22nd International Conference
on Automated Planning and Scheduling
June 25-29, 2012, Atibaia – Sao Paulo – Brazil

TAMPRA 2012

Proceedings of the Workshop on
Combining Task and Motion Planning
for Real-World Applications

Edited by
Marcello Cirillo, Brian Gerkey,
Federico Pecora, Mike Stilman

Marcello Cirillo
AASS Research Centre, Örebro University
Örebro, Sweden
marcello.cirillo@oru.se

Brian Gerkey
Willow Garage
Menlo Park (CA), USA
gerkey@willowgarage.com

Federico Pecora
AASS Research Centre, Örebro University
Örebro, Sweden
federico.pecora@oru.se

Mike Stilman
Georgia Institute of Technology
Atlanta (GA), USA
mstilman@cc.gatech.edu

TAMPRA'12
Proceedings of the 2012 ICAPS Workshop
on Combining Task and Motion Planning for Real-World Applications
June 26, 2012
Atibaia, São Paulo, Brazil

Committee

Organizing Committee:

Marcello Cirillo, AASS, Örebro University
Brian Gerkey, Willow Garage
Federico Pecora, AASS, Örebro University
Mike Stilman, Georgia Institute of Technology

Program Committee:

Bhaskara Marthi
Tsz-Chiu Au
Wolfram Burgard
Sachin Chitta
Esra Erdem
Susana Fernández Arregui
Julien Guitton
Kaijen Hsiao
Sven Koenig
Héctor Muñoz Ávila
Volkan Patoğlu
Erion Plaku
Mihail Pivtoraiko
Alessandro Saffiotti
David Sislak
Kartik Talamadupula
Emrah Akın Şişbot
Adi Botea
Marc Cavazza
Minh Do
Jean-Loup Farges
Alberto Finzi
Geoffrey Hollinger
Lars Karlsson
Carlos Linares López
Jeff Orkin
David Pizzi
Roland Phillipsen
Ananth Ranganathan
Sanem Sariel Talay
David E. Smith

Foreword

A longstanding aim of research in AI has been to employ discrete task planning capabilities in the service of mobile robots. Since the early days of Shakey, the planning community has worked to build algorithms that would allow a robot to reason about its own actions before (or while) carrying them out physically. Recent advancements in artificial vision, manipulation, motion planning and control have done a great deal to bring this vision closer to fruition. However, several issues remain open in the combination of task and motion planning. Research has not yet produced the algorithmic and theoretical results necessary to integrate techniques for automated decision making at the task and motion planning levels. Moreover, while autonomous mobile robots have become a commercial reality, existing products often rely on pre-calculated motions and/or static, pre-computed plans.

This workshop focuses on two important challenges: the complex requirements, such as dynamic environments and real-time, continuous operation, posed by real-world applications; and the issue of combining the different search and inference procedures underlying the two forms of planning. TAMPRA follows in the tradition of two previous ICAPS workshops: Combining Action and Motion Planning, at ICAPS 2010, and Bridging The Gap Between Task And Motion Planning, at ICAPS 2009. The accepted papers address three areas of interest: combining motion and task planning for high DoF robots, the use and learning of symbolic knowledge in task/motion planning, and the use of combined task and motion planning strategies in industrially relevant application scenarios.

Marcello Cirillo
Brian Gerkey
Federico Pecora
Mike Stilman

WORKSHOP PROGRAM

June 26

Session 1: Combining motion and task planning for high DOF robots

Session 2: Symbolic knowledge in task and motion planning

Session 3: Application scenarios

Simulating Human-like Motion in Constrained Dynamic Environments

Ming C. Lin Jia Pan Chonhyon Park Dinesh Manocha
University of North Carolina at Chapel Hill

Abstract

In this position paper, we examine the algorithmic and computational challenges in real-time simulation of human-like motion in highly constrained dynamic environments. We briefly survey some related work and our recent progress in generating motions for high-DOF humanoid robots in constrained environments with multiple moving obstacles, as well as real-time replanning techniques. We discuss the benefits of sample-based planning, data-driven approaches, optimization-based techniques, and GPU-computing.

1 Introduction

The problem of modeling and simulating human-like motion arises in different applications, including humanoid robotics, computer animation, virtual prototyping, and exploration of sensor-motor basis for neuroscience. This is a challenging problem due to both combinatorial and behavioral complexities. For example, the entire human body consists of over 600 muscles and over 200 bones and there are no known accurate and efficient algorithms to simulate their motion. Even the simplest human-like models that represent the skeleton as an articulated figure need at least $30 - 40$ joints to model different motions such as navigation, sitting, walking, running, object manipulation, etc. The high dimensionality of the configuration space of the articulated model makes it difficult to efficiently compute the motion. In addition to collision-free and kinematic constraints, we also need to ensure that the resulting trajectory satisfies the posture and dynamic constraints and looks realistic.

Most of the research in computer animation is based on motion capture, which tends to generate the most realistic human-like motion. Many techniques have been proposed to edit and modify or retarget the motion capture (mocap) data. However, it is difficult to capture the motion in constrained environments with multiple obstacles due to occlusion problems. Furthermore, it is hard to reuse or playback the motion in a virtual environment, which is different from the original environment in which the motion is captured (Shapiro, Kallmann, and Faloutsos 2007; Pettre, Kallmann, and Lin 2008; Kallmann et al. 2003; Kalasiak and van de Panne 2001).

In this position paper, we primarily focus on generating human-like motion in constrained environments with many obstacles, cluttered areas, or tight spaces. Some of the challenges we need to address include:

1. Real-time synthesis of high-degree-of-freedom human motion;
2. Planning and scheduling in constrained environments with obstacles; and
3. Generating smooth and realistic motion trajectories.

For (1) and (2), we propose a novel approach of combining motion captured data to exploit the realistic nature of the recorded motion and hierarchical motion planner. For (3), we suggest a fast incremental trajectory optimization technique. The rest of the paper is organized as follows. We first briefly survey prior work, followed by an overview of our recent work on planning in constrained environment and real-time replanning. We conclude by discussing some open research issues in this area.

2 Related Work

In this section, we give a brief overview of prior work on motion generation techniques in character animation and robot motion planning. For the later, we focus on motion planning in dynamic environments, real-time replanning, optimization-based planning, and parallel algorithms for motion planning.

Character Animation

There is extensive literature on motion generation in computer animation. At a broad level, prior methods can be classified into kinematic and dynamic methods. The basic kinematic methods use operators such as re-sequencing and interpolation to recombine the example motions into new motions, as is done in motion blending and motion graphs (Kovar, Gleicher, and Pighin 2002). Some recent variants (Safonova and Hodgins 2007) use optimization methods to satisfy the constraints. These methods can create natural long clips with a variety of behaviors, but their results are restricted within the linear space spanned by the example motions. Moreover, they need to be combined with global planning or collision avoidance schemes in order to handle

constrained environments. Shapiro et al. (Shapiro, Kallmann, and Faloutsos 2007) described an elegant approach to combine mocap data with motion planning.

Recently, dynamics-based methods have become popular in the animation community, e.g. (Abe and Popović 2006). These approaches use a control strategy (e.g. PD control) to actuate a dynamics model (e.g. Newton-Euler equation). In practice, they are primarily used for interactive response and may not work well for collision-free motion computation in constrained environments.

Motion Planning

Sample-based approaches have been successfully applied to human-like robots to plan various tasks and motions. These include efficient planning algorithms for reaching and manipulation that combine motion planning and inverse kinematics (Diankov et al. 2008; Drumwright and Ng-Thow-Hing 2006) or computing the whole body motion (Hauser et al. 2008). Moreover, motion strategies for human-like robots such as walking, sitting or jumping can also be computed by walking pattern generators (Huang et al. 2001; Kajita et al. 2003). In order to plan collision-free and dynamically stable motions, many earlier approaches used a two-stage decoupled framework (Harada et al. 2007; Kuffner et al. 2002; Yoshida et al. 2005). Task-based controllers have also been used to plan and control the whole-body motion (Gienger, Goerick, and Körner 2008; Sentis and Khatib 2006).

To generate natural-looking motion, many authors have proposed a two-stage framework. The planner first computes the motion taking into account a few or partial DOFs of the human model, e.g., cylindrical lower-body (Pettré, Laumond, and Siméon 2003), manipulator (Yamane, Kuffner, and Hodgins 2004), footsteps (Choi, Lee, and Shin 2003), etc. In the second stage, some motion data (e.g. mocap data) is retargetted by using the planned trajectory as constraints, e.g., a 2D trajectory (Pettré, Laumond, and Siméon 2003) or inverse-kinematics (Yamane, Kuffner, and Hodgins 2004). These methods typically work well in terms of generating regular motion (e.g. locomotion) in somewhat open environments without many obstacles.

Planning in Dynamic Environment Most of the approaches for motion planning in dynamic environment assume the trajectories of moving objects are known a priori. Some of them model dynamic obstacles as static obstacles with a short horizon of high cost around the beginning of the trajectory (Likhachev and Ferguson 2009). Another common approach is to use velocity obstacles which determine the velocities that avoid collisions with dynamic obstacles (Fiorini and Shiller 1998; Wilkie, van den Berg, and Manocha 2009). However, these methods cannot give any guarantees on the optimality of the resulting trajectory.

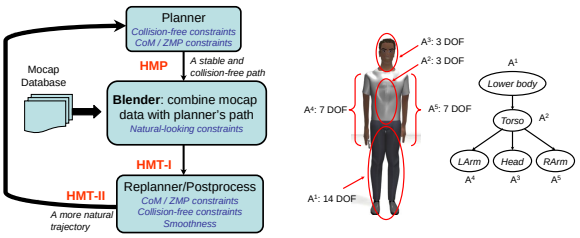
Some of the planning methods handle the continuous state space directly, e.g. RRT variants have been proposed for planning in dynamic environments (Petti and Fraichard 2005). For discrete state spaces, efficient planning algorithms for dynamic environment include variants of A* algorithm, which are based on classic heuristic search (Phillips and Likhachev 2011b; 2011a) and roadmap based algorithms (van den Berg and Overmars 2005).

Most planning algorithms for dynamic environments (van den Berg and Overmars 2005; Phillips and Likhachev 2011b) assume that the inertial constraints, such as acceleration and torque limit, are negligible for the robot. Such assumption implies that the robot can stop and accelerate instantaneously, which is not feasible for real robot.

Real-time Replanning Since path planning can be computationally expensive, planning before execution can lead to long delays during robot's movement. To handle such problem, real-time replanning interleaves planning with execution so that the robot may decide to compute only partial or sub-optimal plans in order to avoid delays in the movement. Real-time replanning methods differ in many aspects. One difference is the underlying planner. Sample-based motion planning algorithm such as RRT have been applied to real-time replanning for dynamic continuous systems (Hsu et al. 2002; Hauser 2011; Petti and Fraichard 2005). These methods can handle high-dimensional configuration spaces but usually cannot generate optimal solutions. A* variants such as D* (Koenig, Tovey, and Smirnov 2003) and anytime A* (Likhachev et al. 2005) can efficiently perform replanning on discrete state spaces and provide optimal guarantees, but are mostly limited to low dimensional spaces. Most replanning algorithms use fixed time steps, while interleaving between planning and execution (Petti and Fraichard 2005). Some recent work (Hauser 2011) computes the interleaving timing step in an adaptive manner to balance between safety, responsiveness, and completeness of the overall system.

Optimization-based Planning Algorithms The most widely-used method of path optimization is the so-called 'shortcut' heuristic, which picks pairs of configurations along a collision-free path and invokes a local planner to attempt to replace the intervening sub-path with a shorter one (Chen and Hwang 1998; Pan, Zhang, and Manocha 2011). Another approach used in elastic bands or elastic strips planning involves modeling paths as mass-spring systems and gradient based methods are used to find a minimum-energy path (Brock and Khatib 2002; Quinlan and Khatib 1993). All these methods require a collision-free path as an initial value to the optimization algorithm. Some recent approaches, such as (Ratliff et al. 2009; Kalakrishnan et al. 2011; Dragan, Ratliff, and Srinivasa 2011) directly encode the collision-free constraints and can work as a motion planner to transform a naive initial guess into a trajectory suitable for robot execution.

Parallel Planning Algorithms Parallelized planning algorithms can improve the responsiveness, safety and completeness of the robot system. These include distributed or multi-core algorithms for sampling-based planner (Amato and Dale 1999; Plaku and Kavraki 2005; Devaurs, Simeon, and Cortes 2011). Many-core GPUs are also used for accelerating sampling-based algorithms (Pan, Lauterbach, and Manocha 2010) and search-based planning algorithms (Kider et al. 2010).



(a) (b)

Figure 1: (a) An overview of our hybrid approach, which can combine the motion computed by planner and the motion from mocap databases to generate a collision-free, dynamic and natural human motion. (b) Our 5-component decomposition scheme for a 38-DOF human-like model. We compute a trajectory for each component in an incremental manner.

3 Hybrid Planner and Trajectory Optimization for Real-time Planning

In this section, we present a brief overview of our recent approaches. For more detail, please refer to *http://gamma.cs.unc.edu/DHM/* and *http://gamma.cs.unc.edu/ITOMP/*.

Hybrid Planner

We present an original hybrid approach that combines motion planning algorithms for high-DOF articulated figures with motion capture data to generate collision-free motion that satisfies both kinematic and dynamic constraints. Our approach performs whole-body planning by coordinating the motion of different parts of the body and later refines the trajectory with mocap data. The two novel components of our work include:

- **Decomposition planner:** In order to deal with high-DOF articulated figures, we use a hierarchical decomposition of the model and perform constrained coordination to generate a collision-free trajectory and maintain static/dynamic balancing constraints. The resulting planner computes the path for low-DOF components in an incremental manner and uses *path constraints* and *path perturbation* to generate a trajectory that satisfies kinematic and dynamic constraints.
- **Trajectory Refinement:** In order to overcome the random nature of sample-based planners and generate realistic motion, we refine the motion computed by our planner with mocap data to compute smooth paths. The resulting motion blending algorithm analyzes the motion and automatically builds a mapping between the path computed by the planner and the mocap data. We ensure that the resulting path still satisfies various constraints.

An overall pipeline of our hybrid planner algorithm is given in Fig 1(a). We do not make any assumptions about the environment or the obstacles in the scene. We assume that a human-like model is represented by an articulated model with serial and parallel joints and there is no limit on the number of DOFs.

We demonstrate the results on generating human-like motion for a 38-DOF articulated model using the CMU mocap database. Our system can handle very cluttered environments to generate object grasping, bending, walking and lifting motions.

Real-time Incremental Trajectory Optimization

Most moving objects motions are not precisely predictable or can only be approximated over a small or local time interval. Such uncertainty about moving objects also makes it hard to plan a safe trajectory for the robot over a long horizon. Another challenge in terms of planning in dynamic environments is that the planning algorithm must be responsive to unpredictable situations, which requires real-time planning capability in terms of computing or updating the trajectory. There exist some recent work trying to accelerate high-DOF planning algorithms, such as GPU parallelism (Pan, Lauterbach, and Manocha 2010) or distributed systems (Devaurs, Simeon, and Cortes 2011). Such a fast planner can definitely improve the responsiveness, but it may not provide an adequate solution for all situations. The reason is that there exist some difficult scenarios, e.g., narrow passages (LaValle 2006), which are hard for any planner in terms of real-time computation. In these cases, planning before the task execution can lead to delays in the movement and decrease the safety of robot movement. One possible solution is by interleaving planning with execution, and the overall algorithm ends up computing partial or sub-optimal plans (Hauser 2011).

To overcome the above challenges, we introduce a novel optimization-based algorithm for motion planning in dynamic environments. Our approach uses a stochastic trajectory optimization framework to avoid collision and satisfies smoothness and dynamics constraints. Our algorithm does not require a priori knowledge about global motion or trajectory of each dynamic obstacle. We compute a conservative local bound on the position or trajectory of each obstacle over a short time and use it to compute a collision-free trajectory in an incremental manner. In order to balance between the planning horizon and the responsiveness to dynamic obstacles, we interleave planning and execution of the robot in an adaptive manner. Moreover, we parallelize the optimization scheme on multi-core processors to improve its runtime performance, convergence and responsiveness.

4 Discussion and Summary

We have briefly presented an algorithm that combines a high-DOF motion planning algorithm with mocap data to generate plausible human motion and satisfy geometric, kinematic and dynamic constraints, as well as an incremental trajectory optimization method. Some of the remaining challenges include (a) satisfying all constraints in the presence of narrow passages in the free space efficiently; (b) obtaining sufficient motion samples to generate realistic motions; (c) developing better searching strategy to find the best-match mocap clips; (d) incorporating non-holonomic and other constraints (Pettre, Kallmann, and Lin 2008) in planning and trajectory optimization; (e) designing high-level, semantic-based planner and scheduler to perform complex tasks.

References

Abe, Y., and Popović, J. 2006. Interactive animation of dynamic manipulation. In *Symposium on Computer Animation*, 195–204.

Amato, N., and Dale, L. 1999. Probabilistic roadmap methods are embarrassingly parallel. In *Proceedings of IEEE International Conference on Robotics and Automation*, 688–694 vol.1.

Brock, O., and Khatib, O. 2002. Elastic strips: A framework for motion generation in human environments. *International Journal of Robotics Research* 21(12):1031–1052.

Chen, P., and Hwang, Y. 1998. Sandros: a dynamic graph search algorithm for motion planning. *IEEE Transactions on Robotics and Automation* 14(3):390–403.

Choi, M. G.; Lee, J.; and Shin, S. Y. 2003. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph.* 22(2):182–203.

Demirel, H., and Duffy, V. 2007. Applications of digital human modeling in industry. In *Digital Human Modeling, Lecture Notes in Computer Science*, volume 4561. Springer Berlin / Heidelberg. 824–832.

Devaurs, D.; Simeon, T.; and Cortes, J. 2011. Parallelizing rrt on distributed-memory architectures. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2261 –2266.

Diankov, R.; Ratliff, N.; Ferguson, D.; Srinivasa, S.; and Kuffner, J. 2008. Bispace planning: Concurrent multi-space exploration. In *Robotics: Science and Systems*.

Dragan, A.; Ratliff, N.; and Srinivasa, S. 2011. Manipulation planning with goal sets using constrained trajectory optimization. In *Proceedings of IEEE International Conference on Robotics and Automation*, 4582–4588.

Drumwright, E., and Ng-Thow-Hing, V. 2006. Toward interactive reaching in static environments for humanoid robots. In *IEEE/RSJ International Conference On Intelligent Robots and Systems (IROS)*, 846–851.

Fiorini, P., and Shiller, Z. 1998. Motion planning in dynamic environments using velocity obstacles. *International Journal of Robotics Research* 17(7):760–772.

Gienger, M.; Goerick, C.; and Körner, E. 2008. Whole body motion planning - elements for intelligent systems design. *Zeitschrift Künstliche Intelligenz* 4:10–15.

Harada, K.; Hattori, S.; Hirukawa, H.; Morisawa, M.; Kajita, S.; and Yoshida, E. 2007. Motion planning for walking pattern generation of humanoid. In *Proceedings of International Conference on Robotics and Automation*, 4227–4233.

Hauser, K.; Bretl, T.; J.C. Latombe, K. H.; and Wilcox, B. 2008. Motion planning for legged robots on varied terrain. *The International Journal of Robotics Research* 27(11-12):1325–1349.

Hauser, K. 2011. On responsiveness, safety, and completeness in real-time motion planning. *Autonomous Robots* to appear.

Hsu, D.; Kindel, R.; Latombe, J.-C.; and Rock, S. 2002. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research* 21(3):233–255.

Huang, Q.; Yokoi, K.; Kajita, S.; Kaneko, K.; Arai, H.; Koyachi, N.; and Tanie, K. 2001. Planning walking patterns for a biped robot. *IEEE Transactions on Robotics and Automation* 17:280–289.

Kajita, S.; Kanehiro, F.; Kaneko, K.; Fujiwara, K.; Harada, K.; Yokoi, K.; and Hirukawa, H. 2003. Biped walking pattern generation by using preview control of zero-moment point. *Proceedings of International Conference on Robotics and Automation* 1620–1626.

Kalakrishnan, M.; Chitta, S.; Theodorou, E.; Pastor, P.; and Schaal, S. 2011. STOMP: Stochastic trajectory optimization for motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 4569–4574.

Kalasiak, M., and van de Panne, M. 2001. A grasp-based motion planning algorithm for character animation. *The Journal of Visualization and Computer Animation* 12(3):117–129.

Kallmann, M.; Aubel, A.; Abaci, T.; and Thalmann, D. 2003. Planning collision-free reaching motions for interactive object manipulation and grasping. *Computer graphics Forum (Proceedings of Eurographics'03)* 22(3):313–322.

Kider, J.; Henderson, M.; Likhachev, M.; and Safonova, A. 2010. High-dimensional planning on the gpu. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2515–2522.

Koenig, S.; Tovey, C.; and Smirnov, Y. 2003. Performance bounds for planning in unknown terrain. *Artificial Intelligence* 147(1-2):253–279.

Kovar, L.; Gleicher, M.; and Pighin, F. 2002. Motion graphs. *ACM Trans. Graph.* 21(3):473–482.

Kuffner, J.; Kagami, S.; Nishiwaki, K.; Inaba, M.; and Inoue, H. 2002. Dynamically-stable motion planning for humanoid robots. *Autonomous Robots* 12(1):105–118.

Laumond, J.-P.; Ferre, E.; Arechavaleta, G.; and Esteves, C. 2005. Mechanical part assembly planning with virtual mannequins. In *International Symposium on Assembly and Task Planning*, 132–137.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge University Press.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically feasible maneuvers for autonomous vehicles. *International Journal of Robotics Research* 28(8):933–945.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Pan, J.; Lauterbach, C.; and Manocha, D. 2010. g-Planner: Real-time motion planning and global navigation using gpus. In *Proceedings of AAAI Conference on Artificial Intelligence*.

Pan, J.; Zhang, L.; and Manocha, D. 2011. Collision-free and curvature-continuous path smoothing in cluttered environments. In *Proceedings of Robotics: Science and Systems*.

Petti, S., and Fraichard, T. 2005. Safe motion planning in dynamic environments. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2210–2215.

Pettre, J.; Kallmann, M.; and Lin, M. C. 2008. *Motion Planning and Autonomy for Virtual Humans*. ACM SIGGRAPH Course Notes.

Pettré, J.; Laumond, J.-P.; and Siméon, T. 2003. A 2-stages locomotion planner for digital actors. In *Symposium on Computer Animation*, 258–264.

Phillips, M., and Likhachev, M. 2011a. Planning in domains with cost function dependent actions. In *Proceedings of AAAI Conference on Artificial Intelligence*.

Phillips, M., and Likhachev, M. 2011b. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of IEEE International Conference on Robotics and Automation*, 5628–5635.

Plaku, E., and Kavraki, L. 2005. Distributed sampling-based roadmap of trees for large-scale motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3868–3873.

Quinlan, S., and Khatib, O. 1993. Elastic bands: connecting path planning and control. In *Proceedings of IEEE International Conference on Robotics and Automation*, 802–807 vol.2.

Ratliff, N.; Zucker, M.; Bagnell, J. A. D.; and Srinivasa, S. 2009. CHOMP: Gradient optimization techniques for efficient motion planning. In *Proceedings of International Conference on Robotics and Automation*, 489–494.

Safonova, A., and Hodgins, J. K. 2007. Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph.* 26(3):106.

Sentis, L., and Khatib, O. 2006. Whole-body control framework for humanoids operating in human environments. *Proceedings of International Conference on Robotics and Automation* 2641–2648.

Shapiro, A.; Kallmann, M.; and Faloutsos, P. 2007. Interactive motion correction and object manipulation. In *Symposium on Interactive 3D Graphics*, 137–144.

van den Berg, J., and Overmars, M. 2005. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics* 21(5):885–897.

Wilkie, D.; van den Berg, J. P.; and Manocha, D. 2009. Generalized velocity obstacles. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5573–5578.

Yamane, K.; Kuffner, J. J.; and Hodgins, J. K. 2004. Synthesizing animations of human manipulation tasks. *ACM Trans. Graph.* 23(3):532–539.

Yoshida, E.; Belousov, I.; Esteves, C.; and Laumond, J.-P. 2005. Humanoid motion planning for dynamic tasks. *International Conference on Humanoid Robots* 1–6.

Constraints on Intervals for Reducing the Search Space of Geometric Configurations

Fabien Lagriffoul, Lars Karlsson, Alessandro Saffiotti

Abstract

The combination of task and motion planning presents us with a new problem that we call *geometric backtracking*. This problem arises from the fact a single symbolic state or action can be geometrically instantiated in infinitely many ways. When an action cannot be geometrically validated, we may need to backtrack in the space of geometric configurations, which greatly increases the complexity of the whole planning process. In this paper, we address this problem using *intervals* to represent geometric configurations, and constraint propagation techniques to shrink these intervals according to the geometric constraints of the problem. After propagation, either (i) the intervals are shrunk, thus reducing the search space in which geometric backtracking may occur, or (ii) the constraints are inconsistent, indicating the non-feasibility of the sequence of actions without further effort. We illustrate our approach on scenarios in which a two-arm robot manipulates a set of objects, and report experiments that show how the search space is reduced.

Figure 1: The DLR humanoid two-arm system JUSTIN (courtesy DLR)

Introduction

Both task and motion planning have been studied for decades (Nau, Ghallab, and Traverso 2004; LaValle 2006), and efficient algorithms have been developed. However, combining them together is a challenge, because motion planning, which is computationally expensive, has to be *interleaved* with task planning (which is itself a hard problem). We are interested in combining task and motion planning in general, but in this paper, we focus on manipulation of objects by a humanoid robot, Justin (Ott and al. 2006) (Fig. 1). The tasks considered are simple ones, for instance sorting objects according to their type, or stacking cups. In this kind of problems, task planning is not complicated because there are few causal relations between actions. Motion planning is not difficult either, because the workspace of the robot is not very cluttered, and we use predefined grasps to avoid doing grasp planning. Despite these favourable conditions, some problems turn out to be intractable because of *geometric backtracking*.

During task planning, geometric configurations, which are the counterparts of symbolic states, are maintained. When the preconditions of a symbolic action are validated, the geometric configuration associated to the current state is used in order to assess the *geometric* applicability of the action. Now, let us describe in detail the geometric backtracking problem through an example, and show how it impairs the whole planning process. We consider a stacking task (Fig. 2). The task consists in stacking four cups at given location (the square area on the table). Symbolically, the domain is simple: four objects, one location, and four possible actions (pick and place, with left or right arm). Looking carefully at Fig. 2, one can see that the right arm of the robot has almost reached full extension. Stacking the first three cups is possible, but placing the last cup on top of the pile is not possible because the kinematic constraints of the robot do not allow for it.

The last action is not feasible because the cup at the bottom of the pile was placed at a wrong position. If the first cup had been placed closer to the robot, the task could have been completed. Hence, the symbolic plan is actually feasible, but the geometric instance chosen for the first action does not allow the planner to complete the sequence. If the planner aborted the search at this point, it would be *incomplete*, because a solution exists but is not reached. In order to remain complete (up to some spatial resolution), the planner must try alternative geometric instances until a solution is found, or reject this last action after exhaustive search. We

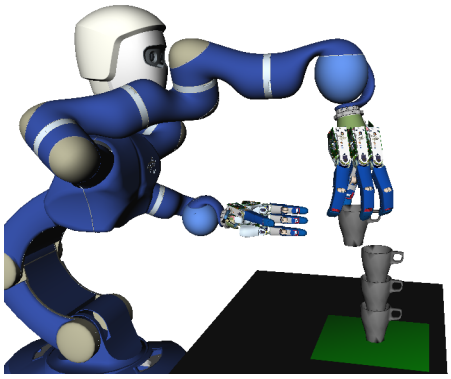


Figure 2: Stacking the last cup is not possible due to kinematic constraints

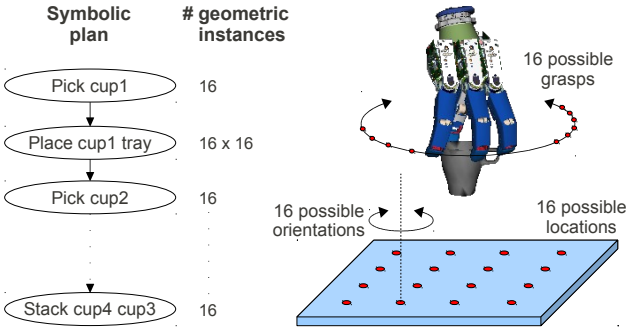


Figure 3: An example of discretization of the space of geometric configurations.

call this process *geometric backtracking*.

Geometric backtracking is a problem for two reasons. First, the size of the space of geometric configurations is infinite, and remains very large even with a gross discretization. Fig. 3 describes a scheme for discretizing the space of geometric configurations:

- the **pick** action can be done using one of the 16 precomputed grasp positions[1];
- the **stack** action can be performed in 16 different ways (16 possible orientations for the cup);
- the **place** action can be achieved in 256 ways (16 locations $\times$16 orientations).

For the "cup stacking" example (Fig. 2), exhaustive search is infeasible ($16 \times 256 \times (16 \times 16)^3 = 6.9 \times 10^{10}$ possibilities). But even on a short sequence composed of two actions, such as *pick* and *place*, the problem arises. If a specific final orientation for the object is required, then the success of the *place* action highly depends on the grasp chosen for the *pick* action. Without a proper strategy, the planner may have to try several grasps before finding one which allows it to place the cup with the desired orientation. And for each candidate grasp tried, we need to try all the 256 possible *place* actions to be sure that the sequence can not be executed with this grasp.

This leads us to the second reason for why geometric backtracking is problematic. Checking for feasibility requires to call the motion planner to ensure that the path is collision-free, which takes time (on average 100 ms). This entails that a simple *pick* and *place* sequence of actions may require a dozen seconds to be solved, and more time to be proved infeasible. This is not acceptable at the task planning level, when plenty of these sequences of actions need to be assessed.

In our view, geometric backtracking is one of the main difficulties in combining task and motion planning. In this paper, we propose an approach to tackle this problem, by introducing an intermediate layer between task planning and motion planning: symbolic actions are not directly instantiated into geometric configurations. Instead, a set of constraints is extracted from the symbolic actions and from the geometric model of the robot. These constraints are used to prune out geometric configurations that can never be part of a solution, which reduces the amount of geometric backtracking. More interestingly, when the set of constraints is inconsistent, we know that the sequence of actions is infeasible without backtracking at all.

[1]In this scenario, only top-grasps are used, but there are no restrictions on the number of grasp types in general

Related work

To our knowledge, aSyMov (Cambon, Alami, and Gravot 2009) was the first planner that combines task and motion planning. Motion planning is done by composition of probabilistic roadmaps (PRMs)(Kavraki et al. 1996), while task planning is based on an A*-like search algorithm. It uses a hybrid state representation: a classical symbolic state, together with its geometric counterpart. During search, the algorithm alternates between finding a plan using the current roadmaps, or adding nodes to the roadmaps in order to refine its geometric knowledge of the world. Each state has a list of candidate geometric configurations, some of which are validated when they are known to be reachable from the previous state. The validation procedure tries to back-trace through valid configurations until the initial configuration is reached. If this is not possible, the algorithm may have (in the worst case) to check for all collision-free paths between all the candidate configurations of each successive state. The search then becomes exponential in the number of robots and objects. aSyMov performs well when the problem is constrained at the task level, but is less efficient on pure geometric problems (cf. the forklifts and boxes experiments).

In (Dornhege et al. 2009), an extension of the planning domain description language (PDDL) is proposed. The operators are augmented with a condition-checker and an effect-applicator, which cause calls to external specialized geometric reasoners during task planning. The grasps are predefined. They use numerical fluents to represent transformation matrices, robot configurations, and poses of objects. It is not clear though how predicates defining

continuous placements on surfaces are dealt with, e.g, *On cup table*, but it seems likely that predefined locations are used. Backtracking occurs at the task level within a set of predefined discrete locations, and the question of geometric backtracking does not arise.

In (L. Pack Kaelbling 2010), a hierarchical task network (HTN) approach is proposed in which the complexity of hybrid planning is indirectly tackled by decreasing the search horizon. The subtasks obtained from the task decomposition are executed as soon as primitive actions are reached. This allows them to re-plan in the now: a useful feature in dynamic or uncertain environments. On the other hand, this prevents them from projecting the geometric consequences of actions far into the future. This implies that geometrically hard problems would require physical backtracking, which is precisely what we try to avoid.

In (Alili et al. 2010), a HTN task planner is combined with a three-layer geometric planner which can deal with different types of constraints. This architecture allows the geometric planner to discover new constraints during planning, and use them for intelligent backtracking. For instance, if after several manipulations of objects, an object cannot be placed at a location imposed by the task, a constraint will be generated for this location. Then, geometric backtracking is performed, and objects will be assigned new locations that do not violate this constraint. This is, to our knowledge, the only approach that considers the issue of geometric backtracking, but few details are given about the exact search space they consider for backtracking .

