where the robot should eventually reach a certain goal state. A liveness concurrent game $G = (\Sigma, \theta, \Gamma_e, \Gamma_r, \delta, \varphi)$ is associated with a set dst of goal states. In a winning play, there is at least one state that is a goal state $s \in dst$. Many robotic mobile manipulation problems can be modeled as liveness games. For example, the task shown in Fig. 1 can be modeled as a liveness game with goal states of robot holding a cleaned dish near the dishwasher.

Consider a set $inv \subseteq \Sigma$ that contains every state in a winning play and a set $round \subseteq \Sigma \times \Sigma$ that contains every pair of consecutive states before the *first* goal state $s \in dst$. The *winning* policies should guarantee that, for every $s \in inv$ and every environment *move* $a_e \in \Gamma_e(s)$, if s is not a goal state $s \notin dst$, the robot *move* $a_r \in \Gamma_r(s)$ selected by the policy $a_r = p(s)$ should lead to a successor state $s' = \delta(s, a_e, a_r)$ such that $s' \in inv$ and $(s, s') \in round$. To ensure a goal state is eventually reached, $round$ should be well-founded, i.e., there is no infinite sequence of states $\sigma = s_0, s_1, ...$, such that each pair of consecutive states (s_i, s_{i+1}) is a member of the set $round$, for all $i \geq 0$.

The following formula summarizes the above constraints:

$$\begin{aligned} & (\text{well-founded}(round)) \wedge \\ & (\ \forall s, a_e.\ (s \in inv) \wedge (s \notin dst) \wedge (a_e \in \Gamma_e(s))\ \rightarrow \\ & (\delta(s, a_e, p(s)) \in inv)\ \wedge\ ((s, \delta(s, a_e, p(s))) \in round)) \end{aligned} \tag{1}$$

Note that Formula 1 places assertions about every pair of consecutive states and the conjunction of all consecutive states is the complete play. Thus, for a play $\sigma = s_0, s_1, \ldots$, instead of specifying what should be satisfied in every state $s_0, s_1, \ldots$, Formula 1 specifies every pair of consecutive states and provides a much more compact representation of constraints. The size of the formula has a great impact on the performance of the SMT solver. Thus, these symbolic constraints greatly improves the performance of our method.

Our policy synthesizer constructs the well-founded set $round$ using the distance DistanceToGoal(s) between the current state and the goal region, i.e., a pair of states $(s, s') \in round$ if $\text{DistanceToGoal(s)} > \text{DistanceToGoal(s')}$. Intuitively, the well-founded set $round$ represents the requirement that the successor state s' should be closer to the goal region than the current state s. Since we consider bounded workspace, it is impossible to have an infinite sequence of states where the value of DistanceToGoal(s) keeps decreasing, which guarantees the well-foundedness.

We iteratively construct the auxiliary set inv:

1. Initially, we use the state space Σ to over-approximate the set inv, i.e., we set $inv = \Sigma$.

2. During the policy synthesis process, if we find an *invalid* state s that always leads to a successor state s' such that (s, s') is not a member of the set $round$, no matter what robot *move* the policy selects, we shrink the set inv by removing this *invalid* state s.

If we find a policy p with a *non-empty* set inv, then p is a *winning* policy that can always maintain a successor state closer to the goal region for every state $s \in inv$. Therefore, a goal state is eventually reached due to the well-foundedness of the set $round$. Otherwise, we can conclude that there is no winning policy for the robot because every state $s \in \Sigma$ will result in a successor state farther to the goal region, no matter what robot *move* the policy selects. Therefore, it is impossible for the robot to reach a goal state.

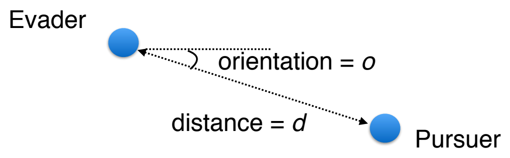

Figure 4: Illustration for one counterexample returned by the verifier, indicating that in this particular game state, the output of the candidate policy is incorrect. We exploit the geometric information, i.e., distance and orientation between the purser and evader to generalize this counterexample.

Generalization Formula 1 guarantees that for a candidate policy p, if the constraints are valid against all admissible environment behaviors, p is a *winning* policy. Otherwise, p is not a *winning* policy and the SMT solver returns a counterexample: a game state s that violates the constraints.

For robotic applications, we can utilize the geometric information learned from the low level motion planers to speed up the synthesis process. In this *generalization* step, we extract additional geometric information from this single counter-example to generalize this failure game state to a set of similar states. Such generalization reduces the number of policy candidates we must generate and verify, greatly improving the efficiency of our method (see Fig. 5).

