An Extensible Software Architecture for Composing Motion and Task Planners

Zakary Littlefield, Athanasios Krontiris, Andrew Kimmel, Andrew Dobson, Rahul Shome, and Kostas E. Bekris*

Computer Science Department, Rutgers University, Piscataway, NJ, 08554, USA,
kostas.bekris@cs.rutgers.edu

Abstract. This paper describes a software infrastructure for developing and composing task and motion planners. The functionality of motion planners is well defined and they provide a basic primitive on top of which it is possible to develop planners for addressing higher level tasks. It is more challenging, however, to identify a common interface for task planners, given the variety of challenges that they can be used for. The proposed software platform follows a hierarchical, object-oriented structure and identifies key abstractions that help in integrating new task planners with popular sampling-based motion planners. Examples of use cases that can be implemented within this common software framework include robotics applications such as planning among dynamic obstacles, object manipulation and rearrangement, as well as decentralized motion coordination. The described platform has been used to plan for a Baxter robot rearranging similar objects in an environment in an efficient way.

1 Introduction and Related Work

The basic motion planning problem for kinematic systems is a traditional, well studied in robotics, for which a variety of solutions have been proposed, many of them based on sampling-based algorithms [1]. Such algorithms have also been extended in the context of planning with significant dynamics, planning for high-dimensional challenges involving kinematic chains, or planning under uncertainty. The maturity of the motion planning field also led to the development of software platforms that facilitate the use of such state-of-the-art solutions in different application domains [2, 3, 4, 5].

One of the most interesting challenges, however, relates to integrating such motion planners with task planning, i.e., the high-level reasoning for completing tasks that requires symbolic, combinatorial or discrete planning. Integrating task and motion planning is receiving increasing attention in the related literature recently [6, 7, 8, 9]. The architecture proposed in this paper focuses on providing a reusable, extensible software platform for integrating task with motion planners. The focus is mostly on providing the communication primitives for composing motion planning primitives.

Some examples that relate to this objective include the following:

- *Manipulation:* Multi-modal motion planning [10, 11] and rearrangement of obstacles in the environment [12].

*Zakary Littlefield's work was supported by a NASA Space Technology Research Fellowship. Any conclusions expressed here are of the authors and do not reflect the sponsors' views.

- *State $\times$ Time Planning:* Planning among dynamic obstacles [13]; Sensor-based task planning [14, 15]; Exploration and coverage of an environment [16].
- *Multi-agent Challenges:* Motion coordination [17, 18]; Adversarial challenges, such as pursuit-evasion [19].
- *Task Sequencing:* Multi-goal challenges, including Traveling Salesman Problems and switching goals [20].

There are also many challenges that might involve multiple aspects of the above tasks. For instance, a scenario where multiple manipulators are operating in the same workspace and need to coordinate in order to relocate a large object may require many levels of reasoning. Rather than having users write a single, highly complex task planner to achieve this, it is desirable to allow the composition of task planners instead out of individual modules, allowing for reuse of existing task planning capabilities.

The goal of the proposed infrastructure is to provide a straightforward framework for integrating task and motion planners so that they can be used across multiple application domains. In such a framework, motion planners can be freely exchanged without affecting task-level reasoning, and task planners can be composed in a general, hierarchical manner to solve complex challenges.

While having such a unified structure for task planning is desirable, it is also inherently challenging, primarily because task planning is application specific and there is such a wide variety of application domains. Creating an interface for task planning which can be used for all conceivable applications is infeasible; however, composing task planners in a hierarchical fashion allows for a simple, generalizable interface. This work utilizes abstractions referred to as task specifications and queries to facilitate this interface.