Generating the constraints

Our approach consists in introducing an intermediate step between task planning and motion planning, in which a set of constraints is generated. These constraints are generated from the sequence of symbolic actions currently explored by the task planner, and from the geometric properties of the robot. These constraints express what can or cannot be geometrically achieved. Intuitively, such constraints capture that *"if object2 is picked from its current position with the right arm and placed on top of object3, the maximum achievable clockwise rotation is 65 degrees"*. These constraints can drastically reduce search in the space of geometric configurations, because they will eliminate many configurations resulting from the discretization process that cannot be part of a feasible sequence. For clarity in this paper, we do not describe the whole planning algorithm, but only how a single sequence of actions (provided by the task planner described in (Karlsson et al. 2012)) is handled.

A state s_j refers to the j^{th} symbolic state of the symbolic action sequence, which corresponds to the j^{th} stage of operation. A state s_j is the result of applying the action A_j to the state s_{j-1} (See Fig. 4). Symbolic states can be geometrically instantiated in many ways, in terms of positions and orientations of objects. Similarly, symbolic actions can be instantiated in many positions and orientations of the tool center point (TCP) of the manipulator. The variables used to

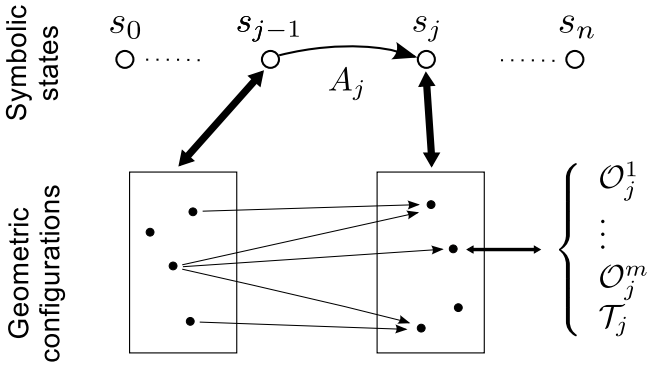


Figure 4: Representation of symbolic states and stages of operation

describe these geometric instances are listed below. These variables are then used to formulate the constraints. The problem is formulated with a set of n variables $\mathcal{V}$, expressed with respect to the world frame:

$$\mathcal{V} = \{x_1, x_2, ..., x_n\}.$$

$\mathcal{V}$ is associated to a *domain* $\mathcal{D}$, which is a set of intervals:

$$\mathcal{D} = \{[\underline{x_1}, \overline{x_1}], [\underline{x_2}, \overline{x_2}], ..., [\underline{x_n}, \overline{x_n}]\},$$

where $\underline{x_i}$ and $\overline{x_i}$ are respectively the lower and upper bounds on the real variable x_i.

Each object o_i is defined by its pose $\mathcal{O}_j^i$, depending on which state s_j is considered:

$$\mathcal{O}_j^i = (x_j^i, y_j^i, z_j^i, \gamma_j^i), \text{ with } x_j^i, y_j^i, z_j^i, \gamma_j^i \in \mathcal{V}$$

where γ is the orientation of the object, i.e., its rotation about z, the vertical axis of the world frame.

At the end of an action A_j the manipulator is represented by the pose of its TCP $\mathcal{T}_j$:

$$\mathcal{T}_j = (x_j, y_j, z_j, \gamma_j), \text{ with } x_j, y_j, z_j, \gamma_j \in \mathcal{V}$$

where γ is the rotation applied to the TCP around the z axis.

Each time an action is applied, we need to create new variables and extend the domain accordingly. For each *pick* action A_k for instance, four new variables have to be created to represent the new position of the TCP, and four intervals are added to the domain:

$$\mathcal{V} = \mathcal{V} + \{x_k, y_k, z_k, \gamma_k\}$$
$$\mathcal{D} = \mathcal{D} + \{[\underline{x_k}, \overline{x_k}], [\underline{y_k}, \overline{y_k}], [\underline{z_k}, \overline{z_k}], [\underline{\gamma_k}, \overline{\gamma_k}]\}$$

The intervals are initially set to arbitrarily large values. Similarly for a *place* action, eight new variables have to be created to represent the new position of the TCP and the new position of the placed object o_i:

$$\mathcal{V} = \mathcal{V} + \{x_k, y_k, z_k, \gamma_k, \ x_k^i, y_k^i, z_k^i, \gamma_k^i\}$$
$$\mathcal{D} = \mathcal{D} + \{[\underline{x_k}, \overline{x_k}], [\underline{y_k}, \overline{y_k}], [\underline{z_k}, \overline{z_k}], [\underline{\gamma_k}, \overline{\gamma_k}]\}$$
$$\mathcal{D} = \mathcal{D} + \{[\underline{x_k^i}, \overline{x_k^i}], [\underline{y_k^i}, \overline{y_k^i}], [\underline{z_k^i}, \overline{z_k^i}], [\underline{\gamma_k^i}, \overline{\gamma_k^i}]\}$$

We can now use these variables to formulate various types of constraints between poses of objects, poses of the TCP, and some constraints imposed by the task.

Location constraints $\mathcal{C}_L$

These constraints are extracted from the symbolic states. For instance, if a symbolic state contains the predicate *On Cup Tray*, one can formulate a set of inequality constraints on the x and y coordinates of the cup, expressing the fact that the cup belongs to the rectangle defined by the tray. We could express constraints for arbitrary regions, by bounding them with a convex polyhedral region. Similar constraints can be extracted from other predicates (i.e. *In, Left, Right, Under, StackedOn,...*). Task constraints also include constraints on desired positions and orientations of objects: the initial and final poses are such constraints.

Transfer constraints $\mathcal{C}_T$

These constraints simply reflect the fact that when an object is manipulated, it undergoes the same translation and rotation than the TCP of the robot. This occurs for instance during a *place* action. For an object o_i manipulated between state s_j and state s_{j+1} we can formulate the constraint:

$$\mathcal{O}^i_{j+1} - \mathcal{O}^i_j = \mathcal{T}_{j+1} - \mathcal{T}_j$$

Grasp constraints $\mathcal{C}_G$

These constraints are formulated each time an object is grasped. They represent the possible relative positions and orientations of the TCP with respect to the object when the object is grasped or released. In our scenario, we use only top-grasps, but such constraints can be formulated for any types of grasp. For a top-grasp, the TCP is situated exactly above the object, which can be formulated as follow for an object o_i, after the *pick* action A_j:

$$\begin{aligned} x_j &= x^i_j \\ y_j &= y^i_j \\ z_j &= z^i_j + H_{top\_grasp}, \end{aligned}$$

where H_{top_grasp} is the distance between the TCP and the object when a top-grasp is performed.

Manipulator constraints $\mathcal{C}_M$

These constraints are the core of our approach. They are very important for manipulation tasks because they express the relationship between the position of the TCP in the workspace and its possible range of rotation. This relationship is non-linear and complex to compute. We approximate it with using linear constraints.

In order to find a linear approximation of these constraints, we compute two maps off-line, using a similar procedure to (Zacharias F. 2007). Essentially, the workspace of the robot is discretized into a 3-dimensional grid, and for each cell, the existence of an inverse kinematic (IK) solution is tested for all possible rotations of a template top-grasp around the vertical axis, by an angle γ. From this data, we can build two maps m_γ and M_γ, which respectively associate the (x, y, z) coordinates of the TCP to a lower and upper bound on γ:

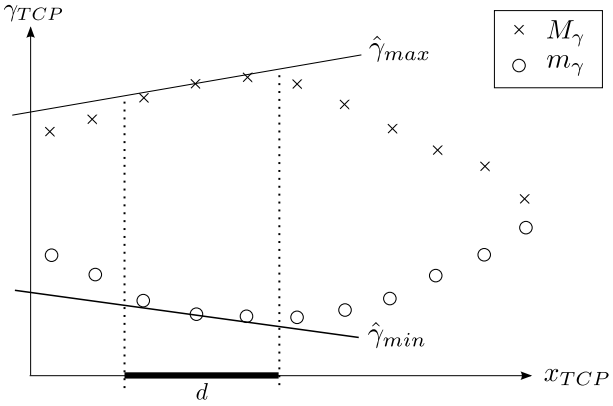


Figure 5: Schematic 2-d view of the 4-dimensional linear outer approximations of m_γ and M_γ on a domain $d \subset \mathcal{D}$

$$\begin{aligned} m_\gamma &: (x, y, z) \mapsto \gamma_{min} \\ M_\gamma &: (x, y, z) \mapsto \gamma_{max} \end{aligned}$$

In order to extract linear constraints from these maps, we define two functions:

$$\begin{aligned} \overline{\mathbf{b}} &= h_{max}(M_\gamma, d) \\ \underline{\mathbf{b}} &= h_{min}(m_\gamma, d), \end{aligned}$$

where d is the domain for the variables representing the position of the TCP only (i.e., a cubic region in space):

$$d = \{\underline{x}_{TCP}, \overline{x}_{TCP}, \underline{y}_{TCP}, \overline{y}_{TCP}, \underline{z}_{TCP}, \overline{z}_{TCP}\}$$

These functions compute the parameters of a lower-bound (resp. upper-bound) hyperplane on the data from m_γ (resp. M_γ) restricted to the domain d. A linear regression is used, followed by a translation towards the lowest (resp. highest) value found in the data. Such hyperplanes are represented by $\hat{\gamma}_{min}$ and $\hat{\gamma}_{max}$ on Fig. 5. It is now possible to formulate two linear constraints on the orientation of the TCP over a domain d during action A_j:

$$\begin{aligned} \underline{\mathbf{b}}\mathbf{x} \;\leq\; \gamma_j \;\leq\; \overline{\mathbf{b}}\mathbf{x} \\ \text{with } \mathbf{x} = (x_j, y_j, z_j, 1)^T, \;\; x_j, y_j, z_j \in d \end{aligned}$$

This constraint is useful in two ways:

- For a given region of space, it provides a lower bound and an upper bound on the possible top-grasps that can be achieved.
- For a given set of top-grasps, it provides a region of space where these grasps can be achieved.

We define the set of constraints $\mathcal{C}$ of our problem:

$$\mathcal{C} = \{\mathcal{C}_L, \mathcal{C}_T, \mathcal{C}_G, \mathcal{C}_M\},$$

and we finally define a geometric configuration c at state s_j as the position of all objects, and the position of the TCP:

$$c = (\mathcal{O}^1_j, \ldots, \mathcal{O}^m_j, \mathcal{T}_j)$$

Constraint Satisfaction on Intervals

The geometric constraints of the problem are formulated with a set of linear inequalities and equalities. The manipulator constraints $\mathcal{C}_M$ have been formulated in terms of lower and upper bounds on the real capacities of the manipulator. Consequently, the set of constraints is *conservative*, meaning that if a solution exists, it must belong to the feasible set defined by the constraints. Conversely, if the set of constraints results in an empty feasible set, it is *guaranteed* that the real problem has no solution. Note, however, that

- we still have to perform search within the feasible set in order to find a solution to the *real* problem;
- we still have to perform motion planning in order to find collision-free paths to connect grasp and release positions.

For these reasons, instead of searching a single solution, we use the constraints to compute a set of tight intervals which contain all the solutions to the problem. Techniques have been developed for solving numerical constraint satisfaction problems on intervals (NCSPI). These intervals can then be used to reduce the search space during geometric backtracking.

Narrowing intervals

Various techniques exist for solving NCSPIs (Davis and Ernest 1987; Lhomme and Rueher 1997; Jaulin 2000). The common approach is to state the problem as a numerical CSP, but each variable is assigned an initial interval, i.e. a lower bound and an upper bound. Then, these intervals are narrowed using branch-and-bound together with local consistency techniques (Lhomme 1993). These techniques allow us to work with non-linear constraints, but their main drawback is that constraint propagation effort is required to decide if a problem is inconsistent. As we will see later, the ability to detect inconsistency fast is essential in our approach.

By restricting ourselves to linear constraints though, it is possible to use a *global* filtering algorithm (adapted from (Lebbah, Rueher, and Michel 2002), see Algorithm 1) which rapidly converges to a global optimum, and detects inconsistency at the first iteration. This algorithm solves several linear programs (LP) in order to find the minimum (resp. maximum) value Z of each variable x_i. Z is then used to update the lower (resp. upper) bound of x_i (lines 9 and 11). The process is repeated until the bounds do not change more than a predefined ϵ value. The result is a domain in which the intervals are narrowed with respect to the constraints.
We have modified the original algorithm at line 6 by adding the function $UpdateManipulatorConstraints(C, \mathcal{D})$ in the main loop. Indeed, after each iteration, the intervals may shrink. If the intervals representing the TCP positions are reduced, it is meaningful to refine the manipulator constraints using the functions $h_{min}(m_\gamma, d)$ and $h_{max}(M_\gamma, d)$, in order to get a tighter linear approximation of the real problem.

Refining the manipulator constraint while filtering the domains is a very effective process. Let us illustrate this with a numerical example for a *pick* action. Initially, the problem consists of four variables representing the position of the object "cup" located in $(0.6, 0.25, 0.1)$ with orientation 0.

$$\mathcal{V} = \{x_{cup}, y_{cup}, z_{cup}, \gamma_{cup}\}$$
$$\mathcal{D} = \{[0.6, 0.6], [0.25, 0.25], [0.1, 0.1], [0, 0]\}$$

The lower bounds are equal to the upper bounds because the values of the variables are determined. The pick action leads us to the creation of 4 new variables for the TCP, which are initially assigned arbitrary large intervals:

$$\mathcal{V} = \{x_{cup}, y_{cup}, z_{cup}, \gamma_{cup}, x_{TCP}, y_{TCP}, z_{TCP}, \gamma_{TCP}\}$$
$$\mathcal{D} = \{[0.6, 0.6], [0.25, 0.25], [0.10, 0.10], [0, 0], [-10, 10], [-10, 10], [-10, 10], [-\pi, \pi]\}$$

The pick action also generates grasp constraints $\mathcal{C}_G$ and manipulator constraints $\mathcal{C}_M$:

$$x_{j+1} = x_j^i$$
$$y_{j+1} = y_j^i \qquad \underline{\mathbf{b}}\mathbf{x} \leq \gamma_j \leq \overline{\mathbf{b}}\mathbf{x}$$
$$z_{j+1} = z_j^i + 0.34$$

After applying the function $FilterDomain$, $\mathcal{D}$ becomes:

$$\mathcal{D} = \{[0.6, 0.6], [0.25, 0.25], [0.10, 0.10], [0, 0], [0.6, 0.6], [0.25, 0.25], [0.44, 0.44], [-0.70, 2.36]\}$$

The grasp constraints have propagated the values of the position of the cup to the position of the TCP. In the second iteration, the bounds on the orientation of the TCP have been updated with the linear approximations of the maps, but since the domain of the TCP is now a single point, these bounds represent the exact possible rotations of the TCP at this point. Now, we know that in order to pick the cup, the orientation of the top-grasp must be chosen between -0.70 and 2.36 radians. This constraint propagation process could for instance solve the stacking problem described in the introduction. It would also give an approximation of a region on the table which is appropriate for placing the first cup.

Algorithm 1: FilterDomain

Function **FilterDomain**$(\mathcal{D}, \mathcal{C})$

input : $\mathcal{D}$: a domain
C: a set of constraints

1 ϵ = minimal domain reduction
2 Construct the Linear Program LP from $\mathcal{C}$
3 $\mathcal{D}' = \mathcal{D}$
4 **repeat**
5 $\quad \mathcal{D} = \mathcal{D}'$
6 $\quad UpdateManipulatorConstraints(C, \mathcal{D})$
7 $\quad$ **forall the** $x_i \in vars(C)$ **do**
8 $\quad\quad$ Solve LP, $Z = minimize(x_i)$
9 $\quad\quad \underline{x_i}' = max(\underline{x_i}, Z)$
10 $\quad\quad$ Solve LP, $Z = maximize(x_i)$
11 $\quad\quad \overline{x_i}' = min(\overline{x_i}, Z)$
12 **until** $(\mathcal{D} - \mathcal{D}') \leq \epsilon$ **or** $\mathcal{D}' = \emptyset$
13 **return** $\mathcal{D}'$

Narrowing intervals during search

In order to find a solution, we use a basic depth-first-search algorithm, endowed with a pruning step (see algorithm 2: $SearchAndFilter$ (SAF)). Geometric instances of configurations are not chosen arbitrarily, but such that the variables representing them ($\mathcal{O}^1, \ldots, \mathcal{O}^m, \mathcal{T}$) belong to their respective intervals. This is the first level of pruning. But after an action has been chosen (e.g., to place the cup at position $(0.7, -0.25, 0.1)$ with $\gamma = \pi/2$), the variables representing this choice are assigned fixed values, so the corresponding intervals can be reduced to single points (i.e., for a variable x_i, $\underline{x_i} = \overline{x_i}$). Then, we can filter the domain *again* in order to propagate this choice to other variables through the constraints. The other intervals will be shrunk accordingly, which will reduce even more the possibilities for further actions. This process is repeated each time an action is chosen, so that intervals are shrunk *as* the search progresses.

Algorithm 2: SearchAndFilter

```
Function SearchAndFilter(c_1, Seq, D)
input : c_1: a geometric configuration
        Seq: a sequence of symbolic actions
        D: a domain
1  if Seq = ⟨⟩ then return c_1
2  Action = Seq.head
3  Rest = Seq.tail
4  foreach A_i ∈ geometricInstanceOf(Action) do
5      c_2 = getSuccesorConf(c_1, A_i)
6      if c_2 ∈ D then
7          D' = assignValues(D, c_2)
8          D' = filterDomain(D')
9          if D' ≠ ∅ then
10             feasible = pathPlanning(c_1, c_2)
11             if feasible then
12                 s = SearchAndFilter(c_2, Rest, D')
13                 if s ≠ false then
14                     return ⟨c_2, s⟩
15 return false
```

The algorithm is initially called with the initial geometric configuration, the sequence of symbolic actions, and the initial domain filtered according to the constraints of the problem. An action A_i is chosen among the possible geometric instances of $Action$ (e.g., 16 for *pick*, 256 for *place*). c_2 is the result of applying A_i to c_1. If this configuration belongs to the domain, we apply the strategy described above, that is assigning the values to the domain and filter it again (lines 7-8). If no inconsistency appears, the motion planning algorithm is called to check if a collision-free path exists to reach c_2. If a path exists, the function is recursively called on c_2 with the remaining actions and the shrunk domain $\mathcal{D}'$, otherwise the next action A_i is tried. If all the actions fail, the function returns $false$ to the calling function via the return statement line 15. If a final configuration is reached (line 1), the solution is incrementally built (line 14) and returned to the main calling function. The result is a list of geometric configurations which can be used to execute the final plan, because it contains the initial and final poses of the TCPs of the robot for each action.

Detecting inconsistency and pruning

One of the main problems of geometric backtracking is when no geometric instantiation of the action sequence exists. This happens often during task planning, because no geometric information is used. For instance, the task planner may try a sequence in which the right arm of the robot grasps an object situated on the left side. In the worst case, for such a sequence, all the space of configurations has to be searched in order to discover that it is infeasible, which may be computationally expensive. The only solution to avoid this is to impose a time limit on the backtracking process. By doing this unfortunately, completeness is lost for cases when the problem *is* feasible.

On the other hand in our approach, inconsistency can be detected *before* entering the backtracking procedure, while we filter the initial domain according to the constraints of the problem. This is more efficient since no search is required. Inconsistency can also be exploited *during* search in order to prune out a whole branch of the search tree. This happens when the problem is initially consistent, and at some point in the search, an action is chosen that makes the problem inconsistent. This will be detected during filtering (line 8-9 in the Algorithm 2). Then, we do not need to search further with this action sequence, and can try another action.

Experimental results

Experimental setup

In Section I, we have identified geometric backtracking as being the main source of complexity in combining task and motion planning. Geometric backtracking occurs while evaluating the feasibility of a sequence of symbolic actions. What a task planner does is essentially to evaluate many of these sequences. Hence, we evaluated our approach by evaluating single sequences of actions. We compare our algorithm SAF to a standard depth-first-search (DFS) procedure, i.e., SAF without the filtering process (lines 6 to 9), and without using the argument $\mathcal{D}'$ at line 12. We compare our algorithm against DFS, because DFS is equivalent to the strategies used in similar work (see Section II), i.e., a non-informed backtracking search.

We use a simulation environment provided by DLR[2] for the robotic platform Justin (Ott and al. 2006). Justin is a humanoid robot with two arms with 7 DoF each. The robot is situated in front of a table, on which are placed 30 $cm \times$ 30 cm trays, and some cups that can be manipulated. The space is discretized with a resolution of 5 cm for the trays and 15 cm for the table, and orientations with an angular value of $\pi/8$. (which means 36 possible positions on trays,

[2]Deutsche Zentrum für Luft-und Raumfahrt

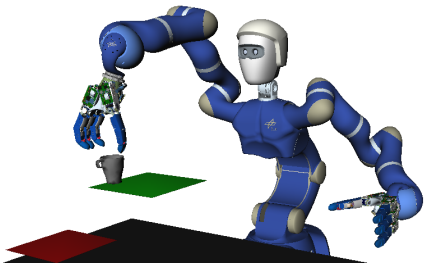


Figure 6: An example of constrained regrasping in Experiment 1

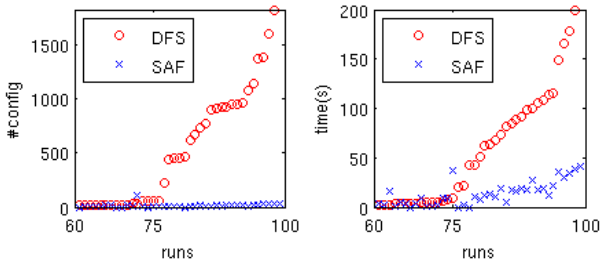


Figure 7: Results for Experiment 1: #config on the left, time on the right.

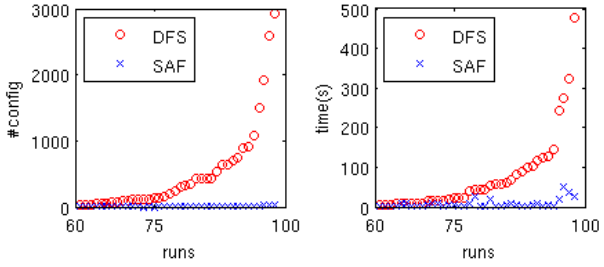


Figure 8: Results for Experiment 2: #config on the left, time on the right. A similar trend is observed.

32 on the table, and 16 possible orientations). We evaluated our approach on two different sequences of actions:

In **Experiment 1** (see Fig. 6), we used one object, a sequence of four actions, and only the right arm:

- *Pick right cup1*
- *Place right cup1 tray1*
- *Pick right cup1*
- *Place cup1 tray2,*

where *tray1* can be randomly situated from 10 cm to 40 cm above the surface of the table. We also imposed a constraint on the final orientation of the cup ($\gamma_1^{(4)} = \pi$).

In **Experiment 2**, we used one object, a sequence of five actions, and both arms:

- *Pick right cup1*
- *Place right cup1 table*
- *MoveAway right-arm*
- *Pick left cup1*
- *Place left cup1 tray1,*

where the cup is initially located on the right side of the table, and the tray on the left side, with random variation. The initial orientation of the cup was randomly chosen, and a constraint was imposed on its final orientation $\gamma_1^{(5)}$, which was also randomly chosen.

For all experiments, we have measured the number of geometric configurations explored (#config), and the search time (time). Both algorithms were run on the same problems, and 100 runs were conducted. For solving the linear program, we use the Gurobi linear solver(gur). For motion planning, we use a bi-directional rapidly exploring random tree (RRT) planner. We only compute a raw trajectory to assess the feasibility of the path. This allows us a faster computation during planning (100 ms on average), and the final smooth trajectory is computed before execution of the plan. The algorithms are implemented in java, and run on a MacBook Pro (Intel Core i7 dual-core 2.66 GHz).

Results

The results for Experiment 1 and 2 are shown on Fig. 7 and Fig. 8 respectively. The horizontal axis represent the runs, sorted by increasing number of configurations explored (resp. time) by the DFS algorithm. Hence, the horizontal axis represents the complexity of the problem measured "ex post facto" by DFS. We do not show the 60 shortest/easiest runs, for which both algorithms have a similar performance (the problem is solved in 1.5 s on average). The difference becomes noticeable for more "complicated" cases, i.e., when geometric backtracking is required.

In Experiment 1, geometric backtracking is necessary when *tray1* is high or ill-placed. The possibilities for regrasping from there are then limited (see Fig. 6), which may cause the last *place* action to fail. Hence, the choice in the intermediate position and orientation of the cup is important to complete the sequence. In Experiment 2, a position for the cup has to be found on the table where both arms can reach it, and where the kinematic constraints of both arms are satisfied during grasping.

In 75% of the cases, these tasks do not require backtracking, and both algorithms perform well. But in 25% of the remaining cases, due to the initial conditions, the time spent by DFS explodes because it arbitrarily selects the configurations to explore, which is often a wrong choice in such "difficult cases". This entails backtracking, and increases exponentially the number of configurations explored, depending on the depth of backtracking.

On the other hand, SAF takes advantage of the constraints to choose a suitable intermediate position, which reduces backtracking. This is clear on the longest runs: the number of configurations visited explodes for DFS (until 3000 in Experiment 2), whereas it stays below 30 for SAF. In terms of time, the trend is similar, since the RRT planner is called

for each configuration explored. In Experiment 1 however, the time for SAF increases slightly more than the number of configurations explored. This is because in Experiment 1, difficult cases are also complicated in terms of cluttering of the scene, because $tray1$ acts as an obstacle. Hence, the RRT planner takes longer time to compute each path.

In cases where the task is not feasible (depending on the initial positions of objects), a similar trend is observed. If the cause for infeasibility arises early, DFS may find it quickly. But if the problem has to do with the last action, or if it depends on a complicated interaction of the constraints, SAF is much more efficient. In extreme cases SAF, may detect infeasibility without backtracking at all, by simply detecting inconsistency in the system of constraints, which takes less than 100 ms on average.

In these experiments, a comparison to a state-of-the-art planner (i.e., asymov) is missing. Indeed, PRM techniques have proved good performance on various manipulation problems. However, we hypothesize that in the kind of problems we aim at solving, PRMs would not perform that well, their efficiency is based on re-using path segments which are known to be collision-free. This strategy is no longer efficient when objects can be moved around, and when two manipulators share the same workspace, because the roadmaps have to be substantially updated after each action.

Conclusion

The main contribution of this paper is twofold. First, we have identified *geometric backtracking* as one of the major sources of complexity when combining task and motion planning. While new approaches that combine task and motion planning are being increasingly proposed, to the best of our knowledge the problem of geometric backtracking has not been explicitly identified and addressed until now. The second contribution is a method for dealing with geometric backtracking. The core idea is to extract a set of linear constraints from the symbolic plan and the kinematics of the robot, and to apply constraint satisfaction techniques on intervals in the space of geometric configurations. This narrows the geometric search space and avoids unnecessary calls to the motion planner.

The proposed technique is efficient for geometrically constrained tasks, and tasks in which action dependencies impose to backtrack far back in the sequence. Another advantage of the proposed approach is to quickly detect unfeasible cases, which is an important feature when used in combination with a task planner. In this work, we have used intervals to deal specifically with geometric constraints. Intervals are a compact representation which allows us to reason about space without being affected by the curse of dimensionality caused by the discretization process. Therefore, we believe that intervals are appropriate for bridging the gap between task and motion planning in general.

ACKNOWLEDGMENTS

This work was partially supported by EU FP7 project "Generalizing Robot Manipulation Tasks" (GeRT, contract number 248273). We would like to thank in particular Florian Schmidt from the Robotics and Mechatronics Center of DLR, which developed for us functionalities in Justin's simulation environment which made this work possible.

References

Alili, S.; Pandey, A. K.; Sisbot, E. A.; and Alami, R. 2010. Interleaving symbolic and geometric reasoning for a robotic assistant. In *ICAPS Workshop on Combining Action and Motion Planning*.

Cambon, S.; Alami, R.; and Gravot, F. 2009. A hybrid approach to intricate motion, manipulation and task planning. *Int. J. Rob. Res.* 28(1):104–126.

Davis, and Ernest. 1987. Constraint propagation with interval labels. *Artif. Intell.* 32:281–331.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proc. of the 19th Int. Conf. on Automated Planning and Scheduling (ICAPS09)*, 114–122.

Gurobi optimizer, http://www.gurobi.com/.

Jaulin, L. 2000. Interval constraint propagation with application to bounded-error estimation. *Automatica* 36(10):1547–1552.

Karlsson, L.; Bidot, J.; Lagriffoul, F.; and Saffiotti, A. 2012. Combining task and path planning for a humanoid two-arm robotic system. In *TAMPRA'12 (ICAPS Workshop)*.

Kavraki, L.; Svestka, P.; Latombe, J.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Int. Conf. on Robotics and Automation*, 566–580.

L. Pack Kaelbling, T. L.-P. 2010. Hierarchical planning in the now. In *Proc. of Workshop on Bridging the Gap between Task and Motion Planning (AAAI)*.

LaValle, S. 2006. *Planning Algorithms*.

Lebbah, Y.; Rueher, M.; and Michel, C. 2002. A global filtering algorithm for handling systems of quadratic equations and inequations. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming*, CP '02, 109–123.

Lhomme, O., and Rueher, M. 1997. Application des techniques csp au raisonnement sur les intervalles. *Revue d'intelligence artificielle* 11(3):283–311.

Lhomme, O. 1993. Consistency techniques for numeric csps. In *Proceedings of the 13th international joint conference on Artifical intelligence*, 232–238.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*.

Ott, C., and al. 2006. A humanoid two-arm system for dexterous manipulation. In *2006 IEEE Int. Conf. on Humanoid Robots*, 276–283.

Zacharias F., Ch.Borst, G. H. 2007. Capturing robot workspace structure: representing robot capabilities. In *Proc. of IROS'07, the IEEE Int. Conf. on Intelligent Robots and Systems*, 3229–3236.

Combining Task and Path Planning for a Humanoid Two-arm Robotic System

Lars Karlsson and **Julien Bidot** and **Fabien Lagriffoul** and **Alessandro Saffiotti**
Centre for Applied Autonomous Sensor Systems (AASS), Örebro University, Sweden

Ulrich Hillenbrand and **Florian Schmidt**
German Aerospace Center (DLR), Oberpfaffenhofen, Germany

Abstract

The ability to perform both causal (means-end) and geometric reasoning is important in order to achieve autonomy for advanced robotic systems. In this paper, we describe work in progress on planning for a humanoid two-arm robotic system where task and path planning capabilities have been integrated into a coherent planning framework. We address a number of challenges of integrating combined task and path planning with the complete robotic system, in particular concerning perception and execution. Geometric backtracking is considered: this is the process of revisiting geometric choices (grasps, positions etc.) in previous actions in order to be able to satisfy the geometric preconditions of the action presently under consideration of the planner. We argue that geometric backtracking is required for resolution completeness. Our approach is demonstrated on a real robotic platform, Justin at DLR, and in a simulation of the same robot. In the latter, we consider the consequences of geometric backtracking.

Introduction

The robot Justin, which has been developed at the institute of Robotics and Mechatronics at the German Aerospace Center (DLR) in Oberpfaffenhofen, is an advanced humanoid robot with two arms with four-fingered human-like hands, a head with two video cameras for stereo vision, and a base with four wheels mounted on extensible legs. The upper body of Justin has 43 degrees of freedom: 7 degrees of freedom for each arm, 12 for each hand, 3 for the torso and two for the neck (Ott et al. 2006).

Justin, like other complex manipulators, was until recently dedicated to performing only tasks involving prespecified objects and action sequences, at least at an abstract level. In this article, we present the ongoing efforts to provide Justin with a higher degree of autonomy within the scope of the EU FP7-project GeRT (see `http://www.gert-project.eu`). Here, we focus on planning for tasks, but there is also work in the project on perception and grasping. The overall aim of the project is that Justin should be able to generalize from existing programs for specific tasks and from known objects of certain classes, to perform new tasks consisting of the same types of basic operations but combined in new ways, and with new objects belonging to the same classes. In this paper, we report on a prototype of a planner for Justin. Of course, this problem is not limited to Justin, but applies to other advanced robotic systems as well.

There is a large body of work on task planning which represents the world in logical terms (Nau, Ghallab, and Traverso 2004). But such representations are insufficient for modeling the kinematic and geometric properties of a robot such as Justin and its environment. When planning with Justin, one must take into account how it can move its arms and hands, how it can grasp different objects, and whether there are obstacles that can block its movements.

There is also a large body of work on path and motion planning (LaValle 2006). These algorithms plan in continuous state spaces, and include kinematic (or even dynamic) models of the robotic system as well as models (often in terms of polyhedrons) of obstacles.