To see how the generalization procedure works, consider a pursuit-evasion game where we synthesize a winning policy for the evader. The proof verifier discovers one particular game state where the output of the candidate policy is incorrect (i.e., the evader will be captured by the pursuer). Since the policy is represented symbolically and thus all the states with the same relative pose between the pursuer and evader ($\mathrm{distance} = d$ and $\mathrm{orientation} = o$) are assigned the same incorrect next move. Therefore, we also need to include such states into our counterexample set.

Experiments

Experimental Setup: Liveness Task Benchmark We evaluate our policy synthesizer in a kitchen environment (Fig. 1) with a simulated PR2 robot and uncontrollable agents (e.g., chefs) moving within the *FoodPrep Region*. The robot must avoid collisions with these chefs. The *liveness* task requirement is that the robot should eventually pick up a cleaned dish from the dishwasher. We assume the robot can sense the exact location of the chefs and the distance that the chefs can move between two sense actions is bounded.

This kitchen domain is a special instance of the more general pursuit-evasion games described in the previous section, where safety requires that the robot (evader) maintain a minimum distance from chefs (pursuers). In this experiments, we apply the same generalization heuristic for pursuit-evasion games to speed up the synthesis process.

The placement graph in our experiments models motion for the human agent (x, y), the robot base (x, y, θ), and one 7-DOF arm, giving 12-DOF total. We spent about 10 seconds on computing this placement graph, and we reused this graph for all policy computations. This graph contains about 80 nodes and 400 edges.

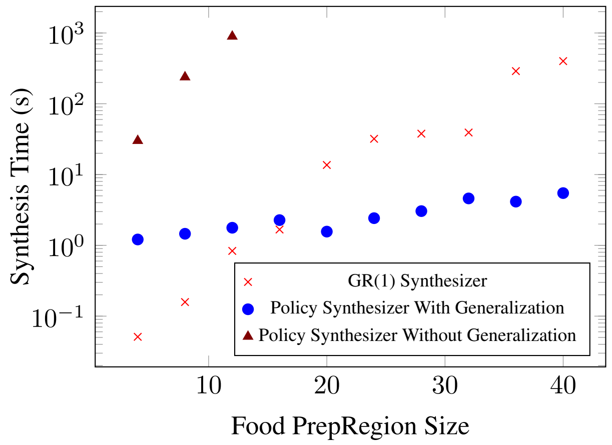

Figure 5: Performance of our policy synthesizer (with different settings) and LTLMoP back-end GR(1) synthesizer as the size of the *FoodPrep Region* increases (semi-log scale).

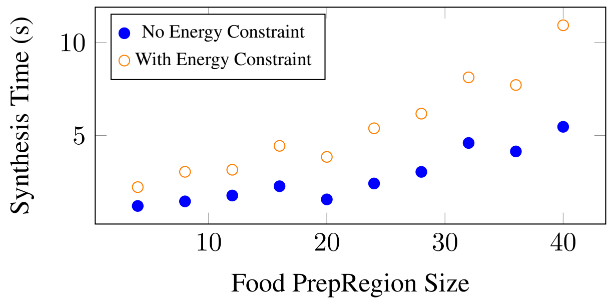

Figure 6: Performance of our synthesizer with energy constraint as the size of the *FoodPrep Region* increases. With energy constraint, our policy synthesizer still scales well.

All experiments were carried out on an 8 core 3.4 GHz machine with 8 GB memory. We use Z3 (De Moura and Bjørner 2008) as our backend SMT solver and utilize the *linear arithmetic* and *uninterpreted functions* theory solvers of Z3. For all benchmarks, the size of the workspace is fixed, and there are 2 chefs in the kitchen.

Evaluation We compare our policy synthesizer with the back-end of *LTL MissiOn Planner* (LTLMoP), a state-of-the-art synthesis tool for robotic applications (Finucane, Jing, and Kress-Gazit 2010). The back-end of LTLMoP is based on the GR(1) synthesis algorithm presented in (Piterman, Pnueli, and Saar 2006). We note also that LTLMoP contains many front-end features, such as natural language processing, that are orthogonal or even complementary to the policy synthesis work we present in this paper.

The comparison results for the liveness task benchmark are shown in Fig. 5. To evaluate the gains from the generalization step (i.e., using domain-specific heuristics to generalize verification failures), we also test our policy synthesizer with generalization turned off (green-triangle plot).

As we can see from Fig. 5, for small size problems, the

GR(1) synthesizer performs slightly better while for problems with size greater than 16, our method performs better. Therefore, our approach provides better scalability in these tests. Moreover, the results demonstrate that generalization gives order-of-magnitude performance improvements. Without generalization, our method scales worse than the GR(1) synthesizer and cannot solve the problem with *Food-Prep Region* larger than 12 in 30 minutes.