The proposed software infrastructure, referred to as `PRACSYS` is an extension of a previous effort by the authors [21]. In the previous version, the focus was on the introduction of controllers and planners. In this version, integrating planners to solve task planning challenges is the main objective, which has distinctive characteristics in comparison to related software efforts, while at the same time it can be integrated with many of them. `PRACSYS` offers a robust infrastructure for task planning, as well as a control framework unavailable in other motion planning platforms, such as the Open Motion Planning Library (OMPL) [2] and MoveIt! [5]. Other packages such as Gazebo [22] offer simulation capabilities, but they do not offer the planning capabilities of `PRACSYS`. The Reflexxes motion library [23] also focuses on control, generating trajectories in real-time while integrating sensing information, but does not focus on longer-horizon planning components. `PRACSYS` also supports planning over controllers, due to its unique architecture, which is not afforded by the aforementioned packages.

2 General Architecture of `PRACSYS`

`PRACSYS` consists of several different components, which are operating as different processes and which are responsible for different aspects of the overall infrastructure. The Robot Operating System (ROS) is the middle-ware that allows for the different components to communicate with each other through message passing, loads the necessary input parameters and provides compatibility with other software packages [24]. The overall architecture of `PRACSYS` is shown in Figure 1.

The `simulation` module performs the simulation of the physical world, including obstacles, and receives sensing data. It is able to detect whether undesirable collisions occur for different states of the simulated world. Furthermore, it provides a hierarchical tree of control systems, which simulate both the physical robots and other moving systems in the world as well as controllers that operate over them. This hierarchy supports the construction and composition of low-level controllers to perform control, as detailed in the authors' previous work [21]. The purpose of controllers is to perform reactive control give access to the state of the systems under them in the hierarchy and potential access to sensing data. At the lower level of the simulation hierarchy, there are always physical "plants", i.e., robots that receive controls and update their state accordingly. In situations that the software infrastructure is not used to directly plan for a physical system, the `simulation` module takes the role of the ground-truth model of the world. Through message passing, this module can transmit the ground-truth state of the simulated world to the other models of `PRACSYS`, such as the `sensing` and the `planning` nodes.

The `planning` node is responsible for performing the high-level logic of task and motion planning, which is discussed in detail in this paper. `planning` uses an internal representation of the world called a "world model", which internally performs simulation in order to achieve longer-horizon planning, in contrast to the short-horizon controllers of the `simulation` node. It receives the true state of the world from either a ground truth `simulation` node or from the `sensing` node. The `sensing` node can either simulate a sensor given information from the `simulation` node or it can directly communicate with a real world sensor to generate the corresponding data.

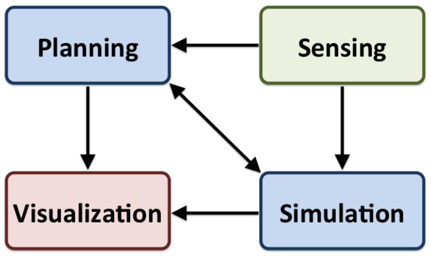


Fig. 1. The major inter-node interactions in `PRACSYS`.

The communication in `PRACSYS` facilitates the interactions between the nodes. The `simulation` node is often responsible for publishing ground truth information for planning and sensing purposes, as well as for visualizing the world. The `planning` node sends computed trajectories and planning structures to `visualization`, and forwards plans to `simulation` for execution. The `visualization` node provides a user interface to validate results and allows for interactive applications. `PRACSYS` can run with or without a `visualization` node. `PRACSYS` also contains a set of core functions which resides in a *utilities* package, which is used by all of the nodes.

3 Integration of Motion and Task Planners

The `planning` node is designed to easily compose planners in a hierarchical fashion. At the top of the hierarchy lies the *planning application*, which can access *task planners*. Task planners address specific high-level tasks and can internally call other task planners. *Motion planners* exist at the lowest level of the hierarchy and have a specific interface. Both task and motion planners are extensions of abstract planners and both have access to *planning modules*, which correspond to useful primitives, such as sampling of states and controls, as well as steering functions. Planners and modules can access the *world model*, which encapsulates the control systems operating in the world.