While a path planner can find collision-free paths for various movements, it is not able to decide whether such a movement is a step in solving a complex task. A path planner is in general incapable of the kind of means-end reasoning that a task planner can do.

What Justin needs is a combination of task and path planning. It could be achieved by first solving the task planning problem and then for each action in that plan solving the corresponding path planning problem. However, it might happen that no path can be found for an action in the task plan. For instance, the presence of obstacles such as a big box in the middle of the workspace, may render certain parts of the workspace inaccessible for one or other of the arms. Yet, the task planner has no way of determining that, and may generate plans where the wrong arm is chosen for e.g. picking up an object from such a position. Hence, a solution with task planning first and path planning after might result in plans that are invalid at the geometric level. Instead, task and path planning must be integrated. A hybrid approach is required.

In this article, we present the hybrid task and path planning system we have developed for Justin. It is the first time that hybrid planning has been used for such a complex robotic system. Previous work has mainly relied on simulation and sometimes very simplified models. Thus, the fact that we adopt hybrid task and path planning for a real two-

armed humanoid robotic system is the first contribution of this paper. We present how the planner works together with other components of Justin's software, in particular for perception and execution, and we provide details of how the planner works. In particular, we show how several different solvers for path planning, linear interpolation of paths, and inverse kinematics need to be combined in order to find paths. This is the second contribution of this paper. We also consider the issue of geometric backtracking. Choices of how to perform actions at the geometric level may have negative consequences for later actions. Geometric backtracking is the process of revisiting geometric choices in previous actions in order to be able to apply the action presently under consideration. For instance, if the task is to place two cups on a small tray, the first cup may be placed in the middle of the tray, leaving insufficient space for the second cup. When the action to place the second cup is found to be inapplicable, one needs to go back to the first place action and reconsider where the first cup is to be placed. We show how geometric backtracking is performed in our planner, including how alternative geometric choices are sampled. We also argue that geometric backtracking is necessary for achieving resolution completeness for the hybrid planner. This is the third contribution of the paper. Finally, we present a number of demonstrations performed on the Justin platform, and experiments on a simulated version of Justin. The former demonstrate that our approach actually works on a real robotic system, and the latter investigate the benefits and costs of hybrid planning and in particular geometric backtracking. The experiments include a large obstacle (which makes path planning essential), and a number of objects that are put in a limited space (requiring geometric backtracking). In the demonstrations and experiments, the robot only manipulates objects for which it has a priori models, which implies that grasping can be done with previously stored grasps. While this paper focuses on Justin, we believe that what can be learned from planning with Justin can also be applied to other advanced robotic systems.

Related work

The approaches to combining task and path planning we have encountered in the literature can roughly be divided into two categories, defined in terms of how the task and path planning components relate to each other.

Path planning guided by task planning. In these approaches, path planning is primary, and task planning secondary. The planners mainly work on a path planning problem, but there is also a symbolic interpretation of the domain which can be used to structure the path planning problem and determine where to direct the search. These approaches include aSyMov (Cambon, Alami, and Gravot 2009) and SamplSGD (Plaku and Hager 2010). I-TMP (Hauser and Latombe 2009) should also be mentioned here, although it strictly speaking does not involve a task planning algorithm but a given task graph which represents a set of potential plans. These approaches address path planning problems involving a number of movable objects and/or multiple robots and/or a robot with many links. Such path planning problems have high-dimensional configuration spaces. In order to reduce that dimensionality, the problem is divided into tasks or actions corresponding to lower-dimensional subproblems. Such actions can for instance be to move one single object to a specific position while all other objects remain in position. The role of the task planner is to determine what actions/subproblems are to be explored. For instance, aSyMov only invokes the task planner as a heuristic for selecting actions.

Task planning querying path planning. In these approaches, a task plan is generated, and some of the actions involve path planning problems which are solved by dedicated path planners. Each path planning problem is solved separately. These approaches include Guitton and Farges (2009), Alili et al. (2010), SAHTN (Wolfe, Marthi, and Russell 2010), semantic attachments (Dornhege et al. 2009a; 2009b), and HTN and motion planning (Kaelbling and Lozano-Perez 2010). Typically, specific clauses in the preconditions and/or effects invoke calls to a path planner. For instance, the semantic attachments represent a general approach to invoking external solvers. A precondition clause such as *([check-transit ?x ?y ?g])* may invoke a call to a path planner. Information about the current robotic configuration is encoded in the states of the task planner by terms *q1 ... qn* and the transformation matrix for the pose of object *o* is encoded by terms *p0(o) ... p11(o)*.

It is noteworthy that these approaches have rarely been applied to real robots. With the exception of I-TMP, which has been demonstrated on a climbing Kapuchin robot, they are (as far as we know) only tested on simulated systems or very simple robots.

Our approach for Justin belongs to the second category. Besides being aimed at an advanced real robot, it also distinguishes itself by using geometric backtracking: only an extended abstract by Alili et al. (2010) appears to address that topic before (and only briefly, so there is not sufficient information to make a comparison). However, the first category of planners such as aSyMov (Cambon, Alami, and Gravot 2009) can perform similarly by exploring multiple paths between states.

Task and path planning

In task planning (Nau, Ghallab, and Traverso 2004) a state s is a set of atomic statements $p(c_1, \ldots, c_n)$ where p denotes a property of or a relation between objects denoted by names c_i. An action a has preconditions P_a (a logical combination of statements) that specify in what states a is applicable, and effects E_a (for instance a set of literals) that specify how a state changes (i.e. what statements are added or deleted) when a is applied. A planning domain D consists of a set of actions, and a planning problem is comprised of a domain, an initial state s_0 and a goal formula g. A plan is a sequence of actions $P = (a_1, \ldots, a_n)$. The result of a plan is the state s obtained by applying the first action a_1 to get a state s', and then recursively applying the rest of the plan to s'. In a valid plan, each action a_i is applicable in the state obtained by applying the preceding actions $(a_1, \ldots, a_{i-1})$ starting from the initial state s_0. A solution to a planning problem is a plan P with actions from D which when applied to the initial state s_0 results in a state s in which the goal g is satisfied.

Fig. 1 shows an example of an action schema from one of the domains for Justin.

act: pick(h,g,o)
pre: empty(h) and graspable(o) and can-move-pick(h,g,o,τ)
eff: not empty(h) and grasped(h,o) and is-picked(h,g,o,τ)

Figure 1: Action schema for Justin, representing a set of *pick* actions. The parameter *h* indicates what hand to use (*left* or *right*), *g* indicates the type of grasp (e.g. *top*), *o* is an object (e.g. *cup1*), and τ represents a path to be followed by the hand *h* during a *pick* action.

Path planning (LaValle 2006), on the other hand, considers a continuous space. There is a world space $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$. The obstacles in the world space are defined by the obstacle space $\mathcal{O} \subseteq \mathcal{W}$. There is a robot which can be a rigid body $\mathcal{A}$ or a collection of connected links $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

The configuration space $\mathcal{C}$ is determined by the various translations and rotations that the robot (or its links) can perform. $\mathcal{A}(q)$ is the space occupied by the robot transformed according to the configuration q (and equivalently for $\mathcal{A}_1, \ldots, \mathcal{A}_n$). $\mathcal{C}_{obs}$ is the obstacle region in the configuration space, defined as the set of configurations where the interior (int) regions of $\mathcal{O}$ and $\mathcal{A}$ intersect. $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$ is the free space where the robot can move.

A path planning problem is defined by the above entities (the domain), a start configuration $q_1 \in \mathcal{C}_{free}$ and a goal configuration $q_G \in \mathcal{C}_{free}$. A solution to a path planning problem as defined above is a continuous path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ where $\tau(0) = q_1$ and $\tau(1) = q_G$. This is the most basic version of the path planning problem. There might for instance be parts (objects) that the robot can transport, or there might be multiple robots. In the case of Justin, the path planning problems addressed concern one 7-degree arm at a time, and any transported parts simply follow the hand. Hence, $\mathcal{C}$ for each path planning problem effectively consists of 7 parameters. The obstacle space $\mathcal{C}_{obs}$ comprises the table surface, objects on the table, and the other arm.

Task planning and path planning representations can be linked through the object names and atomic statements at the task planning level: certain names correspond to parts, positions or regions at the path planning level, and certain statements correspond to properties of or relations between the parts and/or the robot. A state s then consists of a symbolic component σ_s (a set of statements) and a geometric component γ_s (containing the current configuration and poses of objects). The preconditions P_a of an action a can refer to both the symbolic and geometric state components, and the effects E_a can alter them both.

In Fig. 1, the predicates *can-move-pick* in the preconditions and *is-picked* in the effects are geometric: the former concerns the existence of a path τ in γ_s from the present arm configuration for the selected arm to one where the selected object can be picked, and the latter updates γ_s to such a target configuration.

A statement that is interpreted geometrically can be true in many different geometric states. This implies that a geometric effect can be realized in many different ways. For instance, if an object o is to be positioned in a certain region r, then $in(o, r)$ can be achieved by a set of different poses which is constrained by the borders of r, the presence of other objects in r and so on. Hence, an action with an effect such as $in(o, r)$ can be implemented in many different ways geometrically. In addition, how it is implemented may affect subsequent actions. For instance, if a later action has the effect $in(o_2, r)$, it will be constrained differently depending on the selected pose for o, and in some cases may even be infeasible.

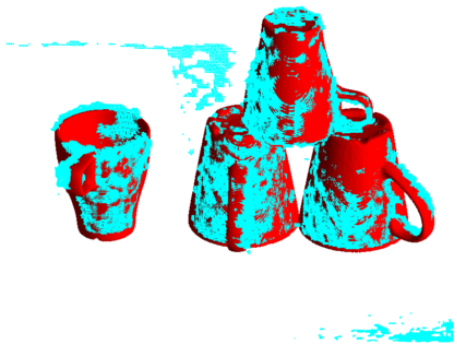


Figure 2: Point clouds (cyan) from a Kinect camera together with best matching models (red).

System overview

Here, we present the relevant modules of Justin: perception, world model, planning and execution.

The *perception module* is based on analysis of point clouds obtained from stereo processing or some other range sensor. Given geometric models of possible objects in the actual scene, an interpretation of that scene in terms of these objects is computed. In a first step, a set of hypotheses (several tens) is computed for each object through pose clustering (Hillenbrand 2008; Hillenbrand and Fuchs 2011). In the second step, these hypotheses are tested by aligning the object models with the data (Fig 2) and scoring the inliers by proximity to the model surface and similarity of surface orientation. Finally, collisions of models are detected and lead to pruning the hypotheses with the lowest scores.

The *world model* receives information from perception about classes, shapes and poses of objects in the scene. Presently, the world model has access to polygon mesh models of the corresponding example objects. When novel objects are introduced, these will have to be generated online from the point clouds obtained from range sensing. In particular, the objects' poses come with some uncertainty, and the robot's actions need to be robust enough to compensate for that.

In addition, the world model has access to a set of grasps for each object. These grasps are represented in terms of configurations for the individual fingers and the relative pose of the *tool center point* (TCP) which is roughly in the center of the wrist.

The *planner* queries the world model in order to get geometric information about the planning problem addressed. From the world model, it can construct an initial geometric state with the 3D models of the objects positioned in the correct poses. Purely symbolic information is given in a problem file, as is the goal. Objects in the geometric state are automatically given names from the symbolic states.

Next, the planner searches for a plan: how that is done will be described in the next section. If successful, the plan is used to generate a robot program in the form of a sequence of Python scripts. Each action model in the domain corresponds to one parameterized script segment, and each action in the plan generates one segment in the final script, instantiated with the appropriate objects and poses, grasps and possibly also paths. The scripts may contain, among other things, arm motions to specific frames, arm motions according to a given path, finger motions to given configurations, guards for specific conditions such as resistance due to contact with some object, perceptual operations such as looking for a specific object, and exception handling. These scripts are then executed in the Justin execution environment. Python was already extensively used in the execution environment, and provides an expressive and efficient high-level interface language between planning and execution. Fig. 3 shows one such Python script. In the future, we intend to add execution monitoring and recovery techniques to the system.

The planner

Our hybrid task and path planner is based on forward chaining task planning in combination with bidirectional rapidly exploring random tree (RRT) planning (LaValle 2006). Currently, we are using the hierarchical task network planner JSHOP2 (Nau et al. 2003) as the task planning component; it was chosen because it is a progressive planner and searches among fully specified states. From the perspective of the task planner, two modifications are made:

- The state is augmented with a geometric component, which contains information about the (predicted) configurations of the robot and of any movable objects, as well as their shapes (the latter are the same for all states).
- Atomic statements with certain predicates are not evaluated in the symbolic component of the state but in the geometric one. When such a statement is encountered while testing a precondition of an action, a method is called that evaluates whether it is true in the geometric state component. When a statement with a geometric predicate is encountered while adding the effects of an operator, a method is called which updates the geometric state accordingly.

Thus, the application of an action results in updating both the symbolic state, by adding/removing statements, and the geometric state, by invoking the associated methods. Notice, that the interaction between task and path planning occurs exclusively through the geometric predicates, in the pre- and postconditions of operators and possibly when the goal is evaluated. Thus, the only modifications of the task planner are how preconditions and effects are applied, and the inclusion of a geometric component into the state.

A method for the geometric state may be of two kinds. It may involve a simple computation, e.g. if it concerns the position of a certain object. It may also involve a more complex computation such as searching for a path in the configuration space of one of the arms. The latter is done as follows for a statement with the predicate *can-move-pick*:

1. The present configuration in the geometric state is the initial configuration for the path planning.
2. The goal configuration is computed by first determining a desired pose for the tool center point of the selected arm. There are typically several alternative grasps and hence several TCP poses that constitute a sample. An inverse kinematic solver for Justin's arms then computes a set of arm configurations that puts the tool center point in the desired pose. These are tested for collisions, and one of those found to be collision free will be the goal configuration.
3. Inverse kinematics is also used to generate a configuration at some distance from the object, and this will be the approach configuration. Passing through this configuration reduces the risk of failing the grasp due to e.g. unexpected collisions between the hand and the object.
4. A bidirectional RRT planner attempts to find a collision-free path for the arm between the initial and approach configurations. The RRT planner employs a forward-kinematic model for projecting Justin's arm into the work space.
5. A path between the approach configuration and the goal configuration is computed by linear interpolation.
6. The fingers are closed according to a grasp-specific configuration.
7. If a path is found, it is stored for later use, and the statement is considered true in the state.

The predicate *can-move-pick* is used in preconditions. The corresponding effect predicate, *is-picked*, updates the geometric state such that the selected arm and hand are set to the target configuration generated by *can-move-pick*. In addition, the grasped object is constrained to follow the hand.

For the predicate *can-move-place*, which is used when grasped objects are moved to a new position, the sequence is: select a target pose (there may be many, if the target is an extended area), compute inverse kinematics for this pose, compute inverse kinematics for an approach pose and a lift pose, do linear interpolation between the start and lift configurations, call the RRT planner for a path between the lift and approach configurations, and do linear interpolation between the approach and target configurations.

As mentioned before, each type of action is associated with a parameterizable Python script that can be executed on the robot. The parameters include paths found during path planning, and these are subject to smoothing in the script before they are executed. The scripts may also contain guards in order to detect e.g. when an object that is to be placed has contact with the table. The scripts for the actions in the plan

```
### user header 'pyrs_source'
# move hand to pregrasp config with side left, type top and object 4
path0 = [[0.2443, 0.0873, 0.1745, -0.0, 0.0, 0.6109, -0.0, 0.0, 0.5236, -0.0, 0.0, 0.6109]]
execute_path(path0, path_is_for_manipulators=['left_hand'])
# move arm to REAL pre-grasp (RRT-path which needs shortening):
path1_1 = [[-0.7907, -1.4714, 0.239, 1.6179, 0.7105, -0.6163, 0.6261],... ]
execute_path(path1_1, path_is_for_manipulators=['left_arm']) # <-- with joint path shortener!
# move arm to REAL grasp (along a Cartesian line without shortening):
path1_2 = [[-0.9226375374193561, 1.3244301190878176, 0.9000000000000015, 1.473463845957323,
2.441791818050337, 0.8452778932497086, -0.12911968091025275],...]
execute_path(path1_2, path_is_for_manipulators=['left_arm'], skip_optimization=True)
# grasp it!
path2 = [[0.2443, 0.2793, 0.3142, -0.0, 0.384, 0.6109, -0.0, 0.384, 0.5236, -0.0, 0.384, 0.6109]]
execute_path(path2, path_is_for_manipulators=['left_hand'])
rave.bind('leftArm', 'mug1_1')
execute_path(path1_2[::-1], path_is_for_manipulators=['left_arm'], skip_optimization=True)
exit('out')
```

Figure 3: Python script generated by planner for a grasping action (paths have been truncated). Note the different phases: pregrasp configuration for hand and then for arm, grasp configuration for arm, and actual grasp with hand.

are then executed in a sequence, making the robot perform the plan.

Geometric backtracking

When the planner selects an action, it not only chooses the type of action and what objects and locations are involved. It also needs to make geometric choices that are not visible to the task planner, but are related to the interpretation of certain geometric statements. These might be the exact TCP to use when grasping an object as in the statements *can-move-pick(h,g,o,τ)* and *is-picked(h,g,o,τ)*. These might also concern the exact pose when an object is put down at a given location, as in *can-move-place(h,g,o,τ)* and *is-placed(h,g,o,τ)*. As mentioned before, such choices may very well affect the applicability of actions later on. However, these choices are done locally, and are not informed about constraints imposed by subsequent actions. Hence, it is important that they can be reconsidered. Otherwise, the planner would be incomplete relative to its sampling at the geometric level. For instance, consider the example with the small tray and two cups from the introduction. If the planner only tries to put the first cup in the center of the tray (this might be the first sampled placement), then it will never find a placement for the second cup and will ultimately fail to find a plan.

To sample statements with geometric predicates in a systematic manner, we use the van der Corput and Halton sequences (Kuipers and Niederreiter 2005). These sequences guarantee a uniformly distributed sampling over $[0,1]^n$, and can straightforwardly be used to sample a bounded n-dimensional space.

Backtracking occurs along a single sequence of actions/states

$$(a_n, s_n, a_{n-1}, s_{n-1}, \ldots, a_{n-k}, s_{n-k})$$

where a_n is the most recent action. Other parallel search branches at the task planning level are unaffected. Backtracking is triggered when a_n is not applicable because some particular geometric statement related to motion was false. The most recent van der Corput indices that were used for geometric sampling for each action are also maintained: $(i_n, i_{n-1}, \ldots, i_{n-k})$. The most recent index i_n is incremented, giving a new geometric sample for the previously failed geometric predicate. If that fails, the procedure is repeated. If a maximal value for the index has been reached, it is reset to 0, and we move up one step to a_{n-1} and increment its index i_{n-1}, giving a new geometric sample at that level and for the relevant geometric predicates there. If successful, we update the geometric state γ_{s_n} and move downwards to a_n and i_n again. If after repeated failures at a_{n-1} the index i_{n-1} reaches its maximal value, it is reset to 0 and we move up yet another level and so on, in a recursive manner. Fig. 4 shows an example of geometric backtracking.

In our current implementation, we use the van der Corput sequence to sample (1) orientations of TCP relative to target object for the *can-move-pick* predicate where the domain $[0,1]$ of the van der Corput sequence is mapped to $[0, 2\pi]$ (radians), and (2) position (*x* and *y*) and (optionally) orientation for the *can-move-place* action. In general, the sampling schema is built into the interpretations of the various predicates.

Another important factor is the sample size, which determines the resolution at the geometric level. Presently, we have a fixed size for each predicate. This is basically where we define the balance between task planning and path planning: a large sample size will effectively result in more effort being spent on path planning and inverse kinematics tests for each action.

Overall, we consider how to perform the sampling and how large to make the sample size as central questions in hybrid task and path planning. This issue will be discussed further within the context of our experiments, where we also give some concrete examples of the utility of geometric backtracking.

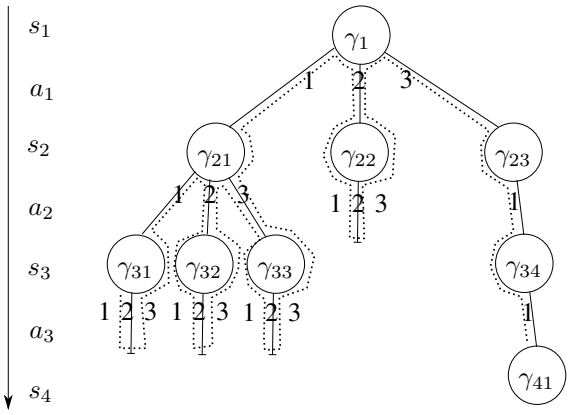


Figure 4: Geometric backtracking. The arrow shows the temporal order of the (partial) plan being explored. The circles with γ_k are different geometric states generated that belong to the hybrid states s_k (the symbolic components of the latter do not change), a_k are actions with geometric preconditions and/or effects, and the numbers on the lines are van der Corput indices that are used for sampling. The lines ending with a horizontal stroke are failed attempts to satisfy the corresponding p_k. The dashed line shows the order of traversal, starting from γ_{31} with the failed application of action a_3 and ending successfully in γ_{41} with the same action. Note, that we always consider the same sequence of actions: what varies is the sampling at the geometric level.

Demonstrations on real Justin

A number of demonstrations have been performed on the real Justin robot. The purpose with these demonstrations is to provide evidence that our hybrid planning approach is relevant to a real robot, and not just limited to simulation. They show how we have connected perception, planning and execution on the Justin robot.

We worked on two scenarios where small cups are to be manipulated by Justin. The initial geometric state was the same: 2 cups were placed on the table in front of Justin (see Fig. 5). Information about this geometric state was obtained through perception, as explained above. The planner had no a priori information about the geometry, but knew what objects would be present.

In the first scenario, the 2 cups are to be placed in a region on the table on the left side of Justin, which is shown with red in Fig. 5. Justin plans and successfully executes 4 symbolic actions with the left arm:

pick(left,top,cup2), place(left,red-region,cup2),
pick(left,top cup1), place(left,red-region,cup1).

Of course, this only shows the symbolic actions. In reality, the plan also includes specific paths for the arm and specific grasps, and those imply specific poses for the objects.

In the second scenario, the 2 cups are to be placed in two different regions on the table. *Cup2* is to be placed in a region on the right side of Justin (green). *Cup1* is to be placed first in a region in front of Justin (grey) and then it is to be placed in a region on the left side of Justin (red). Justin plans

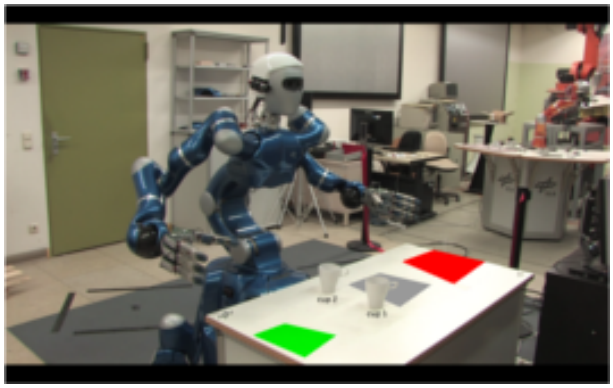


Figure 5: The setup of the first two scenarios. The regions where the cups should be placed are indicated.

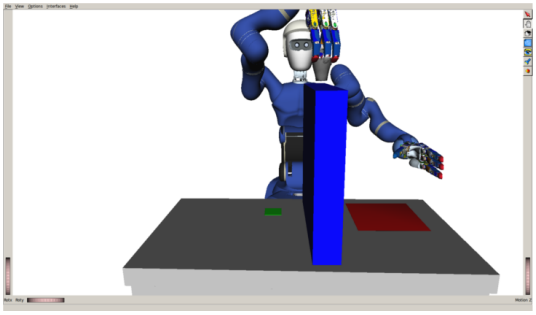


Figure 6: Simulated Justin using the top of the box as a temporary placement for the cup during the first series of experiments (P1).

and successfully executes 7 symbolic actions involving both arms:

pick(right,top,cup2), place(right,green-region,cup2),
pick(right,top,cup1), place(right,gray-region,cup1),
move-hand-away(right,cup1), pick(left,top,cup1),
place(left,red-region,cup1).

Note how *cup1* is handled by both hands, and how the right hand is moved away from it before it is grasped by the left hand.

Experiments in simulation

In addition to the demonstrations on the real Justin, a number of experiments in simulation have been conducted. The aims of these experiments are (1) to test the planner with more challenging tasks, and (2) to explore the benefits and costs of using geometric backtracking. The planner was running in Java 1.6.0 RTE with 32-bit native libraries used for collision detection (VCOLLIDE), inverse kinematics, and forward kinematics computation. The computer had an Intel CORE i5 vPro processor (2.5GHz, 64 bits, 2 cores, 3MiB for the cache memory), 8 GB memory and Linux kernel 2.6.38.

The first series of experiments involves a large box at the center of the table. The presence of this box causes repeated

failures of the path planner when objects placed near the center are to be moved from one side of the table to the other. However, the box can also be used for temporarily placing objects (Fig. 6), and this gives the robot an opportunity to shift hands. Hence, the following plan works well for moving a single cup from one side to the other:

pick(right,top,cup1), place(right,box-top,cup1), move-hand-away(right,cup1), pick(left,top,cup1), place(left,red-region,cup1)

Table 1 presents the results from these experiments (the three lines marked P1, with varying sampling resolution).

What is striking is the amount of time spent on geometric reasoning (inverse kinematics and path planning): it is several orders of magnitude more than the time spent on task planning. We present average path planning times with four decimal precision only to show how little time is spent on task planning. Also note the number of times the geometric reasoning fails to find a solution. This indicates that if we had solely relied on task planning, the risk of obtaining a plan that was not executable due to obstacles and kinematic constraints would have been considerable. Inspections of the logs of the planning process confirm this.

The problem was solved with varying resolution in the sampling for the geometric backtracking. Lower values for resolution where also tried, but then the planner often failed to find a solution. Not surprisingly, the choice of resolution strongly influences the total planning time, and the number of geometric configurations considered. Most of the time by far was spent on geometric backtracking (compare columns *#conf* and *#fail conf* to *#conf bt* and *#fail c bt*).

The second series of experiments involves moving a number of cans (shaped as cylinders) onto a tray positioned on the table. Each tray had a fairly small area which requires planning when putting more than one can on it. We varied both the number of cans — 2, 3 or 4 — and the size of the trays: they could just fit 2, 3 or 4 cans, respectively. We made a series of runs where the position of one cup can trigger a geometric backtracking when later cups are placed. Without geometric backtracking, the problems could not be solved. The planner would simply have considered the second (or third) placement action as inapplicable, and would have backtracked at the task planning level.

The following is a plan generated for putting three cans on a tray large enough for three cans (P3).

pick(right,top,can1), place(right,tray,can1), pick(right,top,can2), place(right,tray,can2), pick(right,top,can3), place(right,tray,can3).

Table 1 shows the results of the experiments. The problems are: P2 with 2 cans and a tray for 2, P3 with 3 cans and a tray for 3, P4 with 4 cans and a tray for 4, P5 with 2 cans and a tray for 4, and P6 with 3 cans and a tray for 4. Again, a considerable amount of time is spent on geometric reasoning and especially backtracking. The task planning problem, on the other hand, requires comparatively little effort. However, for P5 and P6 (with plenty of extra space on the tray), there is none or little backtracking.

A general problem appears to be to determine in advance how much geometric backtracking (if any) is needed. This is due to the complexity of the problem: each arm is a 7-degrees of freedom system with complex kinematics, there are constraints on how the objects can be grasped, there are obstacles that constrain movements (including other objects that can be moved), and the goal positions can be constrained in different ways. Hence, it may be a good idea to for instance iteratively increase the resolution instead of setting a fixed level. This also applies to the maximal number of nodes for path planning. We should also point out that the way we sample at the geometric level may not always be optimal. For instance, in order to fit several objects into a constrained area, it might be better to focus the sampling near the borders of the (remaining) space.

Summary and conclusions

In this article, we have presented a prototype of a hybrid task and path planning system for a humanoid two-arm robotic system. It is as far as we know the first time hybrid task and path planning has been applied to such an advanced robotic system. The planning system integrates a task planner and several methods for generating paths: a bidirectional RRT planner for longer movements and linear Cartesian interpolators for shorter approach and lift motions. The interface between task and path planning consists of a number of geometric predicates that can occur in the preconditions and effects of actions. We have presented how the planner works together with other components of the robot's software, and we have demonstrated that we have a working system. We also consider the issue of geometric backtracking: the process of revisiting geometric choices in previous actions in order to be able to apply the action presently under consideration. Simulated experiments have been made to illustrate the utility of geometric backtracking. We think that efficient methods for geometric backtracking (in particular how to sample and how much to sample) are vital for achieving efficient hybrid planning, and much remains to be done.

The current system has some limitations. First, there is no grasp planning available yet, which restricts us to a priori known objects for which there is a set of directly applicable grasps. Second, although some robustness towards uncertainty has been built into the path planning and the produced Python scripts, a monitoring and recovery component would also be needed. Both these issues will be addressed in the near future.

Acknowledgements The project receives funding from the European Community's Seventh Framework Programme under grant agreement no. ICT- 248273 GeRT.

References

Alili, S.; Pandey, A. K.; Sisbot, E. A.; and Alami, R. 2010. Interleaving symbolic and geometric reasoning for a robotic assistant. In *ICAPS Workshop on Combining Action and Motion Planning*.

Cambon, S.; Alami, R.; and Gravot, F. 2009. A hybrid approach to intricate motion, manipulation and task planning. *Int. J. Rob. Res.* 28(1):104–126.

Problem	Time	#plans	#sym bt	#acts	Time path	#conf	#fail conf	#conf bt	#fail c bt	Resol
P1	17.3638	10	5	5	17.3597	225.2	1087	215.2	950.8	6×60
P1	5.4522	10	5	5	5.4479	73.6	221.8	63.6	93.4	2×60
P2	4.7092	4	0	4	4.7053	26.1	2484.1	23.1	2362.9	16×120
P2	1.7159	4	0	4	1.7129	13.6	484.4	10.6	423.4	6×60
P2	0.7986	4	0	4	0.7962	8.4	82	5.4	50.9	2×30
P3	11.9506	6	0	6	11.9467	59.2	6064.6	55.2	5822.6	16×120
P3	4.1042	6	0	6	4.1006	30.4	1253.7	26.2	1143.1	6×60
P3	2.2240	6	0	6	2.2215	19.6	461.8	15.6	339.8	2×60
P4	1056.9276	8	0	8	1056.9241	4193.7	459769.7	4186.7	459564.7	16×120
P4	154.1280	8	0	8	154.1246	716	37909.4	709.1	37819.7	12×60
P4	15.0828	8	0	8	15.0792	84	3647	77	3562	6×60
P5	0.3184	4	0	4	0.3162	4	9	0	0	16×120
P5	0.3420	4	0	4	0.3393	4	9	0	0	6×60
P5	0.3175	4	0	4	0.3145	4	9	0	0	2×30
P6	0.7151	6	0	6	0.7121	6.1	89.4	0.2	2.2	16×120
P6	0.5324	6	0	6	0.5284	6.1	30.5	0.2	2.3	6×60
P6	0.6374	6	0	6	0.6349	6.6	40.6	0.7	12.4	2×60

Table 1: Results from simulated experiments. *Time* is the time in second spent during the whole planning process (both task and path planning). *#plans* is the number of partial symbolic plans that were visited during the whole planning process. These plans consist of primitive actions. *#symb bt* is the number of times the task planning process has backtracked. *#acts* is the number of symbolic primitive actions of the solution plan. *Time path* is the time in second spent on path planning. *#config* is the number of geometric configurations that were visited during the path planning process. *#fail conf* is the number of failed attempts to create geometric configurations. *#conf bt* is the number of geometric configurations that were visited during geometric backtracking. *#fail c bt* is the number of failed attempts to create geometric configurations during geometric backtracking. *Resol(ution)* indicates the number of grasps that are tried to pick an object (cup or can) and the number of poses that are tried for the place actions. Each line presents the average of 10 runs. The RRT planner had a limit of 500 nodes per attempt.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009a. Semantic attachments for domain-independent planning systems. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS09)*, 114–122.

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

Guitton, J., and Farges, J.-L. 2009. Taking into account geometric constraints for task-oriented motion planning. In *Proc. Bridging the gap Between Task And Motion Planning, BTAMP'09 (ICAPS Workshop)*.

Hauser, K., and Latombe, J.-C. 2009. Integrating task and PRM motion planning: Dealing with many infeasible motion planning queries. In *Proc. Bridging the gap Between Task And Motion Planning, BTAMP'09 (ICAPS Workshop)*.