To demonstrate the scalability of our method with *quantitative constraints*, we also run the liveness task benchmark with one additional energy constraint: the robot's energy state $\mathrm{EnergyState}$ should be always greater than a certain threshold e_{min}, i.e., $\mathrm{EnergyState} > e_{min}$. The results are shown in Fig. 6. The orange-circle plot shows the performance of our approach with this energy constraint. For comparison, we also include the results for the liveness task benchmark without the energy constraint (blue-circle plot). With the additional energy constraint, our policy synthesizer still scales well and the performance is roughly one-time slower than the performance without the energy constraint.

Conclusion

We have presented a symbolic approach for task and motion policy synthesis where the robot must accomplish tasks in environments with uncontrollable agents. Our results show that, compared to an existing robotic synthesis tool – the GR(1) back-end of LTLMoP – our approach scales better in the tested benchmark. We also show that our approach can handle quantitative constraints such as energy limits efficiently.

There are multiple avenues to extend this work. Currently, we assume the robot has perfect sensing with no observation uncertainty. An important ongoing question is how to extend the current approach to handle uncertainty in the environment, such as sensor noise or other probabilistic beliefs. In addition, we have so far only explored generalization heuristics for pursuer-evader games, an important though specific class of problems. Additional generalization heuristics for broader domains is another promising future direction.

Acknowledgments This work was supported in part by NSF CCF 1139011, NSF CCF 1514372, NSF CCF 1162076 and NSF IIS 1317849.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *FMCAD*, 1–8.

Alur, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *J. ACM* 49(5):672–713.

Beyene, T.; Chaudhuri, S.; Popeea, C.; and Rybalchenko, A. 2014. A constraint-based approach to solving games on infinite graphs. In *POPL*, 221–233.

Bhatia, A.; Maly, M.; Kavraki, L.; and Vardi, M. 2011. Motion planning with complex goals. *Robotics & Automation Magazine, IEEE* 18(3):55–64.

Cirillo, M.; Pecora, F.; Andreasson, H.; Uras, T.; and Koenig, S. 2014. Integrated motion planning and coordination for industrial vehicles. In *ICAPS*.

Dantam, N., and Stilman, M. 2013. The motion grammar: Analysis of a linguistic method for robot control. *Robotics, IEEE Transactions on* 29(3):704–718.

de Alfaro, L., and Henzinger, T. 2000. Concurrent omega-regular games. In *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*, 141–154.

De Moura, L., and Bjørner, N. 2008. Z3: An efficient smt solver. In *TACAS*, 337–340.

Decastro, J. A., and Kress-Gazit, H. 2015. Synthesis of nonlinear continuous controllers for verifiably correct high-level, reactive behaviors. *Int. J. Rob. Res.* 34(3):378–394.

Dornhege, C.; Gissler, M.; Teschner, M.; and Nebel, B. 2009. Integrating symbolic and geometric planning for mobile manipulation. In *Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on*, 1–6.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Erdem, E.; Haspalamutgil, K.; Palaz, C.; Patoglu, V.; and Uras, T. 2011. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *ICRA*, 4575–4581.

Finucane, C.; Jing, G.; and Kress-Gazit, H. 2010. Ltlmop: Experimenting with language, temporal logic and robot control. In *IROS*, 1988–1993.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *ICRA*, 1470–1477.

Kavraki, L. E.; Svestka, P.; Latombe, J. C.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Nedunuri, S.; Prabhu, S.; Moll, M.; Chaudhuri, S.; and Kavraki, L. 2014a. Smt-based synthesis of integrated task and motion plans from plan outlines. In *ICRA*, 655–662.

Nedunuri, S.; Wang, Y.; Prabhu, S.; Moll, M.; Chaudhuri, S.; and Kavraki, L. E. 2014b. Synthesis of integrated task and motion plans from plan outline using smt solvers. Technical report, Rice University.

Piterman, N.; Pnueli, A.; and Saar, Y. 2006. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*, 364–380. Springer.

Solar-Lezama, A.; Tancau, L.; Bodik, R.; Seshia, S.; and Saraswat, V. 2006. Combinatorial sketching for finite programs. In *ASPLOS*, 404–415.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In *ICRA*, 639–646.

Udupa, A.; Raghavan, A.; Deshmukh, J. V.; Mador-Haim, S.; Martin, M. M.; and Alur, R. 2013. Transit: Specifying protocols with concolic snippets. In *PLDI*, 287–296.

Ulusoy, A.; Marrazzo, M.; and Belta, C. 2013. Receding horizon control in dynamic environments from temporal logic specifications. In *Robotics: Science and Systems*.

Wolfe, J.; Marthi, B.; and Russell, S. J. 2010. Combined task and motion planning for mobile manipulation. In *ICAPS*.

Wongpiromsarn, T.; Topcu, U.; and Murray, R. M. 2010. Receding horizon control for temporal logic specifications. In *HSCC*, 101–110.