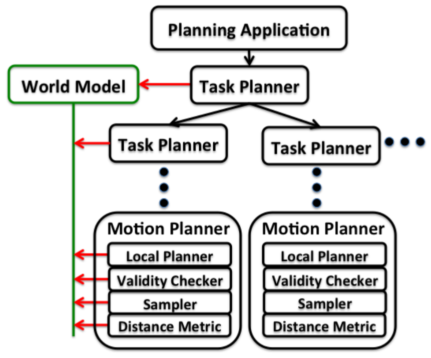


Fig. 2. Hierarchy of planning.

The interactions between the different classes of the `planning` node are illustrated in Figure 2. The arrows in the figure imply that the class from where the arrow originates has access to and calls functions of the class the arrow points towards. Task planners can be composed to perform more complex tasks; for instance, a navigation task and manipulation task can be used by a higher-level task planner to perform a retrieval task using a mobile manipulator. The leaves of the planning hierarchy are always motion planners, which actually select the controls that the underlying control systems of the world model will use. To do so, they make use of the planning modules, which provide them access to the world model, which is detailed in Section 3.1. The interface between planning modules and task planners with the world model are via direct function calls. For instance, the world model can be queried to check if a state results in an undesirable collision. It also provides access to sensing, which is described in Section 3.2.

The planners interface one with another through the use of *task specifications* and *queries*, both detailed in Section 3.3. Specifications are used to inform lower-level planners what are the parameters of the task to be solved,. Two planners can be stacked one on top of the other, only if the lower level one addresses task specifications requested by the higher level one. For example, motion planning specifications are of a specific type and inform a motion planner what low-level modules to use when building the planning data structure (e.g., a roadmap or a tree), and identify a stopping criterion. Queries are similarly passed from higher-level to lower-level planners. The latter are responsible to fill the query with the requested information and return it to the higher-level planner. For example, a motion planning query contains a start state-goal state pair, and the motion planner is responsible for returning the feasible trajectory and plan which brings the system from the start to the goal.

3.1 World Model and Simulator

In order to properly perform planning, the evolution of the environment and the control systems in it need to be modeled. This is encompassed by the *world model* abstraction. The *world model* includes an internal simulator, which can be structured to model the controllers and underlying dynamics of agents. The simulators employed by the world model are the same found in the `simulation` node. This internal simulator provides the capability of collision checking and potentially simulating complex physical phenomena through the use of the Bullet physics engine [25].

The *world model* abstraction allows for the definition of a *state space* and a *control space* used by the planning process. The *state space* represents the necessary information in order to fully specify a snapshot of the world in terms of the relevant kinematic and dynamic parameters of the involved moving or movable systems. The *control space* provides the input to the simulator that allows it to modify the state over time. These abstractions provide the ability to store states and controls so that the world model can be placed into a desired configuration from a planner or a planning module.

These definitions also allow diverse options for task planners to create different *planning contexts* for the motion planners to plan in. A *planning context* consists of different divisions of the underlying simulator's *state* and *control space*, and is comprised of three subspaces: a *planning space*, *object space*, and *inactive space* as shown in Figure 3. The *planning space* is the subset of a space that a motion planner would plan over, such as the state space of a robot manipulator. The *object space* consists of all systems that are not directly "controlled" by the planner, but still are considered for collisions and may move during the simulation. A task planner can change the state of such systems (e.g., when an object is grasped and moves according the motions of a manipulator) and inform the *world model*, or they can have their own controllers that handle their evolution over time (e.g., in the case of dynamic obstacles). Finally, the *inactive space* consists of all the systems that may be present in the simulation but are not considered by the planning process. These may be used by other motion planners in the task planning hierarchy.

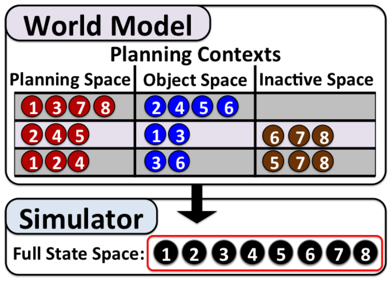


Fig. 3. An example of different planning contexts that can be created. Each planning context consists of a *planning space*, *object space*, and an *inactive space*. Each of these spaces has a different meaning to the task planners in PRACSYS.