Hillenbrand, U., and Fuchs, A. 2011. An experimental study of four variants of pose clustering from dense range data. *Computer Vision and Image Understanding* 115:1427–1448.

Hillenbrand, U. 2008. Pose clustering from stereo data. In *Proceedings VISAPP International Workshop on Robotic Perception (RoboPerc 2008)*, 23–32.

Kaelbling, L. P., and Lozano-Perez, T. 2010. Hierarchical planning in the now. In *Proc. of Workshop on Bridging the Gap between Task and Motion Planning (AAAI)*.

Kuipers, L., and Niederreiter, H. 2005. *Uniform distribution of sequences*. Dover Publications.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge, UK: Cambridge University Press.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research, special issue on the 3rd International Planning Competition* 20:379–404.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ott, C.; Eiberger, O.; Friedl, W.; Bäuml, B.; Hillenbrand, U.; Borst, C.; Albu-Schäffer, A.; Brunner, B.; Hirschmüller, H.; Kielhöfer, S.; Konietschke, R.; Suppa, M.; Wimböck, T.; Zacharias, F.; and Hirzinger, G. 2006. A humanoid two-arm system for dexterous manipulation. In *Proceedings IEEE-RAS International Conference on Humanoid Robots (Humanoids 2006)*, 276–283.

Plaku, E., and Hager, G. 2010. Sampling-based motion planning with symbolic, geometric, and differential constraints. In *Proceedings of ICRA10*.

Wolfe, J.; Marthi, B.; and Russell, S. J. 2010. Combined task and motion planning for mobile manipulation. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS10)*, 254–258.

Planning Robot Motions to Satisfy Linear Temporal Logic, Geometric, and Differential Constraints

Erion Plaku
Department of Electrical Engineering and Computer Science
Catholic University of America, Washington DC, 22064

Abstract

This paper shows how to effectively compute collision-free and dynamically-feasible robot motion trajectories that satisfy task specifications given by Linear Temporal Logic (LTL). The proposed approach combines sampling-based motion planning over the continuous state space with discrete search over both the LTL task representation and a workspace decomposition. In distinction from related work, the proposed approach samples the discrete space to shorten the length of the discrete plans and to more effectively guide motion planning in the continuous state space. Experimental results on various scenes, LTL specifications, and a snake-like robot model with nonlinear dynamics and numerous degrees-of-freedom (DOFs) show significant computational speedups over related work.

1 Introduction

Crucial to the goal of enabling robots to complete tasks on their own is their ability to plan at multiple levels of discrete and continuous abstractions. Whether the task is to search, inspect, manipulate objects, or navigate, it generally involves abstractions into discrete actions, which often require substantial continuous motion planning to carry out.

The coupling of the discrete and the continuous, however, poses significant challenges as discrete and continuous planning have generally been treated separately. On the one hand, while discrete planning can take into account sophisticated task specifications, it has generally been limited to discrete worlds (Ghallab, Nau, and Traverso, 2004). On the other hand, while motion planning can take into account obstacles, dynamics, and other continuous aspects of the robot and the world, due to the increased complexity, it has generally been limited to simple tasks, such as planning motions to reach a goal state. (Choset et al., 2005; LaValle, 2006).

To bridge the gap between the discrete and the continuous, researchers are proposing to incorporate task specifications directly into motion planning (Bhatia et al., 2011; Cambon, Alami, and Gravot, 2009; Ding et al., 2011; Fainekos et al., 2009; Hauser and Ng-Thow-Hing, 2011; Kress-Gazit et al., 2007; Plaku and Hager, 2010; Plaku, Kavraki, and Vardi, 2009, 2012; Wolfe, Marthi, and Russell, 2010). More specifically, the problem being studied in this line of research requires generating a collision-free and dynamically-feasible motion trajectory that satisfies a given task specification. In this context, LTL has often been used as the discrete logic in which to express the tasks. LTL makes it possible to express tasks in terms of propositions, logical connectives ($\wedge$ and, $\vee$ or, $\neg$ not), and temporal connectives ($\bigcirc$ next, $\Diamond$ eventually, $\Box$ always, $\cup$ until, $\mathcal{R}$ release). As an illustration, the task of inspecting all the areas of interest can be expressed as

$$\Diamond \pi_{A_1} \wedge \ldots \wedge \Diamond \pi_{A_n}, \tag{1}$$

where π_{A_i} denotes the proposition "robot inspected area A_i." As another example, the task of reaching A_1 before A_2 and A_3 can be expressed as

$$(\neg \pi_{A_2} \wedge \neg \pi_{A_3}) \cup (\pi_{A_1} \wedge \Diamond \pi_{A_2} \wedge \Diamond \pi_{A_3}).$$

The problem of planning motions that satisfy an LTL specification ϕ is often approached by first using model checking to compute a sequence of propositional assignments $\sigma = \tau_1, \tau_2, \ldots$ that satisfies ϕ. Referring again to the LTL specification in Eq. 1, σ could be obtained by setting each $\tau_i = \{\pi_{A_i}\}$, i.e., π_{A_i} is true and every other proposition is false. In a second stage, controllers are used to enable the robot to satisfy the propositions in the order specified by σ. In this context, discrete logic is used in (Arkin, 1990; Payton, Rosenblatt, and Keirsey, 1990; Saffiotti, Konolige, and Ruspini, 1995) to combine a set of behavior schemas which use controllers to link motion to abstract actions. This idea has been revisited more recently in (Kress-Gazit et al., 2011) to synthesize reactive controllers from LTL for car-like systems and in (Ding et al., 2011) to deploy robotic teams in urban environments. The work in (Fainekos et al., 2009) uses LTL to determine the sequence of triangles a point robot needs to visit and relies on a controller to drive the robot between adjacent triangles.

A limitation of these approaches is that it is generally not known in advance which propositional assignments are actually feasible in the continuous world. The LTL specification could present exponentially many alternative discrete solutions, as it is the case in Eq. 1. The underlying assumption in these approaches is that any sequence of propositional assignments that satisfies the LTL specification can be carried out in the continuous world. However, as a result of geometric constraints imposed by obstacle avoidance, the geometric shape of the robot, and the underlying motion dynamics, it may be impossible to carry out certain propositional assignments τ_i. This is in fact one of the main challenges when

incorporating LTL task specifications directly into motion planning, limiting the applicability of these approaches to specific systems and to specific discrete actions for which controllers are available.

In order to be generally applicable, recent work by the author (Plaku and Hager, 2010; Plaku, Kavraki, and Vardi, 2009, 2012) and others (Bhatia et al., 2011) has proposed a two-layered approach that couples the ability of sampling-based motion planning to handle the complexity arising from high-dimensional robotic systems, nonlinear motion dynamics, and collision avoidance with the ability of discrete planning to take into account discrete specifications. While discrete planning guides sampling-based motion planning, the latter feeds back information to further refine the guide and advance the search toward a solution that satisfies the LTL specification. Other approaches that utilize discrete search and sampling-based motion planning have also been developed for multimodal motion planning (Hauser and Ng-Thow-Hing, 2011) and manipulation planning (Nieuwenhuisen, van der Stappen, and Overmars, 2006; Stilman and Kuffner, 2008; Stilman, 2010; Wolfe, Marthi, and Russell, 2010). These other approaches, however, do not take into account LTL specifications.

This paper builds upon the success of combining LTL with sampling-based motion planning (Bhatia et al., 2011; Plaku, Kavraki, and Vardi, 2009, 2012). The search for a collision-free and dynamically-feasible trajectory that satisfies the LTL specification is conducted simultaneously in both the continuous and discrete planning layers. Sampling-based motion planning extends a tree in the continuous state space by adding new trajectories as tree branches. Such trajectories are obtained by sampling input controls and propagating forward the motion dynamics of the robot. Discrete planning guides the sampling-based motion planner by searching over both the LTL formula representation and a workspace decomposition to provide discrete plans as intermediate sequences of propositional assignments that should be satisfied. In distinction from related work (Bhatia et al., 2011; Plaku, Kavraki, and Vardi, 2009, 2012), the proposed approach samples the discrete space to shorten the length of the discrete plans and to more effectively guide motion planning in the continuous state space. Moreover, the search is expanded toward new propositions that enable the sampling-based motion planner to make rapid progress toward obtaining a solution. Experimental results on various scenes, LTL specifications, and a snake-like robot with nonlinear dynamics and numerous DOFs show significant computational speedups over related work.

2 LTL Specifications

Let Π denote a set of propositions, where each $\pi_i \in \Pi$ corresponds to a Boolean-valued problem-specific statement, such as "robot is in area A_i." LTL combines propositions with logical connectives ($\neg$ not, $\wedge$ and, $\vee$ or), and temporal connectives ($\bigcirc$ next, $\Diamond$ eventually, $\Box$ always, $\cup$ until, $\mathcal{R}$ release). A discrete state $\tau_i \in 2^{\Pi}$ denotes all the propositions that hold true in the world. As the world changes, e.g., as the result of robot actions, the discrete state could also change. LTL planning consists of finding a sequence of discrete states $\sigma = [\tau_i]_{i=1}^{n}$ that satisfies a given LTL formula, whose syntax and semantics are defined below.

2.1 LTL Syntax and Semantics

Every $\pi \in \Pi$ is a formula. If ϕ and ψ are formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\bigcirc\phi$, $\Diamond\phi$, $\Box\phi$, $\phi \cup \psi$, $\phi\mathcal{R}\psi$ are also formulas. Let $\sigma = \tau_0, \tau_1, \ldots$, where each $\tau_i \in 2^{\Pi}$. Let $\sigma^i = \tau_i, \tau_{i+1}, \ldots$ denote the i-th postfix of σ. The notation $\sigma \models \phi$ indicates that σ satisfies ϕ and is defined as

- $\sigma \models \pi$ if $\pi \in \Pi$ and $\pi \in \tau_0$;
- $\sigma \models \neg\phi$ if $\sigma \not\models \phi$;
- $\sigma \models \phi \wedge \psi$ if $\sigma \models \phi$ and $\sigma \models \psi$;
- $\sigma \models \bigcirc\phi$ if $\sigma^1 \models \phi$;
- $\sigma \models \phi \cup \psi$ if $\exists\, k \geq 0$ such that $\sigma^k \models \psi$ and $\forall\, 0 \leq i < k : \sigma^i \models \phi$.

The other connectives are defined as $\texttt{false} = \pi \wedge \neg\pi$, $\texttt{true} = \neg\texttt{false}$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\Diamond\phi = \texttt{true} \cup \phi$, $\Box\phi = \neg\Diamond\neg\phi$, and $\phi\mathcal{R}\psi \equiv \neg(\neg\phi \cup \neg\psi)$. More details can be found in (Kupferman and Vardi, 2001).

2.2 Co-Safe LTL and Automata Representation

Since LTL planning is PSPACE-complete (Sistla, 1994), as in related work (Bhatia et al., 2011; Plaku, Kavraki, and Vardi, 2009, 2012), this paper considers co-safe LTL. Co-safe LTL formulas are satisfied by finite sequences of discrete states rather than infinite sequences which satisfy general LTL formulas. Most robotic tasks are finite in nature, hence, the use of co-safe LTL does not limit the general applicability of the approaches. Co-safe LTL formulas can be translated into NFAs (Nondeterministic Finite Automata) with at most an exponential increase in size (Kupferman and Vardi, 2001). As recommended in (Armoni et al., 2005; Plaku, Kavraki, and Vardi, 2012), NFAs are converted into DFAs (Deterministic Finite Automata), which are then minimized. A DFA search has a significantly smaller branching factor, since there is exactly one transition that can be followed from each state for each propositional assignment, while when using an NFA there are generally many more.

Formally, a DFA is a tuple $\mathcal{A} = (Z, \Sigma, \delta, z_{init}, \mathit{Accept})$, where Z is a finite set of states, $\Sigma = 2^{\Pi}$ is the input alphabet, $\delta : Z \times \Sigma \to Z$ is the transition function, $z_{init} \in Z$ is the initial state, and $\mathit{Accept} \subseteq Z$ is the set of accepting states. The state obtained by running $\mathcal{A}$ on $\sigma = [\tau_i]_{i=1}^{n}$, $\tau_i \in 2^{\Pi}$, starting from the state z is defined as

$$\mathcal{A}([\tau_i]_{i=1}^{n}, z) = \begin{cases} z, & n = 0 \\ \delta(\mathcal{A}([\tau_i]_{i=1}^{n-1}, z), \tau_n), & n > 0. \end{cases}$$

$\mathcal{A}$ accepts σ iff $\mathcal{A}(\sigma, z_{init}) \in \mathit{Accept}$. As a result, $\sigma \models \phi$ when the equivalent automaton $\mathcal{A}$ accepts σ.

To facilitate presentation, let *Reject* denote all the rejecting states of $\mathcal{A}$, i.e., states that cannot reach an accepting state. Moreover, let $\delta(z)$ denote all the non-rejecting automaton states connected by a single transition from z, i.e.,

$$\delta(z) = \{\delta(z, \tau) : \tau \in 2^{\Pi}\} - \mathit{Reject}.$$

2.3 Interpretation of LTL over Motion Trajectories in the Continuous State Space

The continuous state space $\mathcal{S}$ gives meaning to the propositions in the task specification. As an example, the proposition "robot is in area A_i" holds iff the robot is actually in A_i. The meaning of each proposition $\pi_i \in \Pi$ is defined by a function $\text{HOLDS}_{\pi_i} : \mathcal{S} \rightarrow \{\texttt{true}, \texttt{false}\}$, where $\text{HOLDS}_{\pi_i}(s) = \texttt{true}$ iff π_i holds at the continuous state $s \in \mathcal{S}$. This interpretation provides a mapping $\text{DISCRETESTATE} : \mathcal{S} \rightarrow 2^{\Pi}$ from the continuous state space $\mathcal{S}$ to the discrete space 2^{Π}, i.e.,

$$\text{DISCRETESTATE}(s) = \{\pi_i : \pi_i \in \Pi \wedge \text{HOLDS}_{\pi_i}(s) = \texttt{true}\}.$$

Moreover, trajectories over $\mathcal{S}$ give meaning to temporal connectives. A trajectory over $\mathcal{S}$ is a continuous function $\zeta : [0, T] \rightarrow \mathcal{S}$, parametrized by time. As the continuous state changes according to ζ, the discrete state, obtained by the mapping DISCRETESTATE, may also change. In this way, ζ gives rise to a sequence of discrete states, $\text{DISCRETESTATES}(\zeta) \stackrel{def}{=} [\tau_i]_{i=1}^{n}$, where $\tau_i \neq \tau_{i+1}$. As a result of this mapping, ζ is said to satisfy an LTL formula ϕ iff $\text{DISCRETESTATES}(\zeta) \models \phi$.

Note that the underlying motion dynamics of the robot are specified as a set of differential equations $f : \mathcal{S} \times \mathcal{U} \rightarrow \dot{\mathcal{S}}$, where $\mathcal{U}$ is a control space consisting of a finite set of input variables that can be applied to the robot (e.g., a car can be controlled by setting the acceleration and the rotational velocity of the steering wheel). The approach can take into account general nonlinear dynamics, relying only on a simulator, which, when given a state s, an input control u, and a time step dt, computes the new state that results from integrating the underlying motion dynamics.

A dynamically-feasible trajectory $\zeta : [0, T] \rightarrow \mathcal{S}$ starting at $s \in \mathcal{S}$ is obtained by computing a control function $\tilde{u} : [0, T] \rightarrow \mathcal{U}$ and propagating the dynamics forward in time through numerical integration of $\dot{\zeta}(h) = f(\zeta(h), \tilde{u}(h))$, i.e.,

$$\zeta(t) = s + \int_0^t f(\zeta(h), \tilde{u}(h))\, dh.$$

The dynamically-feasible trajectory $\zeta : [0, T] \rightarrow \mathcal{S}$ is considered collision free if each state along the trajectory avoids collisions with the obstacles.

2.4 Problem Statement

Given $\langle \mathcal{S}, \mathcal{U}, f, s_{init}, \Pi, \phi \rangle$ compute a control function $\tilde{u} : [0, T] \rightarrow \mathcal{U}$ such that the resulting dynamically-feasible trajectory $\zeta : [0, T] \rightarrow \mathcal{S}$ where

$$\zeta(t) = s_{init} + \int_0^t f(\zeta(h), \tilde{u}(h))\, dh$$

is collision free and satisfies the LTL specification ϕ, i.e., $\text{DISCRETESTATES}(\zeta) \models \phi$.

2.5 Examples

To facilitate presentation, this section provides examples of the robot model, workspaces, and the LTL specifications, which are used in the experiments.

Snake-Like Robot Model The snake-like robot model, as shown in Fig. 1, consists of several unit links attached to each other. The motion dynamics of the robot are modeled as a car pulling trailers (adapted from (LaValle, 2006, pp. 731)). The continuous state

$$s = (x, y, \theta_0, v, \psi, \theta_1, \theta_2, \ldots, \theta_N),$$

consists of the position $(x, y) \in \mathbb{R}^2$ ($|x| \leq 30, |y| \leq 25$), orientation $\theta_0 \in [-\pi, \pi)$, velocity v ($|v| \leq 2$), and steering-wheel angle ψ ($|\psi| \leq 1rad$) of the head link of the snake-like robot, and the orientation θ_i ($\theta_i \in [-\pi, \pi), 1 \leq i \leq N$) of each trailer link, where N is the number of trailers. The robot is controlled by setting the acceleration a ($|a| \leq 1$) and the rotational velocity ω ($|\omega| \leq 1rad/s$) of the steering-wheel angle. The differential equations of motions are

$$\dot{x} = v\cos(\theta_0) \quad \dot{y} = v\sin(\theta_0) \quad \dot{\theta}_0 = v\tan(\psi)$$
$$\dot{v} = a \quad \dot{\psi} = \omega$$
$$\dot{\theta}_i = \tfrac{v}{d}\left(\prod_{j=1}^{i-1}\cos(\theta_{j-1} - \theta_j)\right)(\sin(\theta_{i-1}) - \sin(\theta))$$

where $1 \leq i \leq N$ and $d = 0.05$ is the hitch length. A continuous state s is considered valid iff the robot does not collide with the obstacles and the robot does not self intersect, i.e., non-consecutive links do not collide with each other.

While related work in LTL motion planning (Bhatia et al., 2011; Ding et al., 2011; Fainekos et al., 2009; Kress-Gazit et al., 2007) has generally focused on low-dimensional robotic systems, by increasing the number of trailer links, the snake-like robot provides challenging test cases for high-dimensional motion-planning problems with dynamics, as noted in (Laumond, 1993). In the experiments in this paper, the number of trailer links is varied as $0, 5, 10, 15$ yielding problems with $5, 10, 15, 20$ DOFs.

Workspaces The workspaces in which the robot operates, as in the related LTL motion-planning work (Bhatia et al., 2011; Ding et al., 2011; Fainekos et al., 2009; Kress-Gazit et al., 2007; Plaku, Kavraki, and Vardi, 2009, 2012), are populated with polygonal obstacles and propositions. In this way, a proposition π_i is associated with a polygon p_i, and the function $\text{HOLDS}_{\pi_i}(s)$ is true iff the position component of s is inside p_i. In addition, as in the related LTL motion-planning work, each workspace is triangulated (using the Triangle package (Shewchuk, 2002)). The triangulation conforms to the vertices of the polygons associated with the obstacles and the propositions, i.e., triangulation treats the polygonal obstacles and propositions as holes. The polygonal propositions $p_1, \ldots, p_k$ and the triangles $t_1, \ldots, t_m$ give rise to an adjacency region graph $G = (R, E)$, where

$R = \{p_1, \ldots, p_k, t_1, \ldots, t_m\}$ and

$E = \{(r_i, r_j) : r_i, r_j \in R \text{ are physically adjacent}\}$.

Each region $r \in R$ is labeled by the corresponding discrete state, i.e.,

$$\text{DISCRETESTATE}(r) = \begin{cases} \{\pi_\ell\}, & \text{if } r = p_\ell \text{ for some } 1 \leq \ell \leq k, \\ \emptyset, & \text{otherwise} \end{cases}$$

An illustration of the workspaces, propositions, and triangulations is provided in Fig. 1.

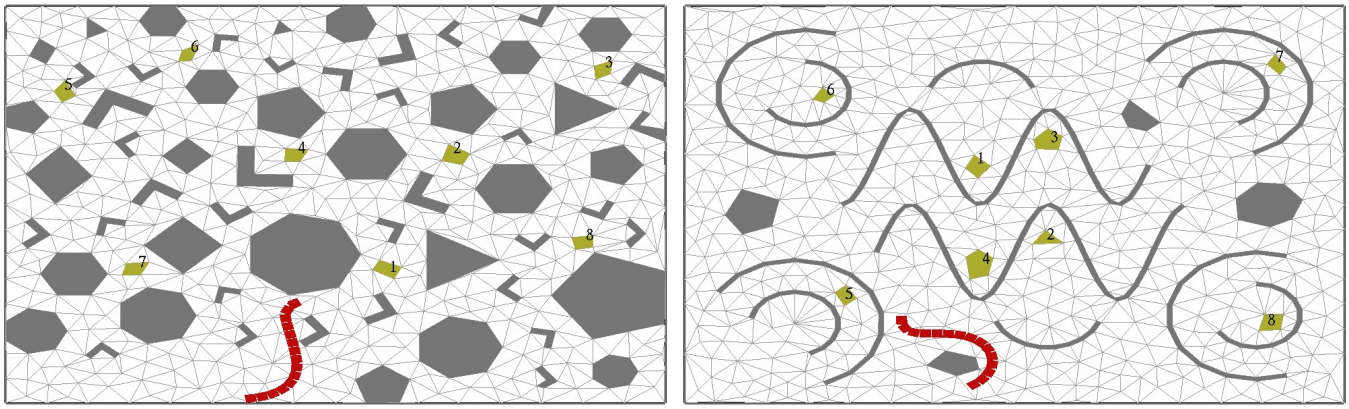


Figure 1: Workspaces used in the experiments. Obstacles are in gray. Propositions are in a golden color and are numbered $1, 2, \ldots, 8$. The robot is in red and is shown in some initial state. The workspace triangulations are also shown. These workspaces provide challenging environments, as the snake-like robot has to wiggle its way through various narrow passages.

LTL Tasks LTL tasks used in the experiments are defined over 8 propositions, as shown in Fig. 1.

The first task is to compute a collision-free and dynamically-feasible trajectory ζ which satisfies the propositions $\pi_1, \pi_2, \ldots, \pi_8$ in succession, i.e.,

$$\phi_1 = \beta \cup (\pi_1 \wedge ((\pi_1 \vee \beta) \cup (\pi_2 \wedge (\ldots (\pi_7 \vee \beta) \cup \pi_8)))),$$

where $\beta = \wedge_{i=1}^{8} \neg \pi_i$.

The second task is to compute a collision-free and dynamically-feasible trajectory which eventually satisfies $\pi_i, \pi_j, \pi_k, \pi_i, \pi_j$ with different i, j, k, i.e.,

$$\phi_2 = \bigvee_{1 \leq i,j,k \leq 8, i \neq j, j \neq k, i \neq k} \Diamond \pi_i \wedge (\Diamond \pi_j \wedge (\Diamond \pi_k \wedge (\Diamond \pi_i \wedge (\Diamond \pi_j)))).$$

This task presents polynomially different possibilities about which propositions to satisfy.

The third task is to compute a collision-free and dynamically-feasible trajectory ζ that eventually satisfies each proposition, i.e.,

$$\phi_3 = \bigwedge_{i=1}^{8} \Diamond \pi_i.$$

This task presents combinatorially many different possibilities regarding the order in which to satisfy the propositions.

3 Method

Let $s_{init} \in \mathcal{S}$ denote the initial state of the robot. Let ϕ denote the LTL specification. In order to compute a collision-free and dynamically-feasible trajectory ζ that satisfies ϕ, i.e., $\textsc{DiscreteStates}(\zeta) \models \phi$, as mentioned earlier, the approach couples sampling-based motion planning in the continuous state space $\mathcal{S}$ with discrete search over both the automaton $\mathcal{A}$ representing LTL and the region graph $G = (R, E)$. To facilitate presentation, the general idea is described first followed by details of the approach.

3.1 Coupling Sampling-based Motion Planning and Discrete Search

Sampling-based motion planning uses a tree data structure $\mathcal{T}$ as the basis for conducting the search in the continuous space $\mathcal{S}$. Each vertex $v_i \in \mathcal{T}$ is associated with a collision-free continuous state, denoted as $\mathit{state}(v_i)$. The vertex v_i also keeps track of its parent in $\mathcal{T}$, denoted as $\mathit{parent}(v_i)$, and the collision-free and dynamically-feasible trajectory that connects its parent state to $\mathit{state}(v_i)$, denoted as $\mathit{ptraj}(v_i)$. Initially, $\mathcal{T}$ contains only its root vertex v_1, where $\mathit{state}(v_1)$ corresponds to the initial continuous state s_{init}. As the search progresses, $\mathcal{T}$ is extended by adding new vertices v_j and new collision-free and dynamically-feasible trajectories $\mathit{ptraj}(v_j)$ as branches of $\mathcal{T}$. As described in Section 3.3, these collision-free and dynamically-feasible trajectories are obtained by sampling controls and propagating forward for several steps the motion dynamics of the robot, and stopping the propagation if a collision is found.

Let $\mathit{traj}(\mathcal{T}, v_j)$ denote the trajectory from $\mathit{state}(v_1)$ to $\mathit{state}(v_j)$, which is obtained by concatenating the collision-free and dynamically-feasible trajectories associated with the tree edges from v_1 to v_j. Then, $\mathit{traj}(\mathcal{T}, v_j)$ satisfies the LTL formula ϕ iff $\textsc{DiscreteStates}(\mathit{traj}(\mathcal{T}, v_j)) \models \phi$. Moreover, recall that $\textsc{DiscreteStates}(\mathit{traj}(\mathcal{T}, v_j)) \models \phi$ iff the sequence of discrete states ends up on an accepting state when run on the automaton $\mathcal{A}$ representing ϕ. In order to make this computation efficient, each vertex v_j is associated with the automaton state, denoted as $z(v_j)$, obtained by running $\textsc{DiscreteStates}(\mathit{traj}(\mathcal{T}, v_j))$ on $\mathcal{A}$. The computation of $z(v_j)$ is done incrementally when checking $\mathit{ptraj}(v_j)$ for collisions, as described in Section 3.3. As it will be explained in the next paragraph, v_j is also associated with $\mathit{region}(v_j)$, which denotes the workspace region that contains the position component of $\mathit{state}(v_j)$.

The objective of the discrete layer is to guide sampling-based motion planning as it expands $\mathcal{T}$ in the continuous space $\mathcal{S}$. To do so effectively, the discrete layer selects at each iteration an automaton state z_{from} and a workspace region r_{from} from which to expand the search and an automa-

Algorithm 1 LTLMOTIONPLANNING($\Pi, \phi, G, s_{init}, f, t_{max}$)

Input
Π: propositions
ϕ: LTL formula
$G = (R, E)$: workspace region graph
s_{init}: initial state of the robot
$f : \mathcal{S} \times \mathcal{U} \rightarrow \dot{\mathcal{S}}$: differential equations of motion
t_{max}: upper bound on time
Output: If successful, method computes a collision-free and dynamically-feasible trajectory ζ that satisfies ϕ

1: $\mathcal{A} \leftarrow$ AUTOMATON(ϕ); $\mathcal{T} \leftarrow$ CREATETREE(s_{init})
2: **while** TIME() $< t_{max}$ **and** SOLVED() = `false` **do**
$\diamondsuit$ *discrete layer*
3: $\langle z_{from}, r_{from}, z_{to} \rangle \leftarrow$ DISCRETETARGET($\mathcal{T}, \mathcal{A}, G$)
4: $\sigma \leftarrow$ DISCRETEPLAN($\mathcal{A}, G, z_{from}, r_{from}, z_{to}$)
5: $\sigma_{active} \leftarrow \emptyset$
6: **for** $i = |\sigma|$ down to 1 **do**
7: $\langle z, r \rangle \leftarrow \sigma(i)$
8: **if** $|vertices(\mathcal{T}, z, r)| > 0$ **then**
9: $\sigma_{active}.pushback(\langle z, r \rangle)$
$\diamondsuit$ *continuous layer*
10: **for** several times **do**
11: $\langle z, r \rangle \leftarrow$ SELECTATRANDOM(σ_{active})
12: $v \leftarrow$ SELECTATRANDOM($vertices(\mathcal{T}, z, r)$)
13: $u \leftarrow$ SAMPLECONTROLINPUT()
14: **for** several times **do** $\diamondsuit$ *extend* $\mathcal{T}$ *from* v
15: $dt \leftarrow$ GETINTEGRATIONSTEP($state(v), u$)
16: $s_{new} \leftarrow$ INTEGRATEMOTIONEQS($f, state(v), u, dt$)
17: $r_{new} \leftarrow$ LOCATEREGION(s_{new})
18: $z_{new} \leftarrow \delta(z(v),$ DISCRETESTATE(s_{new}))
19: **if** COLLISION(s_{new}) = `true` **or** $(z_{new}) \in Reject$ **then**
20: break for loop of extend $\mathcal{T}$
21: $v_{new} \leftarrow$ ADDNEWVERTEX($\mathcal{T}, s_{new}, u, dt, r_{new}, z_{new}, v$)
22: **if** $z_{new} \in Accept$ **then return** $traj(\mathcal{T}, v_{new})$
23: **if** $\langle z_{new}, r_{new} \rangle \notin \sigma_{active}$ **then**
24: $\sigma_{active}.pushback(\langle z_{new}, r_{new} \rangle)$
25: $vertices(\mathcal{T}, z_{new}).pushback(v_{new})$
26: $vertices(\mathcal{T}, z_{new}, r_{new}).pushback(v_{new})$
27: $v \leftarrow v_{new}$

ton state z_{to} toward which to expand the search. The automaton state z_{from} and the workspace region r_{from} are selected from those already associated with the vertices in $\mathcal{T}$. The automaton state z_{to} is selected from those connected by a single automaton transition from z_{from}, i.e., $z_{to} \in \delta(z_{from})$. An additional criterion is that z_{to} should not be associated with the tree vertices, i.e., $\forall v_j \in \mathcal{T} : z(v_j) \neq z_{to}$, so that the search can be expanded toward new automaton states.

The discrete and the continuous layers work in tandem to effectively compute a collision-free and dynamically feasible trajectory that satisfies the LTL task specification ϕ. The search proceeds incrementally until a solution is obtained or an upper bound on computational time is exceeded. Each iteration consists of invoking the discrete layer to select $z_{from}, r_{from}, z_{to}$ and then compute a discrete plan, i.e., a sequence of automaton states and workspace regions that connects $\langle z_{from}, r_{from} \rangle$ to z_{to}. Sampling-based motion planning is invoked next which aims to expand $\mathcal{T}$ from vertices associated with $\langle z_{from}, r_{from} \rangle$ toward z_{to} while using the discrete plan as a guide. As the motion planner expands $\mathcal{T}$, new automaton states and workspace regions could be reached by the vertices and trajectories added to $\mathcal{T}$. As a result, the discrete layer could suggest to the continuous layer a different discrete plan for expanding $\mathcal{T}$ in the next iteration. This coupling of discrete planning in the discrete layer and sampling-based motion planning in the continuous layer, as evidenced by experimental results, allows the approach to efficiently compute a collision-free and dynamically-feasible motion trajectory that satisfies the LTL task specification. Pseudocode is given in Algo 1. More detailed descriptions of the main steps of the approach follow.

3.2 Discrete Layer

DISCRETETARGET($\mathcal{T}, \mathcal{A}, G$) selects an automaton state z_{from} and a region r_{from} from which to expand the search and an automaton state z_{to} toward which to expand the search (Algo 1:3). To facilitate the selection, tree vertices are grouped according to their associated automaton states, i.e.,

$$vertices(\mathcal{T}, z) = \{v : v \in \mathcal{T} \wedge z(v) = z\}.$$

These vertices are further grouped according to their associated regions, i.e.,

$$vertices(\mathcal{T}, z, r) = \{v : v \in \mathcal{T} \wedge z(v) = z \wedge region(v) = r\}.$$

Let Γ denote the set of all automaton states that are already associated with the vertices in $\mathcal{T}$, i.e.,

$$\Gamma = \{z : z \in Z \wedge |vertices(\mathcal{T}, z)| > 0\}.$$

The automaton state z_{from} is selected from Γ according to the probability distribution

$$prob(z) = 2^{1/d(z)} / \sum_{z' \in \Gamma} 2^{1/d(z')},$$

where $d(z)$ denotes the minimum number of automaton transitions to reach an accepting state in $\mathcal{A}$ from z. Since the overall objective is to effectively compute a trajectory that satisfies the LTL specification, the selection of z_{from} is biased toward automaton states that are close to accepting states.

The region r_{from} is selected uniformly at random from those regions associated with $vertices(\mathcal{T}, z_{from})$, i.e.,

$$\{r : r \in R \wedge |vertices(\mathcal{T}, z_{from}, r)| > 0\}.$$

The random selection is commonly advocated in motion-planning literature as it allows the sampling-based motion planner to expand $\mathcal{T}$ from different regions.

To expand the search toward new automaton states, z_{to} is selected among those automaton states not yet associated with the tree vertices, i.e., $z_{to} \notin \Gamma$. To ensure that the discrete plans are not too long, another criterion is that z_{to} should be a non-rejecting automaton state connected by a single transition from z_{from}, i.e., $z_{to} \in \delta(z_{from})$. Taking these criteria into account, z_{to} is selected uniformly at random from the set

$$\delta(z_{from}) - \Gamma.$$

In this way, the selection of $z_{from}, r_{from}, z_{to}$ aims to expand the search from tree vertices associated with automaton states that are close to accepting states and toward new automaton states.

DISCRETEPLAN$(\mathcal{A}, G, z_{from}, r_{from}, z_{to})$ computes a sequence of automaton states and regions that connects $\langle z_{from}, r_{from} \rangle$ to z_{to} (Algo 1:4). The discrete search is conducted over an abstract graph, which is obtained by implicitly combining the automaton $\mathcal{A}$ and the region graph $G = (R, E)$. More specifically, the edges coming out of a vertex $\langle z, r \rangle$ in the abstract graph are computed as

$$\text{EDGES}(\langle z, r \rangle) = \{\langle \delta(z, \text{DISCRETESTATE}(r')), r' \rangle : (r, r') \in E\}.$$

The cost of an edge $(\langle z', r' \rangle, \langle z'', r'' \rangle)$ is defined as the distance from region r' to r''. A vertex $\langle z, r \rangle$ in the abstract graph is considered as a goal vertex iff $z = z_{to}$.

The discrete plan σ from $\langle z_{from}, r_{from} \rangle$ to z_{to} is computed as the shortest path with probability p and as a random path with probability $1 - p$. While shortest paths provide greedy guides to the sampling-based motion planner, random paths allow for exploration of new regions, which provide alternative routes to prevent the sampling-based motion planner from getting stuck. A randomized version of depth-first-search, which visits the out-going vertices in a random order is used for the computation of random paths. Shortest paths are computed using Dijkstra's algorithm.

Note that sampling-based motion planning can expand $\mathcal{T}$ only from those $\langle z, r \rangle \in \sigma$ for which $|\textit{vertices}(\mathcal{T}, z, r)| > 0$. For this reason, σ is scanned backwards and $\langle z, r \rangle$ is added to $\sigma_{\textit{active}}$ if $|\textit{vertices}(\mathcal{T}, z, r)| > 0$. (Algo 1:6-9).

3.3 Continuous Layer

The objective of sampling-based motion planning is to expand $\mathcal{T}$ by adding several collision-free and dynamically-feasible trajectories using $\sigma_{\textit{active}}$ as a guide. Each trajectory is generated by first selecting $\langle z, r \rangle$ from $\sigma_{\textit{active}}$ uniformly at random (Algo 1:11). A vertex v is then selected uniformly at random from $\textit{vertices}(\mathcal{T}, z, r)$ (Algo 1:12) and a control input u is sampled uniformly at random (Algo 1:13). Note that other selection and sampling strategies are possible, as discussed in motion-planning books (Choset et al., 2005; LaValle, 2006). Random selections and sampling are commonly used in motion planning and are also shown to work well for the problems studied in this work.

A trajectory ζ is then obtained by integrating for several steps the motion dynamics of the robot when applying the control input u to $\textit{state}(v)$. To ensure accuracy, as advocated in the literature, this paper uses Runge-Kutta methods with an adaptive integration step (Algo 1:15-16).

Intermediate states s_{new} along ζ are added as new vertices to $\mathcal{T}$. Each new vertex v_{new} is associated with the corresponding region r_{new} and automaton state z_{new} (Algo 1:17-18). Recall that r_{new} is computed as the region that contains the position component of s_{new}. The automaton state z_{new} is computed as $\delta(z(v), \text{DISCRETESTATE}(s_{new}))$, where v is the parent of v_{new}. The integration of ζ stops if s_{new} is in collision or z_{new} is a rejecting automaton state (Algo 1:19-20). Otherwise, if z_{new} is an accepting automaton state, then $\textit{traj}(\mathcal{T}, v_{new})$ constitutes a collision-free and dynamically-feasible trajectory that satisfies the LTL task specification. In such case, the search terminates successfully.

The sampling-based motion planner may augment $\sigma_{\textit{active}}$. In fact, $\langle z_{new}, r_{new} \rangle$ is added to $\sigma_{\textit{active}}$ if not already there (Algo 1:23-24). Such additions enable the sampling-based motion planner to expand the search toward new automaton states and new regions.

4 Experiments and Results

Experimental validation is provided by using several scenes, LTL specifications, and a snake-like robot model with numerous DOFs, as described in Section 2.5. By increasing the number of links, the robot model provides challenging test cases for high-dimensional problems. In the experiments in this paper, the number of links is varied as $0, 5, 10, 15$ yielding problems with $5, 10, 15, 20$ DOFs. The running time for each problem instance is obtained as the average of thirty different runs. Experiments are run on an Intel Core 2 Duo machine (CPU: P8600 at 2.40GHz, RAM: 8GB) using Ubuntu 11.10. Code is compiled with GNU g++-4.6.1

Fig. 2 provides a summary of the results. Comparisons to related work (Bhatia et al., 2011; Plaku, Kavraki, and Vardi, 2009) show significant computational speedups. The speedups are more pronounced in the case of tasks 2 and 3. Recall that task 1 presents only one possible order in which to satisfy the propositions, while tasks 2 and 3 represent polynomially and exponentially many different possibilities. As a result, while the automaton for task 1 has linear size, the automata for tasks 2 and 3 have polynomial and exponential size, respectively. Since related work guides motion planning by using complete discrete plans, i.e., from $\langle z(v_{init}), r(v_{init}) \rangle$ to some $z \in \textit{Accept}$, as the size of the automaton increases, so does the length of the discrete plan and the computational cost to obtain such discrete plans. In distinction, the proposed approach uses short discrete plans from $\langle z_{from}, r_{from} \rangle$ to z_{to}, where z_{to} is connected by only one automaton transition from z_{from}. Moreover, while all the discrete plans in related work start from $\langle z(v_{init}), r(v_{init}) \rangle$, the proposed approach is biased to start discrete plans from automaton states that are close to accepting states.

As expected, the running time increases with the number of DOFs. As more and more links are added to the robot, it becomes increasingly difficult for the robot to move along the narrow passages of the workspaces and satisfy the LTL specifications. Nevertheless, the proposed approach is able to efficiently solve all problem instances, even as the number of DOFs is increased to 20, while related work struggles to find solutions.

In all the above experiments, the number of expansion iterations per discrete target is set to 200 (Algo. 1:10). Fig. 3 summarizes the results of the proposed approach when varying the number of expansion iterations. Significantly large numbers of iterations result in wasted computational time particularly when it is difficult to expand $\mathcal{T}$ to reach the discrete target. A small number of iterations may not give the sampling-based motion planner enough time to make progress in expanding $\mathcal{T}$ to reach the discrete target. As the results indicate, however, the approach works well for a wide

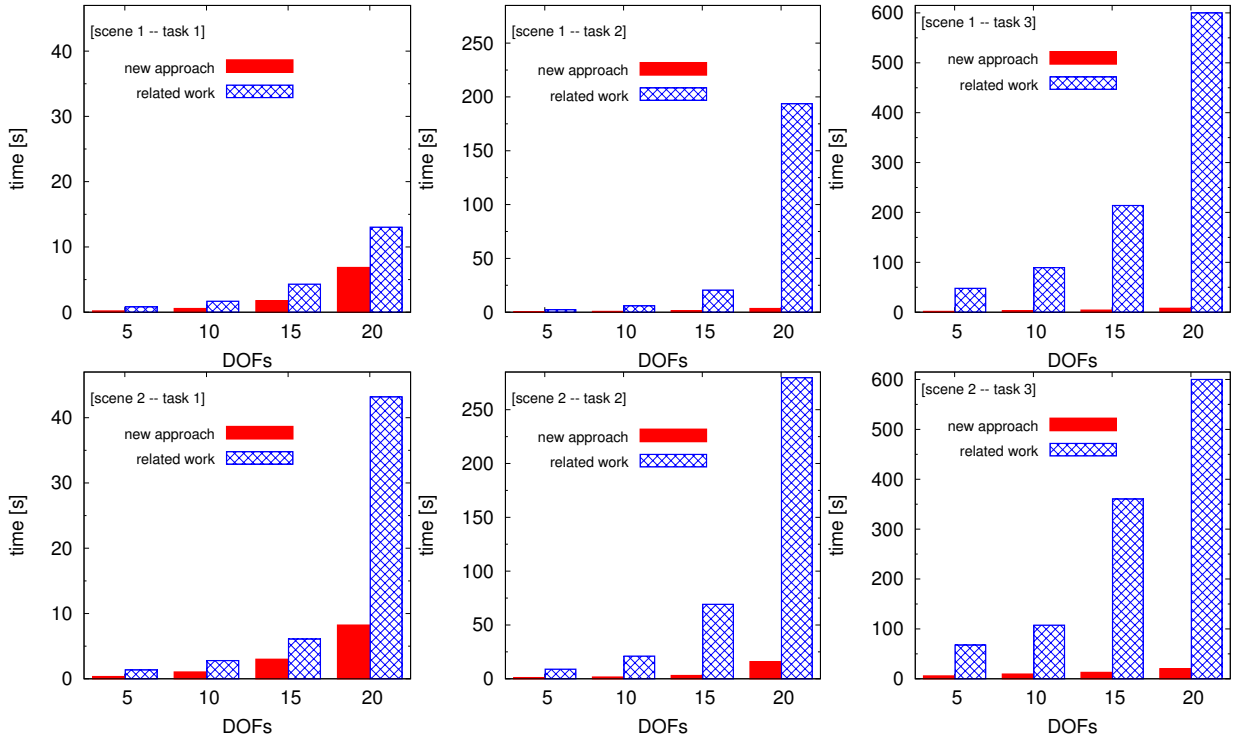


Figure 2: Comparison of the proposed LTL motion-planning approach to related work (Bhatia et al., 2011; Plaku, Kavraki, and Vardi, 2009). Results are shown over the two scenes (Fig. 1) and the three LTL specifications (Section 2.5) when varying the number of DOFs of the snake-like robot model from 5 to 20.

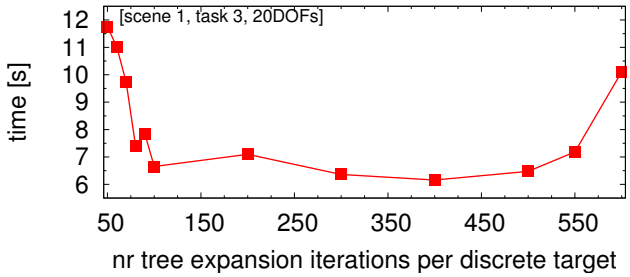


Figure 3: Impact of the number of expansion iterations per discrete target (Algo. 1:10) on the computational efficiency of the approach.

range of values. Similar trends to Fig. 3 which shows the result for scene 1, task 3, and the robot model with 20 DOFs, were observed for the other scenes, tasks, and robot DOFs, but are not shown here due to space constraints.

As in related work, the focus of this paper was on improving the computational time to solve LTL motion-planning problems. The solution trajectories, even though not optimized, are generally short. The table below summarizes the average path length computed as the distance traveled by the reference point of the head link of the snake-like robot. These results are presented as ratios over the lower bound on the solution obtained by the shortest-collision free path for a point robot with no differential constraints.

[scene 1, task 1]

DOFs	5	10	15	20
length ratio	1.11	1.22	1.31	1.34

As the results indicate, the solutions produced by the proposed approach come close to the lower bound. Similar results were obtained for the other scenes and tasks but are not shown here due to space constraints.

5 Discussion

Toward the goal of bridging the gap between the discrete and the continuous, this paper showed how to effectively compute collision-free and dynamically-feasible motion trajectories that satisfy task specifications given by LTL. LTL makes it possible to express sophisticated tasks in terms of propositions, logical connectives, and temporal connectives. The approach combined sampling-based motion planning over the continuous state space with discrete search over both the LTL task representation and a workspace region graph. The discrete search guides the sampling-based motion planner to expand the search from tree vertices associated with automaton states that are close to accepting states and toward new automaton states. In distinction from related work, the proposed approach samples the discrete space to shorten the length of the discrete plans and to more effectively guide motion planning in the continuous state space. Experimental results show significant computational speedups over related work.

Even though the focus of this paper, as in related work,

was on improving the computational time to solve LTL motion-planning problems, the proposed approach generally produces short trajectories. The quality of the trajectories can be further improved by using common postprocessing techniques (Choset et al., 2005; LaValle, 2006). In addition, A* criteria could be incorporated into the approach to select automaton states and workspace regions that promote the generation of shorter trajectories. Recent results in sampling-based motion planning (Karaman and Frazzoli, 2011), which show optimality in the case of point-to-point planning, could provide a basis on how to obtain optimality even when considering LTL motion-planning tasks.

References

Arkin, R. C. 1990. Integrating behavioral, perceptual and world knowledge in reactive navigation. *Robotics and Autonomous Systems* 6:105–122.

Armoni, R.; Egorov, S.; Fraer, R.; Korchemny, D.; and Vardi, M. 2005. Efficient LTL compilation for SAT-based model checking. In *Intl Conf on Computer-Aided Design*, 877–884.

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

Cambon, S.; Alami, R.; and Gravot, F. 2009. A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research* (1):104–126.

Choset, H.; Lynch, K. M.; Hutchinson, S.; Kantor, G.; Burgard, W.; Kavraki, L. E.; and Thrun, S. 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.

Ding, X. C.; Kloetzer, M.; Chen, Y.; and Belta, C. 2011. Formal methods for automatic deployment of robotic teams. *IEEE Robotics and Automation Magazine* 18(3):75–86.

Fainekos, G. E.; Girard, A.; Kress-Gazit, H.; and Pappas, G. J. 2009. Temporal logic motion planning for dynamic mobile robots. *Automatica* 45(2):343–352.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Morgan Kaufmann.

Hauser, K., and Ng-Thow-Hing, V. 2011. Randomized multi-modal motion planning for a humanoid robot manipulation task. *International Journal of Robotics Research* 30(6):678–698.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research* 30(7):846–894.

Kress-Gazit, H.; Conner, D. C.; Choset, H.; Rizzi, A.; and Pappas, G. J. 2007. Courteous cars: Decentralized multi-agent traffic coordination. *Special Issue of the IEEE Robotics and Automation Magazine on Multi-Agent Robotics*.

Kress-Gazit, H.; Wongpiromsarn, T.; ; and Topcu, U. 2011. Correct, reactive robot control from abstraction and temporal logic specifications. *IEEE Robotics and Automation Magazine on Formal Methods for Robotics and Automation* 18(3):65–74.

Kupferman, O., and Vardi, M. 2001. Model checking of safety properties. *Formal methods in System Design* 19(3):291–314.

Laumond, J. 1993. Controllability of a multibody mobile robot. *IEEE Transactions on Robotics and Automation* 9(6):755–763.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge, MA: Cambridge University Press.

Nieuwenhuisen, D.; van der Stappen, A.; and Overmars, M. 2006. An effective framework for path planning amidst movable obstacles. In *International Workshop on Algorithmic Foundations of Robotics*, volume 47 of *Springer Tracts in Advanced Robotics*. 87–102.

Payton, D. W.; Rosenblatt, J. K.; and Keirsey, D. M. 1990. Plan guided reaction. *IEEE Trans. on Systems, Man, and Cybernetics* 20(6):1370–1372.

Plaku, E., and Hager, G. D. 2010. Sampling-based motion and symbolic action planning with geometric and differential constraints. In *IEEE International Conference on Robotics and Automation*, 5002–5008.

Plaku, E.; Kavraki, L. E.; and Vardi, M. Y. 2009. Falsification of LTL safety properties in hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*. York, UK: Springer. 368–382.

Plaku, E.; Kavraki, L. E.; and Vardi, M. Y. 2012. Falsification of LTL safety properties in hybrid systems. *International Journal on Software Tools and Technology Transfer*. invited, in print.

Saffiotti, A.; Konolige, K.; and Ruspini, E. H. 1995. A multivalued logic approach to integrating planning and control. *Artificial Intelligence* 76(1-2):481–526.

Shewchuk, J. R. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications* 22(1-3):21–74.

Sistla, A. 1994. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing* 6:495–511.

Stilman, M., and Kuffner, J. 2008. Planning among movable obstacles with artificial constraints. *International Journal of Robotics Research* 27(12):1295–1307.

Stilman, M. 2010. Global manipulation planning in robot joint space with task constraints. *IEEE Transactions on Robotics* 26(3):576–584.

Wolfe, J.; Marthi, B.; and Russell, S. 2010. Combined task and motion planning for mobile manipulation. In *Intl. Conf. on Automated Planning and Scheduling*, 254–258.

From Low-Level Trajectory Demonstrations to Symbolic Actions for Planning

Nichola Abdo and **Henrik Kretzschmar** and **Cyrill Stachniss**
University of Freiburg
Department of Computer Science
Georges-Köhler-Allee 079
79110 Freiburg, Germany

Abstract

Robots that should solve complex manipulation tasks need to reason about their actions on a symbolic level to compute plans comprising sequences of actions. Planning, however, requires knowledge about the preconditions and effects of all the actions. In this work, we present an approach that allows a robot to learn manipulation skills from teacher demonstrations. Our approach enables the robot to learn to physically execute the motion needed to perform the actions, and, most importantly, to infer the preconditions and effects. Our system can express the acquired manipulation action as symbolic planning operators and thus can use any modern planner to solve tasks that are more complex than the individual, demonstrated actions. We implemented our approach on a PR2 robot and present real world manipulation experiments that illustrate that our system allows non-experts to transfer knowledge to robots.

Introduction

Future service robots must be flexible enough to carry out a variety of day-to-day tasks under diverse conditions. It is, however, practically impossible to preprogram a robot for all kinds of situations that occur in the real world. Therefore, we need means for easily instructing robots and teaching them new skills by non-experts.

Planning for solving complex manipulation tasks can be done using low-level motor commands or on a symbolic level. Computing solutions based on low-level motor commands is infeasible due to the high-dimensionality of the resulting planning problem and thus robots need to reason about their actions on a higher level. Computing plans of high-level actions to achieve some goal, however, requires a high-level symbolic representation describing the preconditions and effects of the robot's actions.

In this work, we aim for a fast and intuitive learning process that allows the robot to learn new actions such that it can later on reason about the actions on both, the motion level and a symbolic level. Our approach is based on learning by demonstration. The robot observes a human teacher and learns how to physically execute the movements in order to solve a manipulation task. In addition to that, the robot learns the preconditions and effects of the actions, which are both needed for planning. While using the learned actions to solve manipulation tasks, the robot monitors its performance and reacts to unexpected changes. In summary, our system (i) encodes the low-level movements, (ii) estimates the preconditions and effects of the individual actions, and (iii) generates a planning problem definition that allows state-of-the-art planning systems to solve new tasks using the learned actions. We implemented our approach and carried out extensive experiments using a PR2 robot to illustrate the capabilities and flexibility of our system. We demonstrate that our approach enables the robot to autonomously solve tasks that are more complex than the basic actions that have been demonstrated to the robot.

Related Work

In the literature, there are various approaches for transferring task knowledge from humans to robots. In recent years, imitation learning methods have become popular to encode robot motion. See Billard et al. (2008) for an overview. The key idea is to speed up the learning process by exploiting demonstrations given by a teacher. For example, Bentivegna et al. (2004) demonstrate to a humanoid robot how to play air hockey by learning primitives that the robot can use in new situations. The robot learns how to choose a primitive in a given situation and practices these primitives to improve its performance. Asfour et al. (2006) use hidden Markov models to encode and reproduce demonstrated actions. Dynamic movement primitives (DMPs) are popular to learn control policies for robotic manipulators from demonstrations and to generalize the movements to new situations. Our approach also relies on these movement primitives as proposed by Pastor et al. (2009) to encode the low-level movements of the actions. Calinon and Billard (2008) propose Gaussian mixture models to represent the variance over time in the demonstrated trajectories of a manipulator to exploit this information in the reproduction step. Also Eppner et al. (2009) consider the variance to guess less relevant parts of the demonstrations. Our method analyzes the variations in the state during the demonstrations to identify the preconditions and effects of the individual actions. This allows our approach to generate a symbolic representation of the actions, which is then used for planning purposes.

There are also a number of approaches that aim at teaching robots skills on a symbolic level for task planning based on teacher demonstrations. Veeraraghavan and Veloso (2008)

demonstrate sequences of actions to teach a robot a plan for solving sequential tasks that involve repetitions. They instantiate preprogrammed behaviors and then learn the corresponding preconditions and effects. Pardowitz, Zöllner, and Dillmann (2006) extract task-specific knowledge from sequences of actions. The robot extracts the relevant elementary actions and task constraints from teacher demonstrations of pick-and-place actions when setting a table. Manipulation skills are arranged in a hierarchical manner with macro actions encompassing elementary ones. The preconditions and effects of actions are expressed by predetermined properties like the relative positions of the objects. Ekvall and Kragic (2006) also provide a robot with demonstrations of tasks related to setting a table. The robot incrementally learns the constraints for each task with respect to the order of executing the actions. This knowledge is then used to choose the best strategy for solving a new task. To identify the different states observed during the demonstrations, they apply k-means clustering to the relative positions and orientations of the objects and inspect the cluster variances. Zhuo et al. (2009) learn action preconditions and effects for hierarchical task networks from given observed decomposition trees. Such trees describe how a task can be broken down into smaller subtasks.

Similar to Ekvall and Kragic, our system applies k-means to features to identify preconditions and effects of actions. By inspecting the variance within the extracted clusters, our system additionally tries to determine if a certain feature or aspect of the action is relevant as a precondition or effect and to recognize similar states across different actions. Unlike the approaches above, we do not require to demonstrate sequences of actions to the robot or to provide task decomposition information. Instead, our system learns the individual actions by demonstration, and uses the identified conditions to chain the actions in plans for solving a variety of tasks.

Many researchers adopt object action complexes (OACs), as presented by Krüger et al. (2009), as a representation that combines low-level robot control and high-level planning. OACs consider objects as important to the robot in terms of the actions that can be applied to them. Pastor et al. (2009) suggested adding a symbolic meaning to DMPs such that they can be used for high level planning in the context of OACs. However, this was not realized in their work. There are approaches that learn simple cause-effect rules based on simulated actions or exploration so that they can be integrated into the OAC frameworks (Petrick et al. 2008). Furthermore, Omrcen, Ude, and Kos (2008) present an approach that allows a robot to learn the effect of poking different objects by exploration. The approach uses a neural network to learn the relation between pushing actions and the predicted motion of the object. This is then used to plan for applying several poking actions to move objects to desired locations.

In contrast to these techniques, our approach does not rely on exploration or simulation to learn the effects of carrying out actions. Instead, our system learns both the preconditions and effects of the actions from a few teacher demonstrations and represents the actions as high-level planning operators. From the same demonstrations, our approach learns the trajectories of the manipulator using dynamic movement primitives.

Overview

Our approach allows a robot to acquire and combine actions to solve complex tasks. By observing a teacher, the robot learns how to physically execute individual actions. The robot then infers symbolic information that allows it to combine the learned actions via a planning system to solve complex tasks that have not been shown to it.

Our work enables a robot to identify preconditions that have to be satisfied to carry out a certain action as well as the effects of an action. An example of such a precondition is the fact that the gripper of the robot needs to be open before an object can be grasped. Identifying preconditions and effects is done by estimating the distribution of world states to identify patterns while the teacher repeatedly demonstrates the same action. These patterns lead to logical predicates, allowing the robot to translate low-level sensory data into a high-level logical representation and verify when the preconditions or effects are satisfied. Consequently, our robot can associate the low-level movements of its end effector with symbolic information and to derive a definition of the action in the Planning Domain Definition Language PDDL. Given the PDDL description, the robot is able to use any modern planning system to solve tasks that are more complex than the individual actions that have been demonstrated to it.

Perception and Predefined Features

Our approach assumes that the robot can identify relevant objects in the scene along with their poses. For this work, we attached checkerboard markers to the relevant objects and used an out-of-the-box detector that is available in the robot operating system ROS. The detector provides the robot with the types of the objects (e.g. block, table, . . .) and their poses. The robot also uses its laser scanner, for example to estimate the state of doors or for 2D obstacle avoidance. Note that our approach is orthogonal to the perception problem. Therefore, our method should be applicable in the same way when using a system for marker-free detection of objects.

To encode the state of the world, our approach relies on features, which can take continuous or discrete values. We derive the preconditions and effects of the different actions using the values of these features during the demonstrations. So far, we applied our system to solve blocks world-like tasks and to operate doors. We defined continuous features such as the opening of the gripper, the relative poses between the gripper and manipulated objects, and the relative poses between objects. We furthermore defined discrete features such as the visibility of objects, and the state of the door. Depending on what the user demonstrates, new features may need to be defined. This can be done easily and does not affect already learned actions.

Recording and Encoding Demonstrations

To teach the robot basic actions, we use kinesthetic training, *i.e.*, the teacher moves the manipulators of the robot, as illustrated in Fig. 1. This method is rather accurate and does not require extra sensors since the robot can directly record the movements using its own encoders. Our approach allows

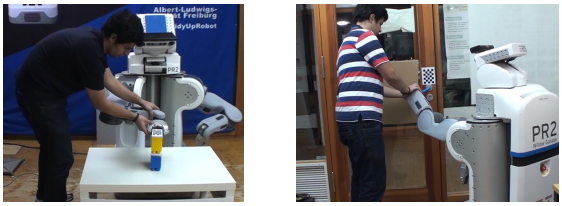


Figure 1: Examples of kinesthetic training showing how to place a block on another one and how to operate a handle.

for demonstrating individual actions one by one and does not require demonstrating sequences of actions.

A popular way to encode movements of a manipulator are DMPs. Our approach uses DMPs as described by Pastor et al. (2009) to encode the trajectory of the robot's end effector as observed in the demonstrations. DMPs allow us to easily adapt the movement to different situations such as new starting or goal points. Our system groups the learned DMPs together so that multiple DMPs for each action are available to the robot. In our experiments, we recorded 10 demonstrations per action. Moreover, we propose in the next section a method for extracting the preconditions and effects from these demonstrations.

Identifying Preconditions and Effects

The preconditions of actions and their effects are expressed in terms of features. To identify the preconditions and effects based on a set of demonstrations, we inspect the recorded values of all features at the beginning and at the end of each demonstration. For each feature, we then seek to find patterns in its values to decide whether or not it is important for an action.

General Problem and Assumptions

In the most general case, the robot cannot be sure that an action can be carried out unless the current state of the world is identical to a state observed in one of the demonstrations. Otherwise, a precondition might not be satisfied and executing the action might fail. Finding the preconditions only based on successful demonstrations does not lead to satisfying results without further assumptions. The resulting unsupervised learning problem can be viewed as a one-class classification problem in which only positive examples are provided. In our case, the examples correspond to demonstrations in which the preconditions and the effects are satisfied.

Such problems can be addressed using density estimation methods or by only estimating the boundaries of the distribution (Schölkopf et al. 2000). A simple nearest neighbor approach considers all states to be fulfilling the conditions that are similar under a distance function to the states observed during the demonstrations. In our setting, a key disadvantage of the nearest neighbor approach is the fact that a large subset of the features are not relevant as preconditions and effects of most actions. As a result, the entire feature space would have to be populated by samples to make the robot ignore irrelevant feature values. This is infeasible in practice since only a few demonstrations can be provided by a teacher. In contrast to the nearest neighbor approach, one-class classification methods such as single-class support vector machines (SVMs) could be more appropriate approaches in this setting (Schölkopf and Smola 2002).

To tackle the above mentioned problem, we assume that the preconditions and effects of the actions can be expressed in terms of the predefined features and their corresponding values, and that the individual features are independent of each other. We consider that a feature is relevant as a precondition or an effect of an action if its values follow certain patterns throughout the demonstrations. Moreover, we assume that the teacher provides demonstrations with variations. If the teacher demonstrates actions with too few variations, the robot may identify additional preconditions or effects that are irrelevant in reality.

Estimating Preconditions and Effects by Analyzing the Variations in the Demonstrations

In our approach, three questions have to be answered: First, which features are relevant for an action as a precondition or as an effect? Second, if a feature is regarded as relevant, which values are typically observed and how to derive a logical predicate that encodes the decision whether the precondition or effect is fulfilled? Third, given two logical predicates, how to decide whether both represent the same condition? The last question is important to allow a planning system to verify beforehand whether the effects of an action match the preconditions of another one. This is essential for planning.

As preconditions, we consider features that take the same or similar values at the beginning of all demonstrations of an action. The same holds for the effects: features that always take similar values after having executed an action are considered to be an effect of that action. Informally speaking, for each action independently, we estimate for each feature the region of the feature space that covers the samples corresponding to the demonstrations. By analyzing the volumes of such regions, we can decide whether the feature is relevant or not. This allows us to derive the logical predicates and to estimate whether two predicates model the same condition or not.

Note that for an action, we only consider those features that involve objects or things that are close to the robot during the demonstrations or that involve only the robot itself. This allows us, for example, to ignore the state of a window, potentially in a different room, while the teacher shows the robot how to operate the door.

Features Taking Continuous Values There exist multiple ways for estimating the boundaries of regions in a potentially high-dimensional space that are populated by samples. Approaches to one-class classification belong to this class of algorithms such as single-class SVMs (Schölkopf et al. 2000). Alternative approaches are the one-class k-means, the one-class PCA, and the one-class k-nearest neighbor (Kennedy, Namee, and Delany 2009).

Compared to most other learning problems, we suffer from having only a small number of training examples. A user is expected to provide around ten demonstrations of one action. This will lead to ten sample points in feature space. With so

few training examples, applying techniques such as SVMs is likely to provide results that do not generalize well. For example, in the context of image classification based on a small number of training images, simpler methods such as nearest neighbor approaches are reported to perform better than SVMs (Boiman, Shechtman, and Irani 2008). Since we consider training sets in the order of 10 sample points, we propose to not use single-class SVMs but follow a simpler approach and apply one-class k-means. Note that one-class k-means does not mean that $k = 1$ but that all centroids represent the single class jointly, which allows for covering multiple modes. Additionally, we consider the variance of the samples within each of the identified clusters.

If all samples are concentrated in one cluster, we can directly compute the variance in the individual feature values over multiple demonstrations. If the variance is small, we can regard the feature as relevant and thus to be a precondition or an effect. There are, however, situations in which such a simple criterion is not successful. For example, before grasping a block, the gripper must be open and the gripper must either be on top of the block or at its side (top grasp or side grasp). For the robot, it can be advantageous to consider this as two distinct actions, but the teacher may teach that as one grasping action. To allow for considering such situations in which feature values can be centered around multiple possible values, we apply k-means clustering to the individual feature values and then analyze the variances in each cluster.

Since the teacher demonstrates an unknown number of ways of performing an action, the system typically does not know the number of clusters k to look for in advance. We therefore perform multiple iterations of k-means with increasing values for k from 1 up to $\sqrt{N/2}$, where N is the number of data points. Note that this upper limit is a heuristic, as suggested by Mardia, Kent, and Bibby (1979).

For a cluster to be considered as representing an important aspect of the action, its average squared intra-cluster distances should not exceed a certain limit. This predefined limit reflects the desired accuracy of executing the manipulation action. For a cluster c with mean μ_c, this condition can be expressed as

$$\frac{1}{N_c}\sum_{i=1}^{N_c} dist(v_i^c, \mu_c)^2 \leq \varepsilon_1, \tag{1}$$

where N_c is the number of data points assigned to cluster c and v_i^c is the value of the i^{th} data point in the cluster. Here, $dist(.,.)$ is a distance measure for the feature under consideration. This can either be the Euclidean distance or the angular difference based on an angle/axis representation in case of a rotation, *i.e.*,

$$dist_{rot}(R_v, R_\mu) = angleOf(R_v R_\mu^{-1}), \tag{2}$$

where R_v and R_μ are the corresponding rotation matrices. The value ε_1 is the maximum allowed variance for each cluster (that is separately defined for the Euclidean and the angular distance function). If the condition in Eq. (1) is satisfied for all clusters, our system could identify a potentially multi-modal pattern in the input data and considers this pattern as a precondition or effect. Then, no further increase of k is needed. However, if this criterion fails for all values of k, the system determines that the feature is irrelevant to the action since no pattern could be found.