In most situations, these spaces are direct subsets of the full state space. Nevertheless, the true planning space may be difficult for motion planners to directly work with. Consider the case of physically simulated systems, where a car-like robot is composed of a chassis and four wheels. In the physics engine, 60 DOFs will be needed to keep track of the parameters of the involved 5 rigid bodies. Nevertheless, most planners would operate over a lower-dimensional projection of that planning space. In these situations, an *embedded space* can be used. This space transforms a higher dimensional space into a lower dimensional given a proper mapping function.

3.2 Integration of Sensing with Planning

Sensing is an important aspect of the PRACSYS framework, and for many applications it plays a vital role in determining the planning contexts, e.g., when a robot uses a sensor to detect the other moving systems in the scene. The sensing framework is comprised of a set of extendable primitives, through which a wide variety of sensing contexts is supported. An example of sensing primitives, along with an example interaction with the simulation node, is shown in Figure 4. The primitives are as follows:

- *Sensors:* They perceive the state of the world. Sensors can either be simulated, such as detecting the configurations or geometries of objects in the simulation, or they can represent actual physical sensors, such as a RGB-D camera generating point cloud data. This is possible through the use of the ROS architecture, since a “sensor” class can simply use external communication through ROS topics and services to query its physical counterparts. A sensor has an individual update frequency, which represents how often it measures the world, and consequently updating its representation.

- *Sensing Information:* This primitive represents how a controller uses the information derived from the sensors’ measurements. A controller may have one or more Sensing Infos (SI), each of which has pointers to individual sensors, which it uses to construct specific information for use by the controller. For example, a controller might need to reason about the proximity of other objects in the world, and thus would need to access both a configuration sensor and a geometry sensor to build such information.
- *Sensing Model:* It owns all the sensors and manages the interactions between external nodes, such as simulation, with the other sensing primitives. It is responsible for checking when sensors need to fire (which corresponds to sensors taking a measurement), as each sensor has a specific firing frequency, as well as checking how often each SI calculates new information.

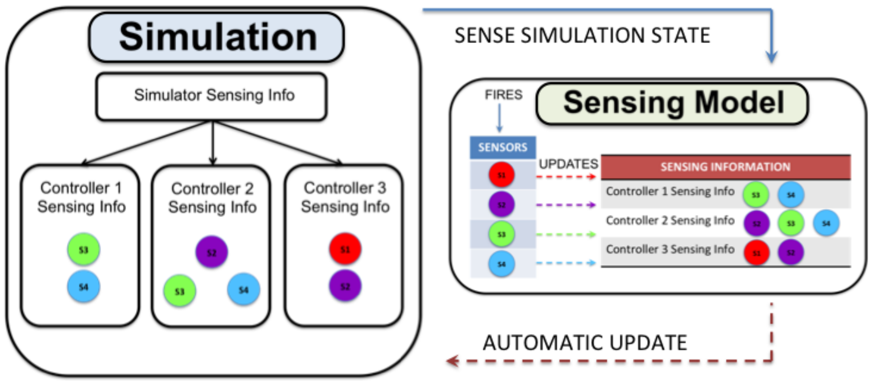

Simulation
Simulator Sensing Info
Controller 1 Sensing Info
Controller 2 Sensing Info
Controller 3 Sensing Info
SENSE SIMULATION STATE
FIRES
Sensing Model
SENSORS
UPDATES
SENSING INFORMATION
Controller 1 Sensing Info
Controller 2 Sensing Info
Controller 3 Sensing Info
AUTOMATIC UPDATE

Fig. 4. An example sensing framework which interacts with the simulation node. Each controller in the simulation has sensing information, with the simulator owning a concatenation of all controller’s sensing information. The simulation informs the sensing model that it has a new simulation state, with which the model fires any sensors that are ready to measure. The simulation is automatically updated by any new sensor readings in the corresponding sensing infos.