To finally make the decision if a state satisfies a precondition or an effect, we have to check, according to the one-class k-means formulation, whether the minimum distance between the centroids and the current feature value v is smaller than a threshold or not. We represent this fact by so-called predicates that are used by the planning system. A predicate $\mathcal{P}_f$ is defined for each action for which the feature f is relevant as a precondition or effect. We may generate an individual predicate for the precondition and effect as well as for each cluster c. The predicate is defined as:

$$\mathcal{P}_{f,c}(v) = \begin{cases} true & \text{if } dist(v, \mu_c) \leq d_{max} \\ false & \text{otherwise,} \end{cases} \tag{3}$$

where d_{max} is a threshold defining the maximum allowed distance to the centroid.

Features Taking Discrete Values Besides features taking continuous values, we also consider features taking discrete values. An example of such a feature is object-is-visible, which can be $true$ or $false$. To decide whether a discrete-valued feature is relevant for an action, we compute the entropy of the distribution of the feature values during the demonstrations. The entropy H is a measure of uncertainty and is defined as

$$H(f) = -\sum_{l=1}^{L} P(f = v_l)\log_2 P(f = v_l), \tag{4}$$

where $P(f = v_l)$ is the probability that the feature f takes the value v_l (out of L possible values). The distribution over the values is computed based on the observations. A low entropy indicates that the probability mass is concentrated in one state (or a few states, depending on the number of possible states) and thus indicates a low variation of the feature value over the demonstrations. To avoid overfitting in case of few demonstrations, a Dirichlet prior can be added.

In our approach, we consider a feature f as relevant if $H(f) < \varepsilon_2$, where ε_2 is a threshold specifying the certainty the system should have about the value of this feature. The value that the feature has to take to satisfy the precondition or effect is then given by

$$v^f = \underset{v_l \in \{v_1 \ldots v_L\}}{\operatorname{argmax}} P(f = v_l). \tag{5}$$

In most cases, the discrete features are binary variables taking $true$ and $false$ as possible values, but there exist also features that can take more than two values. An example of a feature that we found useful in our experiments is a three-state representation of a door: the door can be completely open so that the robot can go through it, or it can be partially open so that the robot first needs to open it further to pass through without having to operate the handle, or the door may be closed completely.

Similar to the continuous case, we can derive a boolean predicate $\mathcal{P}_f(v)$ that is later on used in the planning process

to test whether a discrete precondition is fulfilled as:

$$\mathcal{P}_f(v) = \begin{cases} true & \text{if } v = v^f \text{ and } H(f) < \varepsilon_2 \\ false & \text{otherwise.} \end{cases} \quad (6)$$

Identifying Identical Predicates

Whenever the user teaches actions individually and not as a sequence, the predicates have to be learned for each action individually. To allow a planner to compute a plan, we need to identify which predicates from one action are the same as the predicates from other actions. Consider two predicates $\mathcal{P}_f^{a_1}$ and $\mathcal{P}_f^{a_2}$ generated from two different actions a_1 and a_2 but from the same feature f. To decide if they represent the same condition, we consider the feature values from the demonstrations of a_1 and a_2 as a merged sample set. The predicates $\mathcal{P}_f^{a_1}$ and $\mathcal{P}_f^{a_2}$ will be merged into one predicate if the merged sample set still fulfills the criterion given in Eq. (1) (or the entropy criterion for the discrete case). Otherwise, $\mathcal{P}_f^{a_1}$ and $\mathcal{P}_f^{a_2}$ remain individual predicates.

Generating the PDDL Description

Over the last 15 years, the Planning Domain Definition Language PDDL has been established as a standard language for defining planning problems. Therefore, we developed a system that automatically derives a PDDL description which allows us to easily use most out-of-the-box planning components.

To generate a PDDL description, we first need to define the objects involved in the planning process and their types. This is obtained from the perception system as mentioned before. Second, we need the predicates that define the state of the planner, and which are computed using the method described above. Third, the start and goal states need to be specified. The start state is simply obtained by evaluating all predicates according to the current observations. The goal, obviously, has to be provided by the user in terms of the predicates. Finally, the actions with their preconditions and effects on the state have to be provided.

For expressing each action in terms of its preconditions and effects, we consider the different possible cases for each relevant feature f. Since f could be relevant as a precondition, an effect, or both, our system adds the appropriate predicate, $\mathcal{P}_f$, or its negation in the preconditions or effects part of the PDDL operator. An example of a generated PDDL operator for approaching a block from the top to grasp it is:

```
(:action reachBlockTop
 :parameters (?b-block ?g-gripper)
 :precondition (and (visible ?b)
         (gripperOpen ?g)
 :effect (and (not (visible ?b))
         (gripperAroundBlockTop ?g ?b)))
```

The operator has been learned from demonstrating the reaching motion to the robot. The parameter block represents the typed variables `?b` and `?g` that are involved in the predicates. The types of objects are not learned but are provided by the perception system during the demonstrations. The terms in the precondition and effect blocks correspond to learned predicates. Here, we replaced the automatically generated names by meaningful ones.

Accounting for Physical Constraints

After implementing our approach, we identified that the robot misses background knowledge about its capabilities and the physical world. For example, the robot should not move away from a door if its gripper is still grasping the handle of the door. The robot simply cannot move the door although that might be a valid plan from the PDDL definition point of view. Such constraints could in theory be identified based on a physical simulation system that operates in parallel to the planner and verifies that a plan does not violate any physical constraints. However, such simulations are considerably expensive and complex. We therefore added a few additional constraints manually to the PDDL description. In particular: (i) The robot cannot move away from a door while grasping its handle. A similar rule needs to be added for any object that the teacher grasped during the demonstrations but that cannot be carried away. (ii) The robot is not allowed to release an object from its gripper without placing it somewhere, for example on a table. Otherwise, the object may break or the robot may not be able to pick it up again from the ground—this actually happened during our first experiments. (iii) The robot cannot reach any object that is further away than 70 cm from its torso without navigating first. This encodes the size of the workspace which is given by the size of the robot's arm.

Planning using the Acquired Actions

Given the collection of actions including the PDDL description, the planning problem can be outsourced to any standard symbolic planning system capable of interpreting PDDL. In our system, we use the fast downward planner proposed by (Helmert 2006). We used Helmert's implementation and integrated it into a ROS module.

To execute the next action of a computed plan, the robot has to choose one of the DMPs from its library that belongs to the corresponding action. The DMPs can be adjusted easily to situations that have a different starting or goal point compared to the learning phase. The DMP will generate a new trajectory whose shape resembles the demonstration but generalizes to the new situation. To only enforce minor adaptations due to a new start and goal point, our approach selects the DMP for which the relative pose of the end effector between the goal point g and start point s during training was most similar to the current situation. The relative pose is described by its translational $t(s,g)$ and rotational component $r(s,g)$. We select a DMP using the cosine similarity measure

$$d_t(s,g,i) = \frac{t(s,g) \cdot t(s_i,g_i)}{||t(s,g)||\ ||t(s_i,g_i)||}, \quad (7)$$

where s_i and g_i are the start and goal pose during the demonstration from which the i-th DMP has been learned. Similarly, $d_r(s,g,i)$ is defined for the rotational component and takes into consideration the angle and direction of rotation.

Additionally, we simulate the trajectory for the i-th DMP after setting the new start and goal to check for collisions between the end effector and obstacles. The term $d_o(s,g,i)$ is the minimum distance to the closest obstacle along the

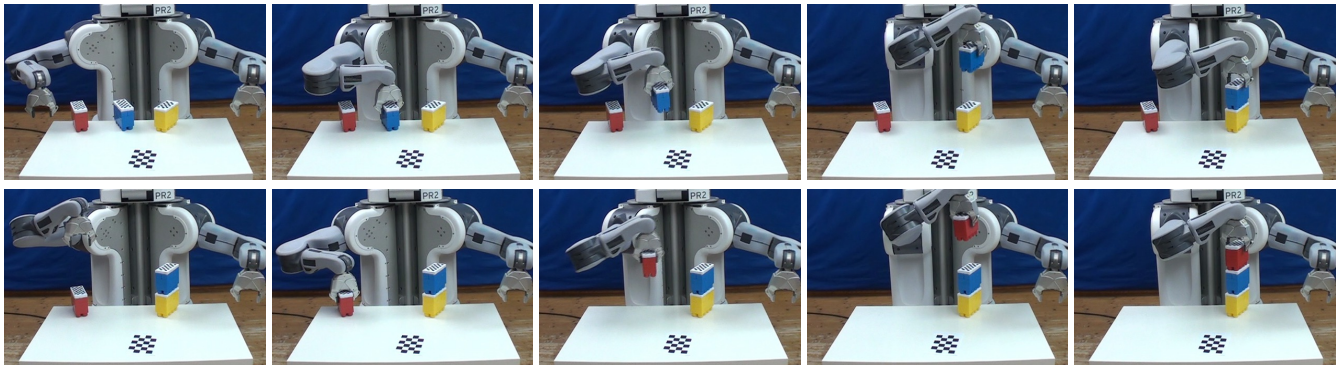


Figure 2: The robot builds a tower of three blocks. To do so, the robot only uses the basic actions that it has learned from demonstrations and combines them in a new way.

trajectory. Then, we choose the DMP with the index

$$i^* = \underset{i}{\operatorname{argmax}}\, \alpha_t d_t(s,g,i) + \alpha_r d_r(s,g,i) + \alpha_o d_o(s,g,i), \quad (8)$$

where α_t, α_r, and α_o are scaling coefficients chosen to reflect the relative importance of the different criteria. Finally, the chosen DMP is instantiated with s and g and executed.

Note that even when properly selecting appropriate DMPs, in the real world a robot may not carry out all actions as expected or the environment may change. Especially for long action sequences, it is unlikely that the individual steps can be executed without corrections. For example, if the gripper slips off the door handle, the robot should be able to detect that and compute a new plan. We therefore implemented a separate module that monitors the execution of the plan, computes the current values of the features, updates the values of the individual predicates, and compares the actual state to the expected effects of the actions. In case something unforeseen happens, the execution monitor triggers replanning using the current state of the world as the start state. The fast downward planner is efficient enough to compute a new plan online so that the robot can proceed without significant interruptions.

Experiments

The evaluation is intended to show the capabilities of our approach. All components of the system have been implemented as ROS modules and our experiments are carried out with a real PR2 robot. We considered tasks from two different manipulation domains: blocks world-like tasks like moving and stacking blocks as well as operating and opening doors. Videos covering the experiments can be found at: http://www.informatik.uni-freiburg.de/%7Estachnis/videos/tampra/

Training and Learning Preconditions and Effects

The first set of experiments is designed to illustrate how our system can learn individual actions from multiple demonstrations and is able to identify the preconditions and effects reliably. Teaching was done by kinesthetic training as illustrated in Fig. 1. We demonstrated 11 different actions to the robot and provided 10 demonstrations per action. Actions include reaching for objects and grasping them, placing an object on a target, turning a door handle, pushing a door, etc. In all our experiments, the DMPs were learned without any problems and stored in the robot's action library. Moreover, the correct set of preconditions and effects was identified by our system, *i.e.*, no necessary conditions were missing and all relevant ones were identified.

Table 1: Success rate for learning preconditions and effects.

#demonstrations	5	6	9	10	>10
success rate	17/20	19/20	19/20	20/20	20/20

For example, for the action `reachHandle`, the system correctly identified as preconditions that (a) the gripper has to be open and that (b) the handle must be visible. As effects, it identified that (a) the door handle is inside the gripper, (b) the gripper stays open, and (c) the handle is still visible. At the same time, the system correctly identified that all other features, like the relative pose of the gripper to the robot's torso and the exact distance of the door handle relative to the robot, are irrelevant.

To provide a more quantitative evaluation, we recorded 20 demonstrations for the action `reachBlock`. We then randomly sampled demonstrations, performed the learning step, and compared the extracted preconditions and effects to the real ones. We repeated this process 20 times and obtained the results shown in Tab. 1. When using 10 or more demonstrations, the system produced the correct results in all cases. With less than 10 demonstrations, the system often failed 1 to 3 out of 20 times in the sense that our approach found at least one false positive precondition or effect. This is due to too little variations in the feature values in the small number of demonstrations.

Building a Tower of Three Blocks

The second set of experiments is designed to show how the robot can use the learned actions to solve novel tasks, *i.e.*, tasks that have not been demonstrated to it beforehand. In this example, the robot was placed in front of a table with three

Figure 3: The robot computes a plan to grasp the two blocks and go through an open door. Once the robot has grasped the two blocks, a person closes the door. After having detected that, the robot computes a new plan to clear its left gripper by first going back to the table and placing the yellow block there. The robot then moves back to open the door, and then back to the table to grasp the yellow block again. Finally, the robot leaves the room with both blocks and reaches the goal state. See also http://www.informatik.uni-freiburg.de/%7Estachnis/videos/tampra/ for a video of this experiment.

blocks on top of it. As the goal configuration, the three blocks should be stacked on top of each other (yellow-blue-red). The planner computed a plan using the learned pick-and-place operators. This involves reaching, grasping, placing, and releasing blocks. Furthermore, the system correctly merged identical predicates. For example, the effect of grasping is equivalent to the precondition of placing. Moreover, each step was executed by choosing a DMP and adapting it to the new situation. Fig. 2 depicts the plan execution.

We repeated this experiment 20 times. In all cases, the robot was able to generate a valid plan to the goal. The only sources of failure that occurred during the execution were checkerboard markers not being detected by the perception system, or an error in estimating the pose of a block.

Reacting to Unexpected Changes in the Environment

This experiment is designed to illustrate how the robot can deal with unexpected changes in the environment while executing plans. The goal was to take two blocks and bring them to the corridor outside the room. Initially, both blocks lie on a table in the room and the door is open. While the robot picks up the two blocks with its manipulators, a person closes the door. After detecting that change, the robot computes a new plan and decides to bring one block back to the table to free one gripper. It then opens the door with the free hand, moves back to the table, picks up the block again, and finally brings both blocks outside the room. Pictures from this experiment are shown in Fig. 3.

Maintaining a Goal State

The last experiment illustrates how the robot can use the learned actions to plan for reaching a goal state from different possible starting states. We placed the robot in front of a door and instructed it to keep it fully open. Then, a human repeatedly closed the door and the robot opened it from basically any possible configuration.

During this experiment, the robot came up with three different plans depending on the current configuration of the door and the visibility of its handle. If the door is completely closed, our robot needs to carry out the following actions: `reachHandle; graspHandle; turnHandle; pullDoor; releaseHandle; moveArmToInnerSide; pushDoor`. If the door latch was not locked, it is sufficient to execute: `reachHandle; graspHandle; pullDoor; releaseHandle; moveArmToInnerSide; pushDoor`. If the door is already partially open but the robot does not see the handle, then executing: `moveArmToInnerSide; pushDoor` is sufficient. Some of these actions are visible in the second and third rows of Fig. 3 showing the previous experiment. Further images had to be omitted due to limited space but the reader may consider the video showing parts of this experiment. We conducted this experiment for more than 20 min without any critical failures. It may happen that the execution of an action fails but the execution monitor always detects that and compensates for it immediately by replanning.

Limitations

Despite these encouraging results, there is space for further improvements. Currently, our system is able to identify only a limited variety of patterns in the feature values to find the

preconditions and effects. To find more complex patterns, which are needed to symbolically represent more complex actions, more sophisticated pattern recognition algorithms than one-class k-means clustering are needed.

Our system furthermore assumes that preconditions and effects can be expressed based on a set of predefined features. It would be interesting to substantially extend the list of features available to the robot, such as features that capture physical aspects like forces and dynamics. As the number of features increases, we expect to require more teacher demonstrations. Alternatively, the robot could explore preconditions and effects in simulation to reject irrelevant features that resulted in learning false positive conditions. For instance, a robot that is taught in front of a table may regard the color of the table as an important aspect, although this is obviously not the case. Such a simulation could also allow the robot to explore physical constraints without having to manually provide them. Finally, our approach relies on several thresholds, which reflect the desired accuracy of performing the actions. These thresholds are currently set manually. Learning them should be considered.

Conclusion

We addressed the problem of learning a library of manipulation actions based on demonstrations provided by a teacher. Our approach requires only few demonstrations of actions and identifies the preconditions that need to be fulfilled for each action to be applicable, as well as the effects that are always fulfilled as a result of executing it. These conditions are represented by logical predicates, leading to a symbolic representation in the Planning Domain Definition Language. Therefore, the robot can use existing state-of-the-art planners to solve manipulation tasks which are in sum more complex compared to the taught actions. Furthermore, from the same demonstrations, the robot learns how to physically execute the actions by encoding the observed trajectories as dynamic movement primitives. We implemented our approach and presented experiments using a real PR2 robot to illustrate the capabilities and flexibility of our system, including its ability to react to unexpected changes in the environment.

Acknowledgments

This work has partly been supported by the EC under grant FP7-ICT-248258-First-MM. We thank Malte Helmert for providing his implementation of the FD planner, Peter Pastor for making his DMP implementation available, and Luciano Spinello for the fruitful discussions about single-class SVMs.

References

Asfour, T.; Gyarfas, F.; Azad, P.; and Dillmann, R. 2006. Imitation learning of dual-arm manipulation tasks in humanoid robots. In *Int. Conf. on Humanoid Robots*.

Bentivegna, D.; Atkeson, C.; Ude, A.; and Cheng, G. 2004. Learning to act from observation and practice. *Int. Journal of Humanoid Robotics*.

Billard, A.; Calinon, S.; Dillmann, R.; and Schaal, S. 2008. Robot programming by demonstration. In Siciliano, B., and Khatib, O., eds., *Handbook of Robotics*. Springer.

Boiman, O.; Shechtman, E.; and Irani, M. 2008. In defense of nearest-neighbor based image classification. In *IEEE Conf. on Computer Vision and Pattern Recognition*.

Calinon, S., and Billard, A. 2008. A probabilistic programming by demonstration framework handling skill constraints in joint space and task space. In *Int. Conf. on Intelligent Robots and Systems*.

Ekvall, S., and Kragic, D. 2006. Learning task models from multiple human demonstrations. In *Intl. Symposium on Robot and Human Interactive Communication*, 358–363.

Eppner, C.; Sturm, J.; Bennewitz, M.; Stachniss, C.; and Burgard, W. 2009. Imitation learning with generalized task descriptions. In *Int. Conf. on Robotics & Automation*.

Helmert, M. 2006. The fast downward planning system. *Journal on AI Research* 26.

Kennedy, K.; Namee, B. M.; and Delany, S. 2009. Learning without default: A study of one-class classification and the low-default portfolio problem. In *Conf. on Artificial Intelligence and Cognitive Science*.

Krüger, N.; Piater, J.; Wörgötter, F.; Geib, C.; Petrick, R.; Steedman, M.; Ude, A.; Asfour, T.; Kraft, D.; Omrcen, D.; Hommel, B.; Agostino, A.; Kragic, D.; Eklundh, J.; Krüger, V.; and Dillmann, R. 2009. Formal definition of object action complexes and examples at different levels of the process hierarchy. Technical report.

Mardia, K.; Kent, J.; and Bibby, J. 1979. *Multivariate Analysis*. Academic press.

Omrcen, D.; Ude, A.; and Kos, A. 2008. Learning primitive actions through object exploration. In *Proc. of the Int. Conf. Humanoid Robots*.

Pardowitz, M.; Zöllner, R.; and Dillmann, R. 2006. Incremental acquisition of task knowledge applying heuristic relevance estimation. In *Int. Conf. on Robotics & Automation*.

Pastor, P.; Hoffmann, H.; Asfour, T.; and Schaal, S. 2009. Learning and generalization of motor skills by learning from demonstration. In *Int. Conf. on Robotics & Automation*.

Petrick, R.; Kraft, D.; Mourão, K.; Pugeault, N.; Krüger, N.; and Steedman, M. 2008. Representation and integration: Combining robot control, high-level planning, and action learning. In *Int. Cognitive Robotics Workshop*.

Schölkopf, B., and Smola, A. 2002. *Learning with Kernels*. MIT Press.

Schölkopf, B.; Platt, J.; Shawe-Taylor, J.; Smola, A.; and Williamson, R. 2000. Estimating the support of a high-dimensional distribution. Technical report, Microsoft Research, TR87.

Veeraraghavan, H., and Veloso, M. 2008. Teaching sequential tasks with repetition through demonstration. In *Int. Conf. on Autonomous Agents and Multiagent Systems*.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Munoz-Avila, H. 2009. Learning HTN method preconditions and action models from partial observations. In *Int. Conf. on Artificial Intelligence*.

Automated Planning and Real Systems Based on PLC: A Practical Application in a Didactic Bench of Manufacturing Automation

João Paulo da Silva Fonseca; Rodrigo Nogueira Cardoso; William Henrique Pereira Guimarães; Kauê de Sousa Ribeiro; Alexandre Rodrigues de Sousa; José Jean Paul Zanlucchi de Souza Tavares.

Manufacturing Automated Planning Lab, College of Mechanical Engineering, Federal University of Uberlândia
Av. João naves de Ávila 2121 – Uberlândia – MG – Brazil – CEP 38400.299
E-mails:, jpaulosfonseca@gmail.com, rodrignog@hotmail.com, will_henrique2003@yahoo.com.br, kauedesr@hotmail.com, alexanrsousa@gmail.com, jean.tavares@mecanica.ufu.br

Abstract

Automated planning systems have been developed for more than 40 years, however, their practical application in real problems is still restricted and, many times, a challenge. One of the reasons for such restriction lies on language ruptures. On the one hand, automated planning systems provide a plan with a sequence of actions represented in the Planning Domain Definition Language (PDDL); on the other hand, real systems use equipment as Programmable Logic Controller (PLC), which utilizes languages such as Ladder and centralizes sensor and actuation activities. This paper presents a mechatronics approach integrating automated planner in industrial systems based on PLC network. First, the planning domain was modeled using itSIMPLE. Next, the model was evaluated by two automated planners (Metric-FF and SGPlan6) in respect to optimization specific metrics, in this case, costs minimization. The best plan was chosen to be implemented in the practical system. The sequence of actions defined in PDDL was translated in a Sequential Flow Chart and then, translated in a Ladder diagram, to be implemented in PLC. A practical example is done in a didactic bench, and finally a comparison between real and planned results is presented and discussed.

Keywords: Automated Planning; Manufacturing Automation; Programmable Logic Controller (PLC); Didactic Bench; itSIMPLE.

Introduction

Discrete systems based on Programmable Logic Controllers (PLC) are widely used on the manufacturing automation processes. The internationally recognized efficiency allied to their robustness guarantees their success in industrial applications. On the other hand, for some years several studies have been showing artificial intelligence techniques like neural nets, genetic algorithms, specialized systems, fuzzy logic, automated planning as enhancement proposals for practical systems. Specifically, the automated planning emerged in 1971 with the presentation of the first automated planner: Stanford Research Problem Solver (STRIPS) (Fikes and Nilsson 1971). According to the authors, automated planners consist in a system for automatic resolution of problems. Automated planners runs a search algorithm, based on a model, specifics criteria, an inputs' set (initial state) and a goal (final state); to define the best sequence of actions.

On the other side, most of commercial PLC work with very simple programming languages, such as Ladder and Sequential Flow Chart (SFC) while automated planners make use of a huge formalism provided by Planning Domain Definition Language (PDDL) (Fox and Long 2003) - an international standard for automated planners.

However, what can be seen is a great abyss between those development standards, making it difficult to integrate newer technologies in industrial applications like, for example, automated planners in manufacturing. Tavares and Fonseca (2011) presented an initial study to evaluate this gap between the theoretical field and practical systems by use of a recurrent problem on a supply chain, through a simple proposal.

This work presents a mechatronics approach integrating automated planners in industrial systems based on PLC network. First of all theoretical foundation is presented, followed by methodology with planning domain and problem definition. The integration between plan and PLC is showed for this specific scenario comparing planned and actual results applied to a didactic bench. Discussion and conclusion is presented, followed by bibliographies.

Theoretical Foundation

Next paragraphs present the theoretical base of this work. Initially it reviews automated planning concepts, followed by the itSIMPLE system (Vaquero 2007), and finally there is a brief description about PLC and its programming languages.

Following Ghallad *et al.* (2004) classical Planning requires eight restrictive assumptions, it means, finite system Σ, fully observable, deterministic, static, a set of

goals state S_g, a linearly sequence of action (a_1 … a_n), implicit time and off-line planning.

State transition system Σ is a tuple of (S, A, E, γ), where *S* is a set of state, *A* is a set of action, *E* is an exogenous event and γ is a state-transition function.

It's possible to think of a modern manufacturing process as sequence of actions and whose completion can be detected by sensors responsible to the controllers' observation. These actions can be performed by devices, numerically controlled machines, automated guided vehicles, conveyor belts, manipulator robots, etc. Therefore these actions consist of relatively complex programs in different languages, also including PLC programming languages.

So the sequencing hereby referred can be seen as a translation activity from plans produced by automated planners to control programs for devices and machines. Even though the perception and actuation processes can also be performed by PLC; it's noted that this level of language is not suitable to fit automated planners results. The designing process by itself is not feasible in a low level language as Ladder, even though in the end, it's desired to have a way to automatically stimulate the plan or sequence of actions when using PLC and detect the termination condition of these actions with sensors perceptions (Tavares et al. 2011).

The possibility of applying new design tools for planning and scheduling systems in this type of integration, with the precision, dependencies' analysis, adaptability and intelligent behavior insertion advantages becomes, therefore, very attractive.

Until recently this possibility was very remote, since the automated planning's problems were solely treated with model problems, and extracted directly in formal specification languages such as PDDL. However, since the beginning of this century, a discussion has taken over the automated planning community, which refers to the possibility of utilizing those artificial intelligence techniques in real problems, as manufacturing ones.

Actual problems of great interest have been analyzed with such techniques like logistical problems in port systems (Dahal et al. 2003) or in logistics on the loading and unloading of oil in the São Sebastião's port (Sette et al. 2008), or even in the raw oil routing in pipelines (Li et al. 2005).

The itSIMPLE System

The itSIMPLE system (Vaquero et al. 2009) was designed to allow the user to have a disciplined design process to create knowledge models from different fields of automated planning. The process suggested and implemented in the domain model design tool follows a cyclic sequence of stages inherited from the Software Engineering and Knowledge Engineering combined with experience gained in the design of planning applications.

The software provides a different approach for modeling the planning domain. Its main feature is to enable the entire modeling process to be done through Unified Definition Language (UML) diagrams (OMG 2010). As the group of the PDDL language experts, and its formalisms, is very limited, itSIMPLE opened the door for a larger group of people to be able to model the planning domain from a graphical language. Hence, the software consists in a tool capable of translating the UML model to a corresponding PDDL that can be used by automated planners allowing then to verify and compare different planners with the same domain description.

The itSIMPLE environment incorporates a set of representation languages and theories capable of dealing with requirements and knowledge engineering, in association with the project life cycle, as shown in Figure 1. Among the various specification and modeling languages, the itSIMPLE proposes the utilization of the semi-formal language UML, a diagrammatic language widely known and used in the software and requirements engineering, for the initial steps in building the model and the domain knowledge. Classical Petri nets (Murata 1989) are used for dynamic verification of the plans. And as the community in AI Automated Planning uses PDDL as a standard representation of domains and also as an input representation to the planners, the itSIMPLE integrates this language as well (up to version 3.0 of PDDL) in its design process (Vaquero et al. 2009).

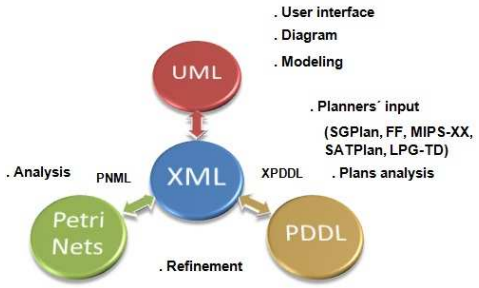


Figure 1: itSIMPLE's structure and languages. (Vaquero et al. 2009)

In the current version, itSIMPLE can communicate with the main planners available in the literature. Therefore, the process of choosing a planner is not in fact done, and this version of the system simply prepares the application domain (practical manufacturing processes and most of the planning problems) for all planners included. In a future version it is intended to eliminate this process, choosing only the planners most appropriate for the problem at hand.

The itSIMPLE is currently a public domain software[1]. This system participated in the three editions of the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS), ranking first in the third edition in 2009.

Programmable Logic Controllers

The PLC are electronic equipment used in flexible automation systems. There are very useful and versatile

1 itSIMPLE. Available at http://code.google.com/p/itsimple/

working tools for applications in command and control systems, therefore they are widely used in industrial markets. The PLC allows the development and easy manipulation of the logic to actuate on the outputs according to the inputs. Thus, we can associate different input signals to control various actuators connected at exit points (Oliveira 1993).

With the evolution of microprocessors, it was the increased processing power and memory of the PLC that started to become attractive, besides all fields of industrial activity and also the area of building automation working in climate control, alarms, and illumination. The current generations of controllers have advanced control functions, availability of a large number of inputs and outputs, as well as ease of communication with supervision systems and smart sensors and actuators.

The languages used in PLC are basically among the five languages defined by IEC 1131-3, which are the Ladder Language, Function Block Diagram, SFC, and other textual languages like Instructions List and Structured Text.

The Ladder diagram is a graphical programming language derived from the representation of the circuit diagram which makes use of relay controllers directly connected. This diagram is characterized by power lines on the right and left of the diagram; these are linked by current paths with switch elements (normally open contacts, normally closed contacts) and coils. Figure 2 illustrates an example of Ladder diagram language related with bushing process of two different parts (A and B).

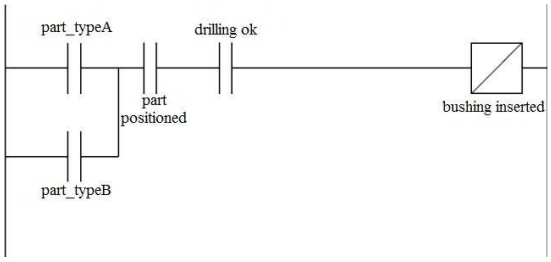


Figure 2: Ladder diagram language example. (Tavares 2011)

Commercially, SFC and Ladder languages are the two most representative of the PLC languages universe. According Miyagi (1996), the Ladder diagram corresponds to a logical representation based on the circuit diagram of relays, whose utilization was widespread before the rise of the PLC, which explains the preference for this type of language for most programmers. On the other hand the SFC development was motivated by the interest in graphical tools to represent explicitly the functions to describe sequential processes for industrial applications.

Methodology

Having presented the theoretical foundation for this work, the methodology used for its development follows Vaquero et al. (2009). Nevertheless, the process and problem of domain modeling has been entirely developed using the itSIMPLE software.

Planning Domain Modeling

Initially it was developed a Didactic Test Bench (Fonseca 2011) where it would possible to evaluate the application of automated planning in practical systems. This bench is composed by one vehicle, one supplier reservoir, two customer reservoirs and two pumps to satisfy several demand from clients. The vehicle must receive product from the supplier reservoir and carry it up to the customer reservoirs. The level of each customer reservoir will be a function of client demand for each customer. Customer reservoir is emptied by its own electro-pump and fulfilled by the electro-pump of the vehicle. Physically, the product is delivered directly to the supplier reservoir, closing the cycle and ensuring the continued functioning of the bench. Figure 3 shows actual photos of the Didactic Bench.

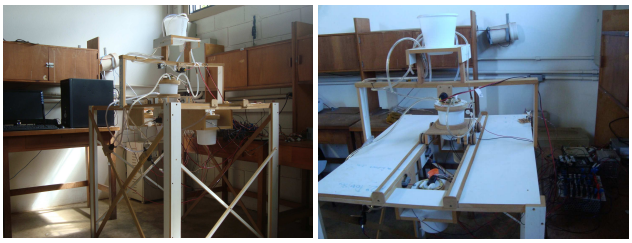


Figure 3: Didactic bench photos.

The modeling process begins at the construction of the *Use Cases Diagram*. An analysis of the characteristics of the proposed problem allows the identification of three agents, the supplier, the vehicle and the customer. The agent *Supplier* will be responsible for carrying out the Use Case *Load*, the agent *Vehicle* will be responsible for carrying out the Use Cases *Move*, *Load* and *Unload* while the agent *Customer* will be responsible for the Use Cases *Unload*, *PartialOrder* (for partial deliveries), *FinalOrder* (to fulfill partial deliveries) and *CompleteOrder* (for full deliveries). The Use Case *Load* requires the activities of both *Vehicle* and *Supplier* agents simultaneously while the Use Case *Unload* requires the activities of both *Vehicle* and *Customer* agents. Figure 4 shows the Use Case diagram of the Bench domain.

Proceeding with the modeling process, the static structure of the domain represented by Classes Diagram must be conceived based on the description of the Use Cases. The primary elements of this domain are the *Vehicle*, the *Customers*, the *Supplier* and the *Client Class*. In addition to these Classes, the diagram is formed by the *LevelTransf* (responsible for the discretization of quantities of product sold and displayed by the level sensors) and *Global* (containing all global variables of the domain), the first being a *Resource Class* and the second a *Global Class* (*stereotype* <<*utility*>>). Figure 5 shows the Class diagram of the Bench domain.