Sensing is seamlessly integrated into planning through the use of the world model, which owns a simulation tree and thus inherently can update SIs after calling the sensing model. Planning contexts, which determine which parts of the simulation tree are active, also determine which SIs are actively updating. Since the sensing model must be informed that it is time to sense, task planners have full control over the information that sensing reason overs, as well as how to use the information obtained from sensing.

3.3 Queries and Specifications

This section describes the task specifications and queries that are used as the interface between task planners and motion planners. Specifications describe the problem that a planner lower in the hierarchy needs to solve. Queries give the initial and final conditions of the problem and will be returned with the answer constructed by the planner below. Figure 5 depicts these exchanges between the planners through appropriate interface.

The higher level planner, a task planner, builds a problem specification for the lower level planner, which can be either a task or a motion planner. The problem specification contains information that a planner needs in order to set up and solve a problem, which generally allows some preprocessing to occur. The information that is given in the specification by the higher level planner has to match with the information that is expected by the lower level planner. The problem specification may define different modules, such as local planners, validity checkers, samplers, distance metrics and/or stopping criteria. These modules are mandatory when the planner below is a motion planner and the corresponding problem specification is called a motion planning specification. Depending upon the application, a motion planner can receive additional information like seed states for the search structure it maintains, commonly a tree or a graph in the case of sampling-based motion planning.

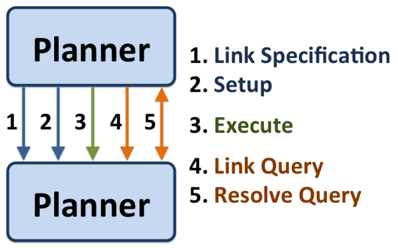


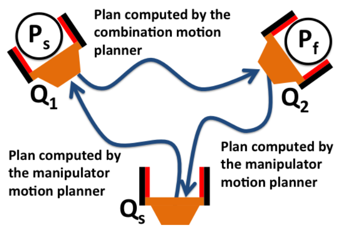
Fig. 5. The interface between planners in the planner hierarchy. First, a planner must receive a planning specification from its parent planner (1). Then, the planner can setup its internal data structures using that specification (2). Finally, after allowing the planner execution time (3), queries can be linked and resolved (4,5).

Once a problem has been already set up according to the problem specification, a query is the request made to a planner to solve a specific instance of that problem. The query is the last interface between the planners in the different levels. A typical query includes the initial and the final configuration for the problem instance. If the planner below is also a task planner, then the query passed to this task planner might be used to construct a new query for underlying planners. In the case of a motion planning query, the query contains the initial state of the system and the goal state(s) that the system has to reach. It then returns the path (sequence of states) and the plan (sequence of controls) that will bring the system to the goal. Similar to problem specifications, the information contained in the query has to correspond to what the lower level planner expects.

4 Use-Cases

4.1 Rearrangement Using Baxter

One important property of this framework is the ability to stack planners in order to reuse them (both task and motion planners). The rearrangement manipulation framework, uses a stack of two task planners and two motion planners (Figure 7). In this problem a robotic arm is used to rearrange geometrically similar and interchangeable objects. The following bottom-up description showcases the reusability of the components.



P_s
P_f
Q_1
Q_2
Q_s
Plan computed by the combination motion planner
Plan computed by the manipulator motion planner
Plan computed by the manipulator motion planner

Fig. 6. Each motion of the manipulator (blue arrows) correspond to a plan computed by a motion planner. The compilation of these individual steps result to move an object to a different pose.

The problem of moving a manipulator can be solved using a motion planner (e.g. PRM*). A typical motion planner builds a data structure (e.g., a graph) and then computes a path on this data structure to connect the start to the goal configuration. The motion planner is able to plan either for transit or transfer motions. The transfer motion planner plans in a space comprising of the state space of the manipulator as the active space and the object it is grasping as the object space, whereas the transit motion planner only deals with the manipulator, which is not grasping any object. All other objects form the inactive space. The use of the two motion planners is shown in Figure 7. The framework can specify the different state spaces by using the planning context, explained earlier in the paper (Fig. 3). The calls to the two motion planners differ because of the different planning modules that are defined in the corresponding specifications.

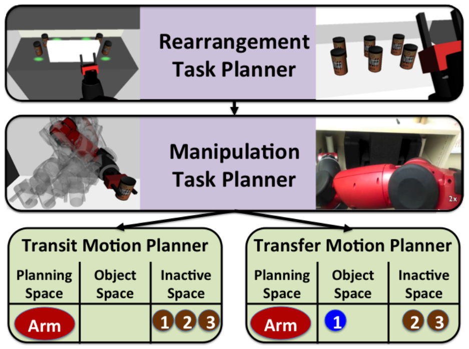


Fig. 7. The hierarchy of planners used in the rearrangement problem.

A manipulation task planner can utilize these motion planners in order to solve a higher level task, which is to use a manipulator to move an object from an initial pose to a final pose. Nevertheless, in order to achieve this task, the planner needs to solve the transit to the first pose, transfer to the second pose with the object in hand and transit back to a safe position (Fig. 6). The problem specification for this planner will have information about the modules that can be used, the stopping criteria for the algorithm, extra information that the manipulator needs to be able to grasp the objects, but most importantly the different planning contexts that the planner needs for the completion of the task. The query for the manipulation planner will request for a path and a plan that will move an object from an initial pose to a target pose. The manipulation task planner given the information from its input problem specification can build the problem specifications for the motion planners.

The high level task is the rearrangement of multiple objects using the manipulator. The manipulator task planner resides under the rearrangement task planner (Figure 7). After setting up the problem specification of the manipulation task planner, the rearrangement task planner can request from the manipulation task planner the computed plan for moving the object from one pose to another. The query comprises of the initial pose of an object as the start state and the final pose of that object as the goal state of the problem. The rearrangement task planner reasons about moving objects between different poses in order to solve the rearrangement problem. This higher level task planner combines all the plans that will rearrange the objects from the initial to the final arrangement. The final plan can be transmitted using ROS messages to a real system, such as Baxter, in order to execute it.

4.2 Planning Among Dynamic Obstacles

One of the main uses for the world model abstraction is using controllers in the planning loop. In this use case, the behavior of dynamic obstacles is defined through the use of a controller. Then, the goal is to find a motion plan for a simple car-like system among

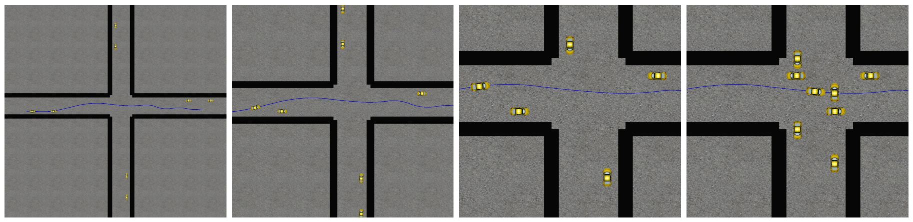


Fig. 8. Snapshots of the dynamic obstacle use case executing the plan found by the motion planner. The problem is to move from the leftmost lane and pass the car in front of it. The screen shots are progressively zoomed in toward the intersection where many cars are entering at the same time. The path of the selected car is shown in the image.

these moving obstacles. Here, a simple kinematic model is used to simulate cars moving in straight lines, but can be modeled more accurately given more complex controllers.

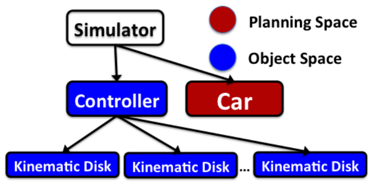


Fig. 9. The world model structure for this example. The car's state space is the only part considered for the planning space, while all of the dynamic obstacles are considered objects that the controller handles, but the motion planner doesn't need to know about.