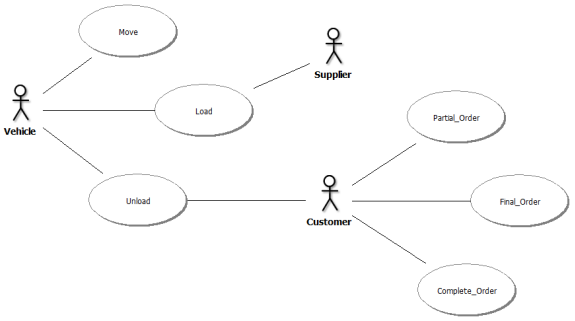


Figure 4: Bench´s Use Case Diagram

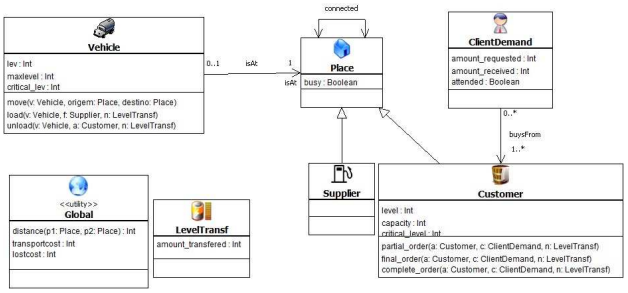


Figure 5: Bench's Class Diagram

Each class must have attributes and methods that represent the behavior of real objects in the system. For example, the *Vehicle* Class has the necessary information to instantiate all objects of the V*ehicle Type* these information are the class's attributes. In this model the *Vehicle* Class has three attributes: *maxlevel* (Int), correspond to the maximum level of the vehicle's reservoir; *critical_lev* (Int), correspond to the minimum level of the vehicle's reservoir; and *lev* (Int), representing the current level. Moreover, the *Vehicle* Class has an association *isAt* with the *Place* Class in order to identify which place the vehicle is at the exact moment. To ensure proper functioning of the system, Agents Classes must take actions to ensure the functionality of the plant. So the *Vehicle* Class has three operators: *move*, *load* and *unload*. The other classes follow the same methodology. The modeling of this domain can be found in better detail in Fonseca and Tavares (2011).

Automated planners work in such a way that, given an initial situation and a final situation, an algorithm scans a search tree in order to find the best solution for the characteristics of the project, which may be minimize or maximize processing time, operation costs and time etc.

The metrics developed in this model, which can be considered minimization of costs, are *transportcost* and *lostcost* (cost for postponed demand), according to Figure 6. The loss cost naturally is something agreed by a clause in the contract between supplier and distributor and, for this specific case was chosen arbitrarily a weight of value 50.

These variables are defined as attributes of the *Global* Class and its logical representation is given at the State Diagrams as rules of the actions of Vehicle (Figure 7) and Distributor (Figure 8). By definition, the state diagram is where the classes with relevant dynamic characteristics are modeled.

```
transportcost = transportcost +
distance(origin,destination)*(v.lev+1)

lostcost = lostcost +
50*(c.amount_requested - c.amount_received)
```

Figure 6: Metrics of the model.

Figure 7: Vehicle´s State Diagram.

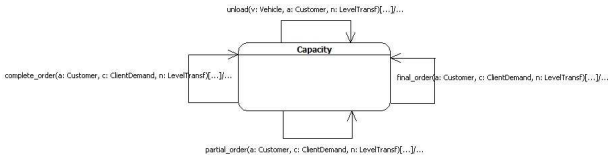


Figure 8: Distributor´s State Diagram.

Problem Definition

The problem to be solved is to move, load and unload the vehicle *v1* in order to attend the demand of Clients. For practical example, initially the Vehicle is stationed under the Supplier reservoir; the Distributors reservoir *a1* and *a2* contains 20% of available stock, the same value of the critical one.

Moreover, the Client *c1* has an unmet demand of 70% of Distributor reservoir *a1*, while the Client *c2* has an unmet demand of 100% of Distributor reservoir *a1*. The final state, defined as a goal snapshot is: Vehicle stationed at the original position (Supplier); Clients *c1* and *c2* with attended demands; Distributors reservoir *a1* and *a2* with intermediate level, specifically 50%. The Figures 9 and 10 shows the initial snapshot and the goal snapshot corresponding to this problem.

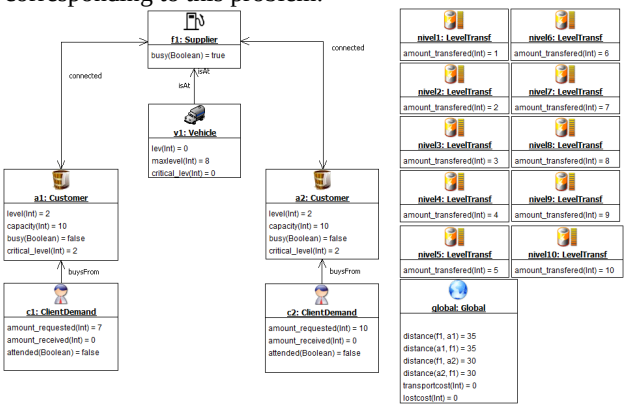


Figure 9: Initial snapshot

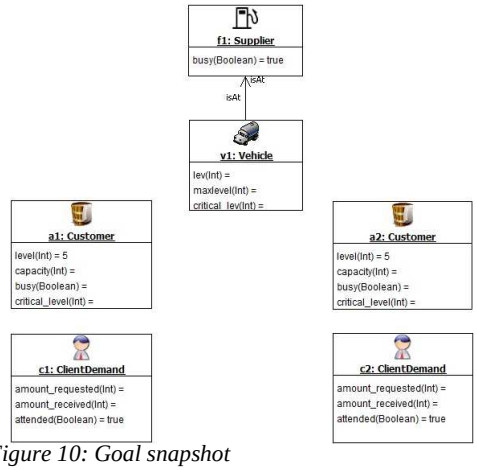


Figure 10: Goal snapshot

Plan Definition

The itSIMPLE has a significant number of automated planners, such as SGPlan6 (Hsu and Wah 2008), Metric-FF (Hoffmann 2003), MIPS-XXL (Edelkamp and Jabbar 2008), LPG-td (Gerevini et al. 2004), HSP (Bonet and Geffner 2000) etc. This work presents the result obtained with the Metric-FF and the SGPlan6 planners, which generated the plan-solutions described below, in Figures 11 and 12, respectively. These possible solutions lead the Vehicle *v1* to perform the actions sequentially to take the system from initial situation to goal situation.

The Integration between Plan and PLC

The plan-solution was translated directly in a SFC model, and, the base of PLC Ladder program. After Ladder deployment, the solution-plan could control the Didactic Test Bench that has the Omron CJ1M ETN21 CPU13 PLC (Omron 2001).

Initially, the sequence of actions defined by the plan-solution model is translated directly into a discrete event model of the SFC type. The model development follows a logic in which every action defined in the plan-solution sequence is related to a state in the SFC. The model begins at the initial snapshot and ends in the goal snapshot. The intermediate states are linked with the preconditions and post-conditions of each action as transitions receptiveness between states of the SFC model. The SFC model which corresponds to the plan-solution suggested by the Metric-FF planner is partially presented in Figure 13.

For each sensor and actuator variables were assigned in the PLC according to Tables 1 and 2.

As the comparison function block of CX-one (programming software of Omron CJ1M CPU13 PLC) works with memory locations, the comparison between the levels perceived by sensors with the expected level in each plan states was performed by auxiliary memories. For this, the perceived values of the level sensors were stored in auxiliary memories, as described in Table 3. The values perceived by the analog input I0.00 (vehicle level sensor) were stored in the memory location D100; values perceived by the analog input I0.01 (distributor1 level sensor) were stored in the memory location D101; and the values perceived by the analog input I0.02 (distributor2 level sensor) were stored in the memory location D102.

```
1: LOAD V1 F1 LEVEL8
2: MOVE V1 F1 A1
3: UNLOAD V1 A1 LEVEL8
4: MOVE V1 A1 F1
5: LOAD V1 F1 LEVEL8
6: MOVE V1 F1 A2
7: UNLOAD V1 A2 LEVEL8
8: MOVE V1 A2 F1
9: COMPLETE_ORDER A1 C1 LEVEL7
10: LOAD V1 F1 LEVEL2
11: MOVE V1 F1 A1
12: UNLOAD V1 A1 LEVEL2
13: MOVE V1 A1 F1
14: PARTIAL_ORDER A2 C2 LEVEL8
15: LOAD V1 F1 LEVEL5
16: MOVE V1 F1 A2
17: UNLOAD V1 A2 LEVEL5
18: MOVE V1 A2 F1
19: FINAL_ORDER A2 C2 LEVEL2
```
Figure 11: Metric-FF's plan.

```
1: LOAD V1 F1 LEVEL8
2: MOVE V1 F1 A2
3: UNLOAD V1 A2 LEVEL8
4: MOVE V1 A2 F1
5: PARTIAL_ORDER A2 C2 LEVEL8
6: LOAD V1 F1 LEVEL5
7: MOVE V1 F1 A2
8: UNLOAD V1 A2 LEVEL5
9: FINAL_ORDER A2 C2 LEVEL2
10: MOVE V1 A2 F1
11: LOAD V1 F1 LEVEL8
12: MOVE V1 F1 A1
13: UNLOAD V1 A1 LEVEL8
14: MOVE V1 A1 F1
15: COMPLETE_ORDER A1 C1 LEVEL7
16: LOAD V1 F1 LEVEL2
17: MOVE V1 F1 A1
18: UNLOAD V1 A2 LEVEL2
19: MOVE V1 A1 F1
```
Figure 12: SGPlan6's plan.

To implement the plan-solution in the PLC, both attributes and operators of each class (represented in the Class Diagram) should be associated with a state set of the sensors and actuators of the Bench. For example, the action 2: (MOVE V1, F1, A1) corresponds to the movement of the vehicle to the left, in other words, digital output 3 (Q1.02) enabled and digital output 4 (Q1.03) disabled. The pre-condition for enabling Q1.02 is the vehicle stationed in the supplier, in other words, *fc1* activated which is the same as digital input 1 (I0.00) enabled; the vehicle level sensor should indicate 100%, which corresponds to the auxiliary memory D100 = 3784; and the distributors´ level sensors 1 and 2 should indicate 20%, which corresponds to the following values for the auxiliary memories: D101=736 and D102 = 1061 (sensor values differ among themselves in a percentage scale and have been adjusted according to a static calibration process, precisely due to

the fact that uncertainties exist in the real world elements) When the system reaches the state presented, the digital output Q1.02 must be enabled, allowing the movement of the vehicle to the left until the post-condition for this action is achieved. The post-condition for this action is the vehicle stationed on distributor *a1*, in other words, *fc2* activated or digital input 2 (I0.01) enabled and reservoir levels similar to those perceived in the precondition. This should occur for all actions in the Plan. Thus, Table 4 represents part of the *from/to Table* needed to facilitate the interface between the plan-solution presented above and the construction of the Ladder diagram used in the PLC.

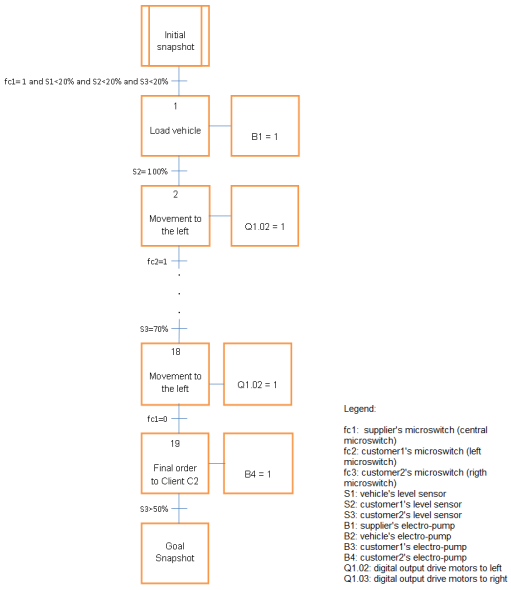


Figure 13: SFC corresponding to plan-solution.

Table 1: Digital input

Microswitch	Digital Input
fc1	I0.00
fc2	I0.01
fc3	I0.02

Table 2: Digital output

Actuator	Digital Output
B1	Q1.00
B2	Q1.01
Drive motors to left	Q1.02
Drive motors to right	Q1.03
B3	Q1.04
B4	Q1.05

Table 3: Analogic Input x Auxiliary Memory Location

Level Sensor	Analogic Input	Auxiliary Memory
S1	I2001	D100
S2	I2002	D101
S3	I2003	D102

Table 4: Partial from/to Table

Snapshot/ Action	**Plan**	**PLC**
Initial Snapshot	-	-
	(isAt v1 f1)	I0.00=1
	(=(lev v1)2)	D100=1322
	(=(level a1)2)	D101=736
	(=(level a2)2)	D102=1061
	(=(amount_reveived c1)0)	
	(=(amount_reveived c2)0)	
Action 1	(Load v1 f1 level 8)	Q1.00=1 until D100>3783
	(isAt v1 f1)	I0.00=1
	(=(lev v1)10)	D100=3784
	(=(level a1)2)	D101=736
	(=(level a2)2)	D102=1061
	(=(amount_reveived c1)0)	
	(=(amount_reveived c2)0)	
Action 2	(Move v1 f1 a1)	Q1.02=1 and Q1.03=0 until I0.01=1
	(isAt v1 a1)	I0.01=1
	(=(lev v1)10)	D100=3784
	(=(level a1)2)	D101=736
	(=(level a2)2)	D102=1061
	(=(amount_reveived c1)0)	
	(=(amount_reveived c2)0)	
. . .	. . .	. . .
Action 19	(Final_order a2 c2 level 2)	Q1.05=1 until D102=1958
Goal Snapshot	(isAt v1 f1)	I0.00=1
	(=(lev v1)2)	D100=1322
	(=(level a1)5)	D101=1854
	(=(level a2)5)	D102=1773
	(=(amount_reveived c1)10)	
	(=(amount_reveived c2)7)	

The actions of the plan-solution are represented in the PLC by the memory addresses W10.xx where xx is the value given to the plan-solution's action (Table 5).

Thus, we started with a bench´s model, which it was used by an automated planner in order to generate a plan from the requirements given. This plan was translated into an operational language, so that, joining it with the

mapping of the PLC inputs and outputs, it was possible to make the Ladder diagram corresponding to the plan-solution. Figure 14 illustrates a part of the Ladder diagram obtained through the methodology described for this paper.

In this case the initial snapshot W10.00 event enables the relay W20.00 (Trans0). This relay marks the transition between the initial snapshot and Action 1 (LOAD V1 F1 LEVEL8). This action is associated directly to power of the electro-pump 1 by enabling digital output Q1.00. In addition, Action No. 1 is subjected to non-enabling the transition 1 (between this action and the next). This, on the other hand, will be enabled as soon as the vehicle´s level sensor indicates 100%, which corresponds to the auxiliary memory D100 = 3784. At this point, the Action No. 0 is disabled and Action No. 1, enabled. This process continues until the system reaches the goal snapshot.

Table 5: Plan Actions and the PLC's Memory Addresses

Action	PLC's Memory Addresses
1	W10.01
2	W10.02
3	W10.03
. . .	. . .
17	W10.17
18	W10.18
19	W10.19

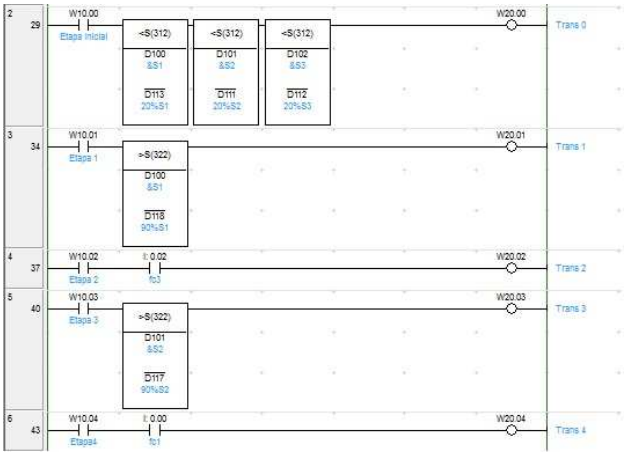


Figure 14: Partial Ladder Diagram of Bench's Domain.

For validating the model developed in itSIMPLE, the real costs of the plan-solution were accounted and compared with planning costs. The Figure 15 shows the comparative result between Planning Transport Cost and Actual Transport Cost, for the planner Metric-FF. Since the Figure 16 illustrates the same result for the planner SGPlan6. The Table 6 presents the results expected and obtained at the goal snapshot.

The transport cost is a function of distance and level transferred. However, the uncertainty of practical system, mainly caused by level sensors, generated a discrepancy between the Planning and Real Transport Cost.

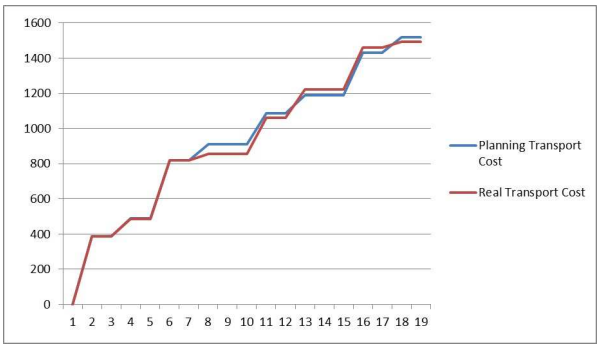


Figure 15 Development of the real and planning transport cost for the planner Metric-FF.

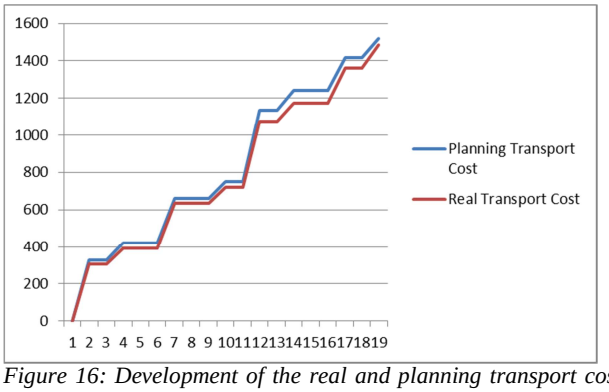


Figure 16: Development of the real and planning transport cost for the planner SGPlan6.

Table 6: Comparison between goal snapshot modeled and tested.

	State	S1(%)	S2(%)	S3(%)
Model	Goal State	-	50	50
Tests				
Metric-FF	State 19	-	47	34
SGPlan6	State 19	-	61	43

Discussion and Conclusion

We can note that, by the automated planning methodology, it is possible to: define the objective state for the system and then act with demand forecast, anticipating the distributors´ stock for future deliveries and market variations; apply the variables optimization metric for minimization of costs; simulate behavior of real system before the practical implementation.

Nevertheless, the results show that the modeling and planning capacity of a system is directly associated with the reliability of the instrumentation system. Systems with high level of uncertainty related with instrumentation measures, can feed planning techniques with wrong data reducing its efficiency. Despite sensors uncertainty, plan-solution can be based on a forecast demand. This work

doesn't deal with stochastic demands and re-planning activities.

However, there are more complex situations where the processing time becomes a variable to be minimized. According to recent studies (Huang et al. 2011), when the processing time is presented as a new variable in the model, a way to optimize this variable is by parallelization process, by the distribution of actions and processes on different machines . Another solution to reduce automated planning processing time is introducing Partially Observable Markov Decision Process (POMDP) (Ghallab *et al.* 2004).The analysis of such cases is a further work for this project.

This work ratifies the applicability of automated planners in practical cases and their use in manufacturing automation. Moreover, new horizons are opening for the automation community of regarding the possibility to use Artificial Intelligence to complete control a productive process, considering specific cases and object-oriented and not only a generic case.

The best of all worlds for the automated planning application in manufacturing automation would be the integration of disparate languages, in this case, PDDL and Ladder. This integration would be possible by developing an application able to realize an interface between PLC and itSIMPLE. Thus, we can mention another further work, which is the development of an application for the automatic integration between the planner allocated in itSIMPLE and the process controller from Petri Nets to Ladder. This interface is PLC dependent.

Acknowledgments

Authors are grateful to Prof. Dr. José Reinaldo Silva, Dr. Tiago Stegun Vaquero, UFU, FEMEC, FAU, CAPES, CNPQ and FAPEMIG.

References

Bonet, B. and Geffner, H. 2000. HSP: Heuristic Search Planner Entry at AIPS-98 Planning Competition, AI Magazine Vol 21(2).

Castrucci, P. and Moraes, C. C. de. 2001 *Engenharia de Automação Industrial*. São Paulo: LTC

Dahal, K.; Galloway, S.; Burt, G.; Mcdonald, J. and Hopkins, I. 2003. Port System Simulation Facility with an Optimization Capability. *International Journal of Computational Intelligence and Applications*, vol. 3, no. 4, pp. 395-410.

Edelkamp, S. and Jabbar, S. 2008. *MIPS-XXL: Featuring External Shortest Path Search for Sequential Optimal Plans and External Branch-And-Bound for Optimal Net Benefit*. In 6th International Planning Competition Booklet, Sydney, Australia.

Fikes, R. E. and Nilsson N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Journal of Artificial Intelligence*, 2:189–208.

Fonseca, J.P.S. Desenvolvimento e Montagem de uma Bancada Didática de Planejamento Automático. 2011. 165 f. Redaction of Course Conclusion, Federal University of Uberlândia, Uberlândia.

Fonseca, J.P.S. and Tavares, J.J.P.Z.S. 2011. Didactic Test Bench for Automated Planning. In Procedings of 21st Brazilian Congress of Mechanical Engineering, Natal, Brazil.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension of PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124.

Gerevini, A., Saetti, A., Serina, I. and Toninelli, P. 2004. LPG-*TD*: a Fully Automated Planner for PDDL2.2 Domains. In *American Association for Artificial Intelligence (AAAI)*.

Ghallab, M., Nau, D., Traverso, P. 2004. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, May 2004

Hoffmann, J. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. 2003. *Journal of Artificial Intelligence Research*, 20:291–341.

Hsu, C.W. and Wah, B.W. 2008. The SGPlan Planning System in IPC-6**.** Illinois, Usa: University Of Illinois, 2008. 3 p.

Huang, C.Y.; Chen, W.L. and Yeh, S.C. 2011. Supply Network Planning for Memory Module Industry by Distributed Parallel Computing. In Proceedings of 21st International Conference on Production Research. Sttutgart, Germany.

Li, J.; Wenkai, L.; Karimi, I. A. and Srinivasan, R. 2005. Robust and Efficient Algorithm for Optimizing Crude Oil Operations, In: American Institute of Chemical Engineers Annual Meeting.

Miyagi, P. E. 1996. *Controle Programável***:** *Fundamentos do controle de sistemas a eventos discretos*. São Paulo: Blucher.

Murata, T. 1989. Petri Nets: Properties, Analysis and Applications. IEEE Proceedings, vol. 77, no.0 4, April.

Nguyen, A. 2003. Challenge ROADEF'2005: Car Sequencing Problem. Renault, France.

Oliveira, J.C.P. 1993. Controlador Programável. São Paulo: Makron Books.

OMG – Object Management Group. Unified Modeling Language specification. Available at: <http://www.omg.org/uml>. Accessed on: jul,14 2010.

Omron Corporation (Japan) (Org.). Programmable Controllers: Operation Manual. Tokyo, 2001. CD-ROM.

Sette, F.M.; Vaquero, T.S.; Park, S.W. and Silva, J.R. 2008 . Are Automated Planers up to Solve Real Problems?. *Proc. IFAC Conf.*, Seoul.

Tavares, J.J.P.Z.S. and Fonseca, J.P.S. 2011. Supply Chain Didactic Testing Bench With Automated Planning Tool. In Proceedings of 21st International Conference on Production Research. Sttutgart, Germany.

Tavares, J.J.P.Z.S.; Fonseca, J.P.S.; Vaquero, T.S. and Silva, J.R. 2011. Integração de Planejamento Automático e Sistemas Reais Baseados em CLP. In Proceedings of X Simpósio Brasileiro de Automação Inteligente. São João Del Rey, Brazil.

Vaquero, T.S. 2007. ITSIMPLE**:** Ambiente integrado de análise de domínios de planejamento automático. 316 f. MsC. diss. University of São Paulo, São Paulo, Brazil.

Vaquero, T.S.; Silva, J.R.; Ferreira, M.; Tonidandel, F. and Beck, J.C. 2009. From Requirements and Analysis to PDDL in itSIMPLE3.0. In Procedings of International Competition in Knowledge Eneginering for Planning and Scheduling (ICKEPS 2009), 54-61.

A Constraint-Based Approach for Multiple Non-Holonomic Vehicle Coordination in Industrial Scenarios

Federico Pecora and **Marcello Cirillo**
Center for Applied Autonomous Sensor Systems
Örebro University, SE-70182 Sweden
<name>.<surname>@oru.se

Abstract

Autonomous vehicles are already widely used in industrial logistic settings. However, applications still lack flexibility, and many steps of the deployment process are hand-crafted by specialists. Here, we preset a new, modular paradigm which can fully solve logistic problems for AGVs, from high-level task planning to vehicle control. In particular, we focus on a new method for multi-robot coordination which does not rely on pre-defined traffic rules and in which feasible and collision-free trajectories are calculated for every vehicle according to mission specifications. Also, our solutions can be adapted on-line to exogenous events, control failures, or changes in mission requirements.

Introduction

Industrial actors involved in the development of autonomous vehicles (e.g., autonomous forklifts for warehouses) are constantly interested in decision support tools which could improve the flexibility and the performance of their products. Atlas-Copco[1] (Larsson, Appelgren, and Marshall, 2010), Kiva Systems[2], INRO[3] (Thomson and Graham, 2011) and Kollmorgen[4], among others, aim to achieve complete automation in Autonomous Ground Vehicle (AGV) deployments. Although it is current practice to employ automated solutions in several aspects of logistics automation, many key parts of the deployment phase are still ad-hoc and manual. For instance, the definition of AGV paths is often done off-line, and these paths are hand crafted for each different setting. Also, large-scale industrial deployments of AGVs rarely include more than very crude heuristics to optimize mission scheduling. Another limitation of current industrial solutions is the resolution of spatial conflicts, which is often performed off-line through manually synthesized traffic rules, whose correctness cannot be formally proved. Other fallacies of real systems include the lack of support for resources and an often only partial support for on-line mission constraint posting (e.g., changed deadlines, new requirements, collapses of resource availability).

When automated solving components are used in industrial AGV deployments, these are usually not integrated. For instance, it is often the case that path planning is de-coupled from trajectory generation, or that the allocation of vehicles to destinations does not depend on the trajectory that will actually be followed by the vehicles. This leads to inefficiencies in the quality of the solutions and reduced flexibility in dealing with contingencies. Furthermore, the methodology for assessing how many AGVs are necessary for a particular deployment typically consists of what-if analyses on simulated scenarios. This analysis becomes more cumbersome and, especially, less accurate if many de-coupled solving modules are employed.

In this paper, we introduce a system which strives to facilitate all phases of deployment of AGVs in real settings. Our approach is modular, in that it can be applied "partially" or in "pieces", depending on the requirements of the particular deployment at hand. For instance, AGV paths may be automatically generated by a path planner (as is the case in this paper), or the routes could be manually decided by a field specialist (as is often the case in industrial settings). The same principle applies to task planning, which can be automated or manually decided by human operators (in this paper, the specific task planning algorithm is omitted). To achieve this, the modules rely on a shared, constraint-based representation of the overall problem, and each module refines this representation from "its own" point of view.

Related Work

Many of the problems underlying the automation of task and motion planning for industrial vehicles have been addressed in research. As a result, important advancements have been achieved in addressing separate parts of the overall problem. Algorithms such as M^* (Wagner and Choset, 2011), an extension of the classical A^* to multi-robot systems, and the work of Luna and Bekris (2011), whose focus is a new method for multi-robot path planning which is computationally efficient and complete, are recent examples of promising theoretical results. A new system for the coordination of large multi-robot teams has been presented by Kleiner, Sun, and Meyer-Delius (2011). The authors propose a system that generates an overall, optimal road map configuration. However, in this work the agents are assumed as moving on a grid, and the local motions are calculated for each robot independently from the motions of other robots.

A common approach for multi-robot path planning which

[1] http://www.atlascopco.com
[2] http://www.kivasystems.com/
[3] http://www.inro.co.nz/
[4] http://www.kollmorgen.com

usually guarantees fast results is the assignment of priority levels to different robots. This can be seen as an improved version of hand-coded traffic rules, but cannot ensure deadlock-free situations. An example of an algorithm which relies on this paradigm is presented by ter Mors (2011). The overall system can find optimal, conflict-free routes in low polynomial time, but relies on a pre-defined roadmap shared by all agents for path planning. Desaraju and How (2011) further extend the idea of prioritized path planning, by substituting the pre-defined priority levels with a merit based token, which is passed among agents. Once a robot has planned its own path, it circulates it to the other team members, which in turn update their trajectories.

In recent years, a number of approaches to multi-robot coordination have been presented which rely on pre-defined paths. Examples include the work of Kleiner, Sun, and Meyer-Delius (2011), whose algorithm is resolution complete and can be easily applied to situations in which a large number of agents is moving. However, the overall coordinated motions lack flexibility, as time is considered only implicitly in configurations along the paths. Therefore, the final result cannot take into account motion delays, or explicit temporal constraints imposed on the single agents and their positions over time.

A Constraint-Based Approach

A trajectory is a sequence of points and an associated temporal profile, which specifies exactly when the vehicle will be in a certain point. Instead of reasoning in terms of one trajectory, in our approach we reason in terms of *temporal* and *spatial constraints* on trajectories. The collection of spatial and temporal constraints on one vehicle's trajectory is called a *trajectory envelope*. We can describe the overall mission planning problem (which will be defined precisely shortly) as a Constraint Satisfaction Problem (Tsang, 1993, CSP) where variables represent vehicles, their values represent possible trajectories that they should execute, and constraints are spatial and temporal requirements on these trajectories. A solution to the overall problem is therefore an assignment of trajectories to vehicles such that none of the requirements on trajectories is violated.

Trajectory envelopes are the key representational elements used to express and solve our problem. More precisely, (see also figure 1):

Definition 1. *A* trajectory envelope *is a triple* $\langle \mathcal{S}, \mathcal{D}, \mathcal{O} \rangle$ *where*

- $\mathcal{S} = \{S_1, \dots S_n\}$ *is a set of linear* spatial constraints *in the form* $A_i x + B_i y \leq C_i$*; each set* S_i *of spatial constraints specifies a convex region in the map within which the vehicle must be contained;*
- $\mathcal{D} = \{D_1, \dots D_n\}$ *is a set of linear* temporal constraints *in the form* $l_i \leq t_i^e - t_i^s \leq u_i$*, where* t_i^s (t_i^e) *represents the time at which the vehicle enters (exits) the area specified by the set of spatial constraints* S_i*; these constraints provide bounds on when the vehicle is within the convex region specified by* S_i*;*
- $\mathcal{O} = \{O_1, \dots O_{n-1}\}$ *is a set of linear* temporal constraints *in the form* $l_i \leq t_i^e - t_{i+1}^s \leq u_i$*; these constraints provide bounds on when the vehicle is within the (convex) spatial overlap between* S_i *and* S_{i+1}*.*

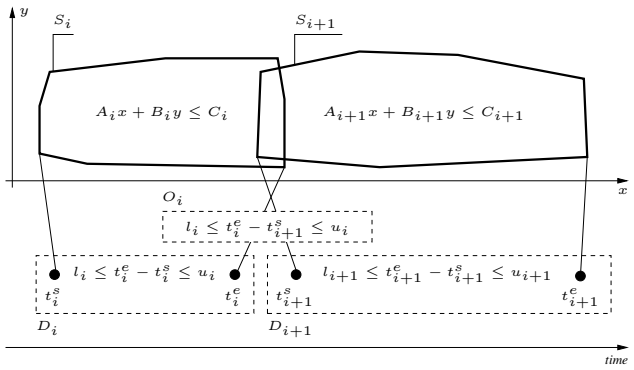


Figure 1: A trajectory envelope consisting of two spatio-temporal polygons.