Figure 9 details the organization of the world model into its planning space and object space. Since the motion planner doesn't need to explicitly model the movement of the moving obstacles, they can be considered for collisions only, and the controller will control the evolution of their states. There is no need to include an inactive space for this task because there is only one planning context that needs to be considered, and the task is a simple motion planning problem. The only requirement for this planning context is that the state of these obstacles is stored during the motion planner's execution. This is done by using a simple embedded space, which for correctness maintains the full state of the world model's internal simulator, but hides it from the motion planner.

The choice of the controller for the obstacles is arbitrary in this setup. Anything that understands how to control the moving obstacles can be placed here, thereby allowing many different behaviors to be modeled. For this experiment, the controllers for the dynamic obstacles are known apriori, therefore the true controller exists in both the ground-truth simulation and the planning's world model.

4.3 Decentralized Multi-Robot Coordination

Scalability is an important requirement, and `PRACSYS` aims towards this along two directions: (a) the number of agents that can be modeled on a single machine, and (b) the number of communicating processes the architecture is able to support. To test the local scalability of `PRACSYS`, an experiment with 250 agents attempting to swap places with one another was used, as shown in Figure 10. On a single computer with a 3.1 GHz Intel i3 processor and 8GB of RAM, this experiment was capable of running in real time at 50 frames-per-second, which also includes the overhead of each individual robot running a local collision avoidance technique while utilizing a proximity sensor.

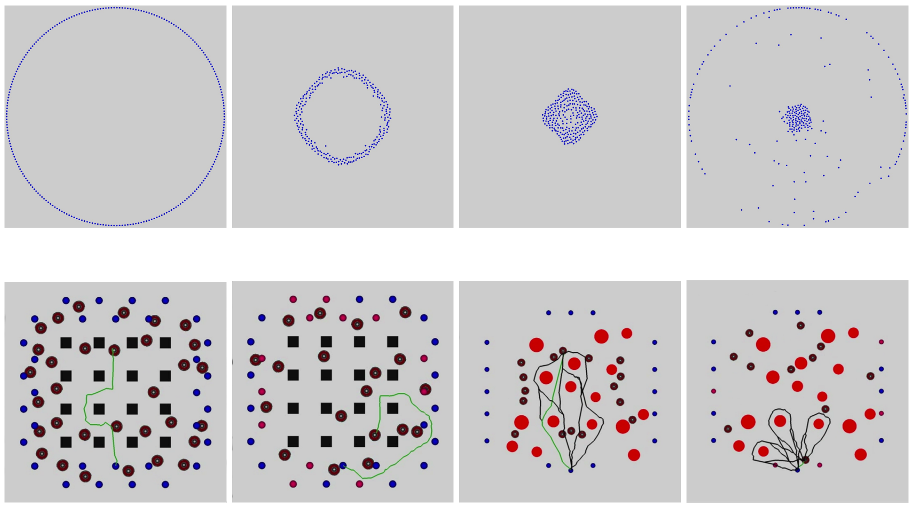


Fig. 10. (Top Row) Snapshots of the local scalability experiment running 250 agents using sensing and local collision avoidance techniques. (Bottom Row) Snapshots of decentralized multi-agent coordination experiments in two different environments.

Given this collision avoidance capability, a decentralized multi-robot coordination challenge was constructed, which utilized every component of `PRACSYS`. The challenge involved multiple robots moving in highly-constraining scenarios with each agent attempting to reach some goal location. This required each robot to operate in a replanning framework, where each agent individually sensed its environment and queried its own planning node to recompute a solution path every 0.5 seconds. The overall structure of the experiment is shown in Figure 11. Beginning at the simulation node, a hierarchical system tree was used, where one branch consisted of: *consumer controller* over a *velocity obstacle controller* over a holonomic *disk plant*. The consumer controller translates the trajectories from planning into relevant controls for the velocity obstacle controller [26], which generates collision-free controls. For some time t during the experiment, each disk agent sends an update of the simulation state to planning - this involved the agent utilizing sensing, along with generating a query asking planning to compute a solution trajectory to some goal location. While the planning node answers this new query, the robot begins executing its trajectory at t to reach a state at $t+1$.