Problem Statement

Given N vehicles, we define the overall mission planning problem in our scenario as follows:

Definition 2. *A* mission planning problem *is a tuple* $\langle M, \mathcal{I}, \mathcal{G}, \mathcal{T}, \mathcal{V} \rangle$*, where*

- M *is a metric map of the environment;*
- $\mathcal{I} = \{l_1, \dots l_N\}$ *is a set of coordinates in the map specifying the* initial location *of all vehicles;*
- $\mathcal{G} = \{G_1, \dots G_m\}$ *is a set of* goals *in the form* $\langle k, s, g \rangle$*, each specifying that* k *loads must be transported from location* s *to location* g*;*
- $\mathcal{T} = \{T_1, \dots T_n\}$ *is a set of* temporal constraints *on* $\mathcal{G} \cup \mathcal{I}$*; these constraints are in the form* $l_i \leq t_x - t_y \leq u_i$*, where* t_x *and* t_y *are start/end timepoints of a goal or initial position;*
- $\mathcal{V} = \{V_1, \dots V_N\}$ *are the capacities of the vehicles (maximum amount* $\in \mathbb{N}$ *of load each vehicle can carry).*

Finding a solution to a mission planning problem is decomposable into three parts; first, compute an allocation of vehicles to goals which achieves the necessary displacement of loads to places; second, compute the trajectory envelopes $\mathcal{E}$ for each vehicle; third, synthesize a set of temporal constraints $\mathcal{T}_{\text{col}}$ imposing that spatio-temporal polygons intersect either only in space, or only in time, or not at all. Note that a solution to a mission planning problem in fact represents sets of possible trajectories for each vehicle.

Our approach is based on four functional modules, namely *task planning*, *trajectory planning*, *trajectory scheduling*, and *control*. All modules output constraints capturing some aspect of the mission's requirements. Together, these constraints define trajectory envelopes for all vehicles, and provide a global representation of the mission, from its high-level goals to the specific trajectories vehicles must execute to achieve these goals. Each of the four modules progressively refines the representation, imposing increasingly specific requirements:

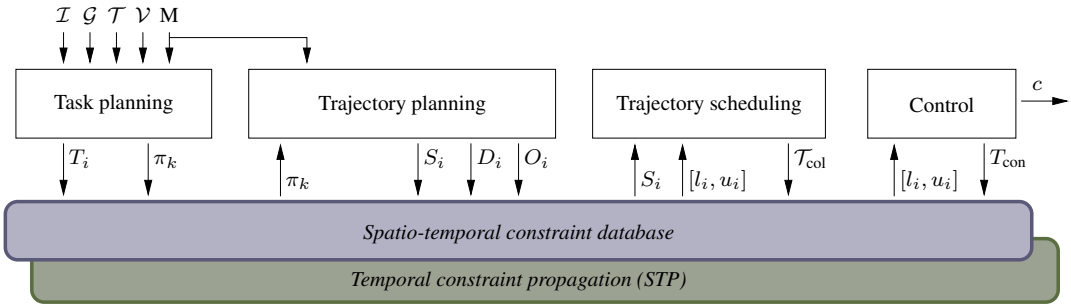


Figure 2: Overall information flow of the four solving modules.

1. **Task planning** decides goal locations for currently available vehicles, therefore constraining the start and end points of vehicle trajectories. Also, task planning may impose quantified temporal requirements on these trajectories, e.g., "vehicle A must reach goal (x, y) before time t", or "vehicle A must reach its goal before vehicle B".
2. **Trajectory planning** decides locations that should be visited in-between goals for each vehicle. Crucially, these are not simple paths between the initial and goal positions, rather they are trajectory envelopes, i.e., temporally-constrained sets of spatial constraints.
3. **Trajectory scheduling** imposes further temporal constraints which ensure that the trajectory envelopes of different vehicles do not intersect in time and space. As explained below, this is a hybrid form of spatio-temporal reasoning based on state-of-the-art scheduling techniques.
4. The **Control** module employs a kinematic and geometric model of a vehicle to generate and follow one specific trajectory that lies within the trajectory envelope obtained as a result of the above modules. This results in appropriate control signals for the vehicle *or* in the selection of a new trajectory within the trajectory envelope in the case that the currently selected trajectory does not adhere to the constraints.

Note that commitment to a specific trajectory in the above scheme is performed only at the control level, and that the controller is given the constraints within which it can select one of many trajectories to follow. As we will show, the way in which these constraints are posted and propagated by the first three modules ensures that for each vehicle there exists a trajectory, within the constraints, which is feasible with respect to *all vehicles*. Also, due to the particular type of hybrid temporal-spatial scheduling performed by the trajectory scheduling module, the selection of mutually-compatible trajectories by all vehicles can be done in polynomial time.

All communication between the four modules occurs through a *spatio-temporal constraint database* (see figure 2), which contains variables and constraints defining an overall CSP. The variables of this problem are spatio-temporal polygons, and the constraints are spatial or temporal relations defining the trajectory envelopes. Note that the temporal constraints in $\mathcal{D}$ and $\mathcal{O}$ constitute a Simple Temporal Problem (Dechter, Meiri, and Pearl, 1991, STP), which is solvable in cubic time through the Floyd-Warshall (Floyd, 1962) temporal constraint propagation algorithm. The algorithm computes the lower and upper bounds $[l_i, u_i]$ of all timepoints given the constraints in the database, and is triggered every time a constraint is added to the database. Thus, through temporal constraint propagation, the bounds of all timepoints t_i of the spatio-temporal polygons are maintained at all times consistent with the temporal constraints that are present in the spatio-temporal database. Note also that if a temporal constraint is added which invalidates previously existing temporal constraints, the constraint database can detect this though a propagation failure, and thus rejects this constraint. This feature of temporal constraint propagation is employed in the trajectory scheduling to search for temporal constraints that avoid collisions between vehicles.

For convenience, we will refer to the sets $\mathcal{S}$ and $\mathcal{D} \cup \mathcal{O}$ as the *spatial* and *temporal envelopes* of a trajectory, respectively. Also, we refer to the i-th set of spatial and temporal constraints $\{S_i \cup D_i \cup O_i\}$ on a trajectory envelope as a *spatio-temporal polygon*. Two spatio-temporal polygons i and j are *spatially overlapping* if $S_i \cap S_j \neq \emptyset$. Temporal overlap is less straightforward: since the underlying STP maintains bounds on the timepoints of spatio-temporal polygons, temporal overlap must be assessed by choosing an earliest start time for the timepoints. Specifically, two spatio-temporal polygons i and j are said to be *temporally overlapping in the earliest start time solution* if $[l_i^s, l_i^e] \cap [l_j^s, l_j^e] \neq \emptyset$. Note that two spatially and temporally overlapping polygons belonging to trajectory envelopes of different vehicles entail that the vehicles may collide.

Solving a Mission Planning Problem

It is often the case in real-world deployments that a particular task allocation strategy, i.e., a task planning module, is given and cannot be substituted. This is due to the often very domain-specific objective functions, preferences and characteristics of the application scenario (e.g., a milk packaging factory vs. an underground mine). For this reason, in the following sections we omit details about task planning and focus on modules (2–4). Consequently, we assume for the purposes of the following description that a task planner has decided, for each goal $G = \langle s, g, k \rangle$, a high-level plan that achieves the displacement of k loads from s to g by an ap-

propriate set of vehicles. Each element of this plan is in the form $\pi_k = \langle i, f, s, g \rangle$, indicating that vehicle i should load an amount $f \leq V_i$ of load in s and transport it to g. Obviously, f can also be equal to 0, when s represents the initial position of a vehicle and g its first load pick-up location.

We can now define the solution to the mission planning problem as follows:

Definition 3. *A* solution *to a mission planning problem $\langle M, \mathcal{I}, \mathcal{G}, \mathcal{C}, \mathcal{V} \rangle$ with N vehicles is a triple $\langle \Pi, \mathcal{E}, \mathcal{C} \rangle$ where*

- $\Pi = \{\pi_1, \ldots, \pi_p\}$ *is a set of high-level plans which achieve the goals in $\mathcal{G}$;*
- $\mathcal{E} = \{\langle \mathcal{S}_1, \mathcal{D}_1, \mathcal{O}_1 \rangle \ldots \langle \mathcal{S}_N, \mathcal{D}_N, \mathcal{O}_N \rangle\}$ *is a set of trajectory envelopes where*
  - *$\mathcal{S}_i$ is a set of spatial envelopes for the trajectory of vehicle i;*
  - *for every $\langle i, f, s, g \rangle$ in the high-level plan Π, $\mathcal{S}_i$ contains a sequence of spatial polygons $\langle S_1, \ldots, S_n \rangle$ where S_1 contains location s, S_n contains location g, and each S_j spatially overlaps S_{j+1};*
  - *$\mathcal{D}_i$ and $\mathcal{O}_i$ are sets of temporal constraints defining the temporal envelope of the trajectory for vehicle i; these constraints impose that overlapping spatial polygons also overlap in time;*
- *$\mathcal{C} = \mathcal{T} \cup \mathcal{T}_{col}$ is a set of temporal constraints between the start/end timepoints of any pair of spatio-temporal polygons in $\mathcal{E}$; this set contains the constraints $\mathcal{T}$ expressing the initial temporal requirements of the mission planning problem, as well as a set of constraints $\mathcal{T}_{col}$ which ensures that the intersection of spatio-temporal polygons for different vehicles is either only spatial, or only temporal, or neither (i.e., these constraints disallow collisions).*

Trajectory Planning

In order to obtain trajectory envelopes, we first employ a lattice-based planner to generate kinematically feasible paths for the (non-holonomic) vehicles in the mission planning problem. A lattice can be seen as a generalization of a grid: instead of using perpendicular lines, the state-space is discretized by repeating the same primitive set of connecting edges. We start from a set of kinematically acceptable motion primitives which can be repeated over and over to obtain a directed graph. Obviously, the graph need not be completely specified from the start, and can be progressively built during search. The graph is then efficiently explored using deterministic, theoretically sound algorithms. In our case, we chose to rely on the classic A^* (Hart, Nilsson, and Raphael, 1968) for optimal path generation[5], and on one of its most efficient anytime versions, ARA^* (Likhachev, Gordon, and Thrun, 2003), which can provide provable bounds on sub-optimality.

Our approach is inspired by existing lattice-based path planners, as the ones successfully used in real world application by Pivtoraiko, Knepper, and Kelly (2009) and by Urmson, Anhalt, and others (2008).

[5]The resulting path is optimal wrt the choice of the set of primitive motions and to the granularity with which the lattice is built.

Each vertex of the lattice represents a pose of the vehicle in the form $\langle x, y, \theta \rangle$, where x and y are coordinates on a grid of a pre-determined resolution, and $\theta \in \Theta$ is the vehicle orientation, where Θ is the set of pre-selected possible orientations for a specific vehicle model. For instance, in the experimental runs presented in this paper, the grid resolution is always equal to 0.2 meters, Θ is a set of 16 angles, equally spaced between π and $-\pi$, and each vertex is connected to 15 others through pre-calculated, kinematically feasible motion primitives. In our setup, the cost is based on the distance covered by each edge of the lattice, multiplied by a cost factor that penalizes backwards and turning motions.

Using off-line computation, it is possible to speed up the exploration of the lattice in environments with obstacles in two ways. First, as each edge is the instantiation of a pre-calculated motion template, we can pre-compute for each primitive the cells which the vehicle will partially or totally occupy during the motion. This way, obstacle detection can be efficiently performed on-line, by checking the occupancy level of each cell in the grid-partitioned environment. Second, a more informed heuristic function can be pre-computed and stored in a lookup table (Knepper and Kelly, 2006) by saving the minimum cost to connect two poses in a specific range (10 meters, in the experimental runs presented in this paper). This proves to be a much more efficient heuristic than simple Euclidean distance, as it uses the kinematic model of the non-holonomic vehicle to factor in maneuvering costs. Both functions are however admissible, and they entail optimal solutions when used with A^*.

When the environment presents obstacles, a third heuristic function is also used. Regardless of the pose of the vehicle, each cell in the environment is associated with a value represented by the distance from the goal in a 8-connected graph. All three heuristic functions are evaluated when a new vertex of the lattice is expanded, and we always use the higher in value. Clearly, the resulting heuristic function is not admissible in environments with obstacles. This, however, is not a big drawback in practical applications, where our goal is to obtain drivable and kinematically feasible, albeit sub-optimal, paths. Also, in real settings (as the one described below), we preferably employ ARA^* to explore the lattice, in order to speed up the computation, therefore relinquishing optimality anyway.

Recall that a trajectory envelope is defined as a set of temporally constrained spatio-temporal polygons. The starting point for computing the spatial constraints $\mathcal{S}_i$ for vehicle i's trajectory envelope is a path obtained through the path planning strategy outlined above. Then, the computed spatial envelope is used together with the path and the minimum and maximum speeds of the particular vehicle to determine the temporal envelope of the trajectory (i.e., the temporal constraints $\mathcal{D}_i$ and $\mathcal{O}_i$). These two procedures are described in the following paragraphs.

Spatial envelope generation. For each vehicle, waypoints are sampled along the path obtained by the path planning algorithm. The sampling procedure is incremental, and works as follows:

1. select the first two points of the path;
2. build a convex polygon around the vector defined by these two points whose shape is the bounding box of the vehicle, centered in the first point;
3. grow the sides of the polygon outwards, stopping the growth of each side when it intersects with an obstacle or when a threshold on growth has been reached;
4. select two points along the path immediately outside the polygon, and go to step (2).

The resulting sequence of polygons is such that (1) the path is completely covered by polygons, and (2) each polygon intersects the next one[6]. The resulting polygons are used to define the spatial envelope $\mathcal{S}_i$ for each vehicle i. All spatial envelopes are then added to the spatio-temporal constraint database.

Temporal envelope generation. Again starting from the first point along a vehicle's path, the path is traversed to compute the distance covered by the vehicle while traveling in each spatial polygon $S_j \in \mathcal{S}_i$. These distances are used to compute the temporal bounds within which the vehicle can possibly occupy each polygon and each area of polygon intersection. For this computation, we employ two constant speeds $(v_{\min}, v_{\max})$ corresponding to the minimum and maximum desired speeds for the vehicle. For each spatial polygon S_j we thus obtain a pair of bounds $[l_j, u_j]$ restricting the temporal distance between its start and end timepoints (t_j^s, t_j^e), as well as a pair of bounds $[l'_j, u'_j]$ restricting the distance between the end time t_j^e of spatial polygon S_j and the start t_{j+1}^s of spatial polygon S_{j+1}. Together, all these constraints constitute the spatial envelope $\mathcal{D}_i \cup \mathcal{O}_i$ of the trajectory of the i-th vehicle, and are added to the spatio-temporal constraint database.

Trajectory Scheduling

The spatio-temporal polygons generated by the trajectory planning module impose vehicles to be in certain (convex) regions within certain temporal bounds. In order to complete the synthesis of the solution to the mission planning problem, further constraints must be added (the set $\mathcal{T}_{\text{col}}$) in order to prune out of the solution trajectory envelopes in $\mathcal{E}$ those trajectories that lead to collisions. This problem is cast as a CSP whose variables are sets of spatio-temporal polygons which have a non-empty spatial and temporal intersection. The values of these variables are temporal constraints which separate these temporally concurrent, spatially overlapping polygons in time. In other words, the trajectory scheduling module resolves concurrent use of floor space by altering *when* different vehicles occupy spatially overlapping polygons. This results in temporal constraints that disallow the concurrent occupation of overlapping polygons by more than one vehicle at a time.

The reduction of the trajectory scheduling problem to a CSP is inspired by the ESTA precedence-constraint posting algorithm (Cesta, Oddi, and Smith, 2002) for resource scheduling. The algorithm is a CSP-style backtracking search (see algorithm `SolveESTA()`). It starts by collecting all pairs of spatio-temporal polygons that overlap both spatially and temporally (line 3–4). These conflicts are the variables of the CSP, and as usual in CSP search, ordered according to a most-constrained-first variable ordering heuristic (h_{var}) — the rationale being that it is better to fail sooner rather than later so as to prune large parts of the search tree. Once a conflict is chosen, its possible resolvers are identified (line 7). These are values of the CSP's variables, and each is a temporal constraint to be imposed between the pair of spatio-temporal polygons that would eliminate their temporal overlap. Note that since conflicts are pairs of spatio-temporal polygons, there are only two ways to resolve the temporal overlap, namely imposing that the end time of one spatio-temporal polygon is constrained to occur before the start time of the other, or vice-versa. Again as is common practice in constraint-based reasoning, the resolver to attempt first is chosen (line 9) according to a least constraining value ordering heuristic (h_{val}) — the rational being that the value which leaves most options open for future choices should be given precedence. The algorithm then attempts to post the chosen resolving constraint into the spatio-temporal constraint database (line 11). If the underlying STP is still consistent, then the procedure goes on to identify and resolve another conflict through a recursive call (line 14). In case of failure (line 15), the chosen value is retracted from the spatio-temporal constraint database and another value is attempted.

Clearly, the efficiency of the search for resolving constraints depends on how well-informed the value and variable ordering heuristics are. In our specific case, we employ two heuristics which take into account both the temporal and the spatial features of the trajectory scheduling problem. The heuristic h_{var} employed for variable ordering gives preference to the pairs of spatio-temporal polygons that are spatially closer to other conflicting pairs. The idea of this

Function `SolveESTA(`$\mathcal{E}$`)` : success or failure

```
1  static T_col ← ∅
2  repeat
3      conflicts ← {⟨(t_i^s, t_i^e), (t_j^s, t_j^e)⟩ ∈ E :
4                   [l_i^s, l_i^e] ∩ [l_j^s, l_j^e] ≠ ∅ ∧ S_i ∩ S_j ≠ ∅}
5      if conflicts ≠ ∅ then
6          MCS ← Choose(conflicts, h_var)
7          resolvers ← {(t_i^e, t_j^s) : D_i, D_{j≠i} ∈ MCS}
8          while resolvers ≠ ∅ do
9              (t_i^e, t_j^s) ← Choose(resolvers, h_val)
10             resolvers ← resolvers \ (t_i^e, t_j^s)
11             STP ← STP ∪ (0 ≤ t_j^s − t_i^e ≤ ∞)
12             if STP is consistent then
13                 T_col ← T_col ∪ (0 ≤ t_j^s − t_i^e ≤ ∞)
14                 if SolveESTA(E) = failure then
15                     T_col ← T_col \ (0 ≤ t_j^s − t_i^e ≤ ∞)
16                 else return success
17 until conflicts = ∅
```

[6]Polygon intersection is not guaranteed with this procedure, which, however, gives very good results in practice.

heuristic is that conflicts that are "close" to other conflicts are more likely to be the most difficult to solve, as the possible choices for resolving these conflicts will depend on how other conflicts are resolved.

As a value ordering heuristic, we follow the method used by Cesta, Oddi, and Smith (2002), whereby the temporal bounds $\left[l_i^{s/e}, u_i^{s/e}\right]$ and $\left[l_j^{s/e}, u_j^{s/e}\right]$ of the start/end timepoints of the chosen pair of spatio-temporal polygons are analyzed to determine which ordering least restricts the temporal slack of the intervals.

From Envelopes to Vehicle Control

The trajectory scheduling module performs the last step in defining trajectory envelopes that solve the mission planning problem. Every vehicle's control module must at this point select one particular trajectory (i.e., a path and a speed profile) within the vehicle's trajectory envelope to execute. However, it is important to note that the particular trajectory chosen by each vehicle's controller depends on which trajectory other vehicles have chosen, as trajectories of different vehicles are temporally dependent.

The presence of temporal dependencies between trajectories entails that vehicle controllers must communicate their choice to other controllers. This choice can be seen as a set of temporal constraints T_{con}, which is added to the shared constraint database. Here, we leverage an important feature of the STP underlying our constraint database: in a fully propagated and consistent STP, i.e., one in which the bounds $[l_i, u_i]$ of all timepoints t_i have been updated to reflect the constraints, there exist two specific assignments of times to timepoints that are temporally consistent, namely the *earliest time assignment (ET)* and the *latest time assignment (LT)*. The former is obtained by choosing the lower bound l_i for all timepoints, and the latter by choosing the upper bound u_i for all timepoints. Therefore, we can immediately obtain the fastest and slowest speed profiles for all vehicles.

Our vehicle control scheme consists in a model predictive controller (Qin and Badgwell, 2003) which synthesizes control outputs to the vehicle. These outputs enable the vehicle to follow the given trajectory, both with respect to the spatial constraints $\mathcal{S}_i$ and with respect to a particular solution to the temporal constraints in $\{\mathcal{D}_i \cup \mathcal{O}_i\}$. Having selected a particular speed profile, this means that a controller must enter and exit spatial polygons exactly at the times prescribed in the particular speed profile selected for that vehicle (e.g., the fastest speed profile). Whenever these times cannot be achieved, vehicle controllers must revise their trajectories and compute new control outputs. Fortunately, to compute an allocation of times which is different from the ET or LT, it is sufficient to impose one constraint which models the desired allocation of one timepoint, and this can be achieved in polynomial time. As a result of propagation, the new ET allocation will clearly be temporally feasible. Indeed, even more interestingly, numerous alternative, globally consistent speed profiles can be computed before hand, each of which reflects one specific time in which vehicles should enter and exit each spatial polygon on their trajectory.

Experimental Evaluation

We now present an experimental validation of the two central modules of our approach, namely trajectory planning and trajectory scheduling. We validate the modules both qualitatively and quantitatively, with a special focus, in the quantitative analysis, on the performance of the trajectory scheduling algorithm. All test runs were performed in simulation. The kinematic model employed in all the experiments is that of a Linde H50D forklift.

Qualitative Evaluation

A single run in an industrial scenario was performed to qualitatively assess the feasibility of the approach in a realistic setting. For this purpose, we used a real map of an underground mine (courtesy of Atlas-Copco Drilling Machines, see figure 3), where we deployed 7 identical vehicles with pre-assigned tasks. Each task consisted in an initial and final pose for one of the vehicles, in the form $\{\langle x_i, y_i, \theta_i\rangle, \langle x_f, y_f, \theta_f\rangle\}$.

The overall run consisted of three phases. First, our lattice-based path planner generated in parallel individual kinematically feasible paths. Second, the paths were sampled to calculate the spatial envelopes for each vehicle. Assuming that all the forklifts would start moving at the same time, and defining the minimum and maximum desired speed in the tunnels (v_{min},v_{max}), we thus obtained a temporal envelope for each vehicle. The `SolveESTA()` algorithm was then invoked to generate a solution to the mission planning problem. As explained above, the algorithm identified all the conflicts in space and time over the temporally and spatially constrained polygons, and solved them by imposing additional temporal constraints $\mathcal{T}_{col}$.

In this specific run, considering the initial temporal and spacial envelopes for each single vehicle, the scheduler identified three groups of conflicting polygons (shaded in figure 3). Each conflict reflects the fact that, with only the temporal constraints stemming from the desired v_{min} and v_{max} of each vehicle along its nominal path, two or more vehicles would be "allowed" to be in overlapping areas at the same time, if they chose some particular velocity profiles.

The scheduler's solution consisted of 13 temporal constraints. This resulted in revised bounds for each of the spatio-temporal polygons such that in any consistent execution (e.g., the earliest start time, or fastest, execution) vehicles yield to each other appropriately in order to avoid collisions.

Extracting a specific trajectory for execution other than the earliest and latest time trajectories takes about 250 milliseconds. The total time required to generate the scheduled trajectory envelopes was less than 40 seconds: the paths were generated using the ARA^* algorithm, with a cut-off time of 5 seconds, and then used to grow a total of 140 polygons for the 7 vehicles, while trajectory scheduling took less than 34 seconds.

Quantitative Evaluation

To evaluate our approach in a more thorough and quantitative way, we generated a benchmark set of 900 trajectory scheduling problems. On an obstacle free map of

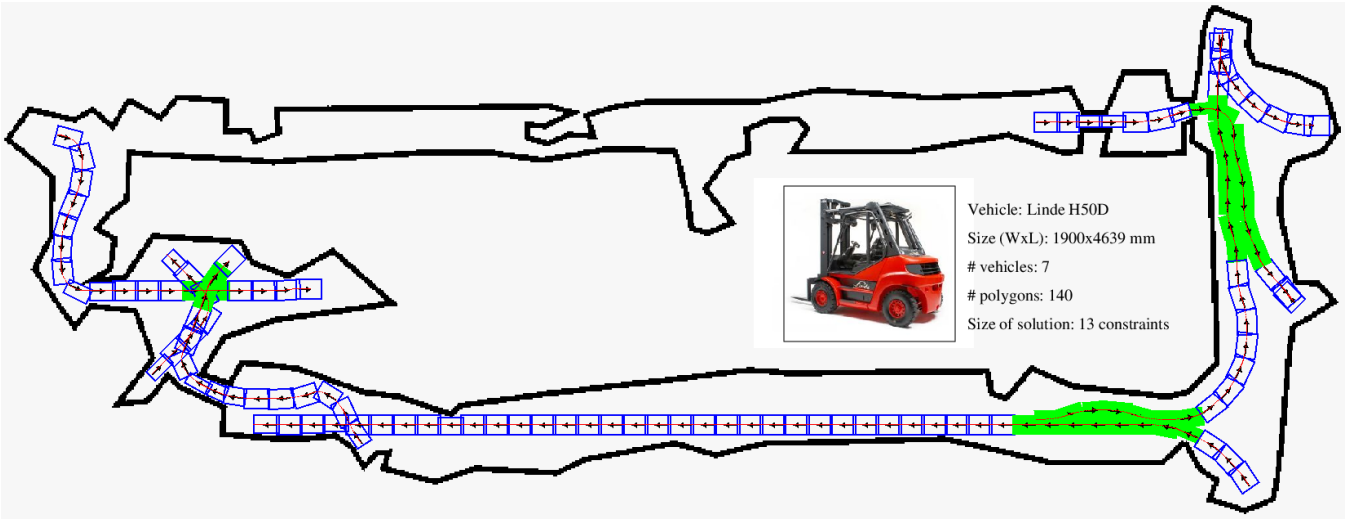


Figure 3: A solution to a mission planning problem involving seven vehicles in an underground mine. Spatio-temporal polygons involved in critical sets during trajectory scheduling are shaded.

width and length of 50 meters, we pre-defined 80 poses $\{\langle x_1, y_1, \theta_1\rangle, \ldots, \langle x_{80}, y_{80}, \theta_{80}\rangle\}$, where the (x, y) coordinates of each pose correspond to one of 10 points spatially distributed on a circle, 40 meters in diameter, and where the orientation θ is one of 8 pre-determined angles. Each pose could be chosen as initial of final pose for a vehicle, with the only constraint that the (x, y) coordinates of the two poses should be different.

The experimental evaluation was performed by defining 9 test sets, each corresponding to an increasing number of vehicles concurrently deployed in the environment, from 2 (the minimum number of vehicles whose spatial and temporal envelopes could generate conflicts) to a maximum of 10. For each set, we performed 100 test runs, as follows. In each run, we randomly chose initial and final poses for the number of vehicles required, only avoiding that two or more vehicles had the same starting or final (x, y) positions. Once generated, the paths were used to obtain the spatial and temporal envelopes with $(v_{\min}, v_{\max}) = (0.05, 15)$ meters per second. In order to make the problems difficult to solve for the scheduler, we also added temporal constraints imposing that the temporal distance between all initial spatio-temporal polygons is zero, thus forcing all vehicles to start moving at the same time. This, combined with a non-zero minimum speed for all vehicles, is what allows some benchmark problems to be unsatisfiable.

Again, we focused on analyzing the trajectory scheduling efficiency, so we measured the time required by the scheduler to find a conflict-free solution for each run, or to identify the problem as unsolvable. The results are shown in figure 4. As expected, scheduling time grows exponentially with the number of vehicles involved.

Two features of these results are interesting. First, note that problem difficulty in this benchmark is somewhat artificially inflated as all vehicles are constrained to operate in an area which is 40-meter diameter circle at roughly the same

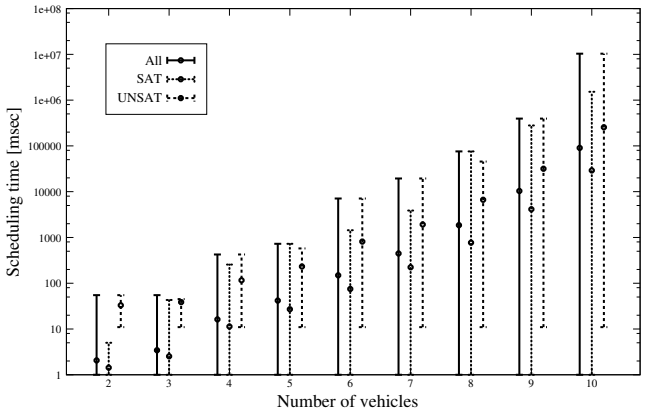


Figure 4: Quantitative evaluation of the trajectory scheduler.

time. Moreover, all vehicles are constrained to start moving at the same time, as all starting spatio-temporal polygons are constrained to occur at the same time and a minimum velocity of zero is not possible. Even under these rather unlikely circumstances, the average resolution time remains under one second up to problems in which we deployed 8 vehicles. Second, recall that the solutions obtained through the scheduling procedure represent a trajectory envelope for each vehicle. Two single trajectories, the earliest time and latest time trajectory, can be extracted for all vehicles in linear time (in the number of polygons). This is, even for the most difficult problem, an operation which takes less than 10 milliseconds. Furthermore, even if one vehicle controller decides it must stray from its current chosen trajectory, the calculation of another trajectory for all other vehicles is also a matter of milliseconds, as it can be done in cubic time. Specifically the most challenging problem of our benchmark contains 94 polygons, and we can extract a new trajectory

for all vehicles in less than 50 milliseconds.

Comparing heuristics. In a comparison against a random choice variable and/or value ordering heuristic, the proposed h_{var} and h_{val} lead to dramatically better performance. We have also compared h_{var} to a heuristic commonly used in scheduling which employs only temporal features of the constraint network to determine the most constrained variable (Cesta, Oddi, and Smith, 2002). Our spatio-temporal heuristic lead to better performance of the backtracking search algorithm, although a complete comparison is necessary to establish whether the effect is due to the particular problem structure of the benchmark.

Conclusions and Future Work

This paper presents a new approach to multiple vehicle coordination in industrial environments. The framework is composed of four different modules for solving logistics problem for AGVs. The modules progressively refine a constraint-based representation of the overall problem, taking into account high-level task planning goals and temporal requirements to ultimately obtain commands for vehicle control. Our approach is engineered in a way that single modules can be used independently, thus providing the flexibility required in industrial settings.

We have focused on the two central modules of our framework, namely trajectory planning and trajectory scheduling. Our main contribution lies in multi-robot system coordination: instead of relying on ad-hoc traffic rules, or predefined priority levels, we used an on-line scheduler to synchronize the movements of the AGVs. Our scheduler allows maximum flexibility, as vehicle trajectories can be globally adapted to exogenous events, control failures, or changed mission requirements. The two modules are evaluated both qualitatively and quantitatively, proving that our approach can be used on-line, and that the results can be immediately employed by low-level controllers.

Our future work will focus on the full development of the remaining two modules, the task planner and a robust model predictive controller, for one or more specific industrial settings. Also, we will explore the use of different heuristics for trajectory scheduling. Finally, we intend to test the full framework and/or parts of it in real industrial scenarios to demonstrate the benefit of our new paradigm in terms of deployability and efficiency.

Acknowledgments. This work is supported by project "Safe Autonomous Navigation" (SAUNA), funded by the Swedish Knowledge Foundation (KKS). The Authors wish to thank Dimiter Driankov, Dimitar Dimitrov and Simone Fratini for their support and useful comments.

References

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A constraint-based method for project scheduling with time windows. *Journal of Heuristics* 8(1):109–136.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49(1-3):61–95.

Desaraju, V., and How, J. 2011. Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*.

Floyd, R. W. 1962. Algorithm 97: Shortest path. *Communication of the ACM* 5:345–348.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Kleiner, A.; Sun, D.; and Meyer-Delius, D. 2011. Armo: Adaptive road map optimization for large robot teams. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*.

Knepper, R. A., and Kelly, A. 2006. High performance state lattice planning using heuristic look-up tables. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*.

Larsson, J.; Appelgren, J.; and Marshall, J. 2010. Next generation system for unmanned lhd operation in underground mines. In *Proc. of the Annual Meeting and Exhibition of the Society for Mining, Metallurgy & Exploration (SME)*.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems* 16.

Luna, R., and Bekris, K. 2011. Efficient and complete centralized multi-robot path planning. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*.

Pivtoraiko, M.; Knepper, R. A.; and Kelly, A. 2009. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics* 26(3):308–333.

Qin, S., and Badgwell, T. 2003. A survey of industrial model predictive control technology. *Control Engineering Practice* 11:733–764.

ter Mors, A. 2011. Conflict-free route planning in dynamic environments. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*.

Thomson, J., and Graham, A. 2011. Efficient scheduling for multiple automated non-holonomic vehicles using a coordinated path planner. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*.

Tsang, E. 1993. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego.

Urmson, C.; Anhalt, J.; et al. 2008. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics* 25(8):425–466.

Wagner, G., and Choset, H. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*.

List of Authors