Each robot runs its own individual planning node, where: given an updated state of the world (i.e., through sensing on simulation) at some time t, the task planner generates an appropriate *planning context* to represent the *world model*, and uses the *sensing information* to construct a *planning query* containing the proximity information of other agents (i.e., the sensed agents' positions and velocities). This query informs the motion coordination task planner (MCTP) which parts of the environment were heavily congested. The MCTP then uses the trajectory at t and simulates where the robot would end up at $t+1$. It uses this newly predicted state, changes the motion planner's *problem specification* so as to compute a trajectory from $t+1$ to $t+2$ that would minimize conflicts with other robots. Once such a solution trajectory is computed by the motion planner, this trajectory is sent back to the simulation side via a ROS topic.

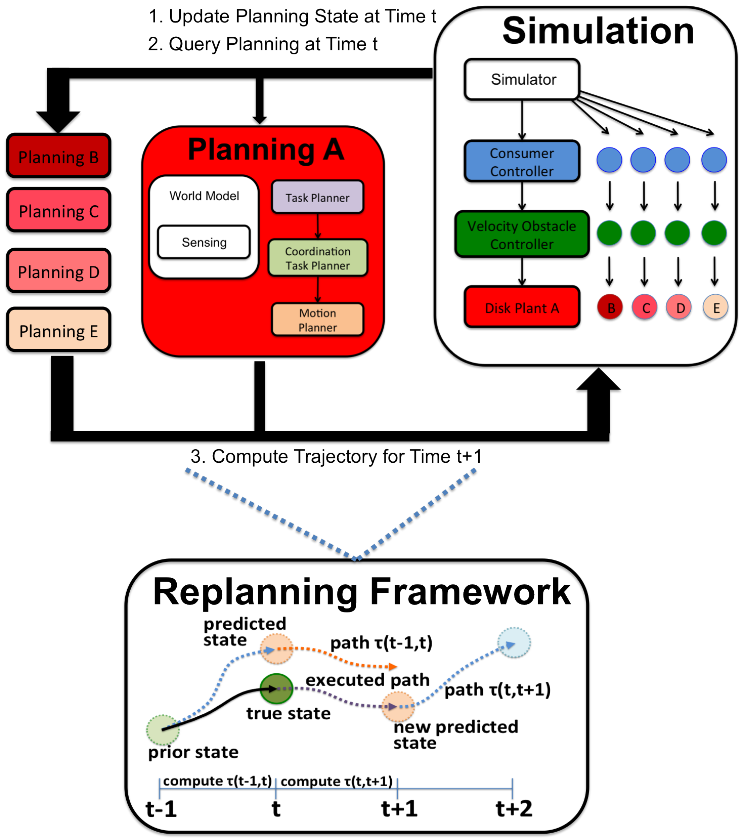


Fig. 11. The interaction between planning and simulation for a decentralized, multi-agent robot coordination challenge.

5 Discussion

This paper describes a software infrastructure for integrating task and motion planners. By taking advantage of the hierarchical nature of real-world tasks, the creation of hierarchical task planners makes software construction natural. PRACSYS provides this infrastructure, along with the necessary modules of a simulator, motion planners, and sensing primitives.

The infrastructure of PRACSYS is a useful tool for developing new types of task planners. The interaction with other software packages, such as with OMPL and Gazebo [2, 22], can result in even more powerful solutions. OMPL provides many different motion planning algorithms, which the community has used in many different applications. The addition of a task planning infrastructure makes OMPL and any other motion planning package extendable and more able to address more complex challenges.

Bibliography

[1] Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion. The MIT Press (2005)

[2] Şucan, I.A., Moll, M., Kavraki, L.E.: The Open Motion Planning Library. IEEE Robotics and Automation Magazine **19**(4) (December 2012) 72–82

[3] Plaku, E., Bekris, K.E., Kavraki, L.E.: ”oops for motion planning: An online open-source programming system,”. In: International Conference on Robotics and Automation (ICRA), Rome, Italy (2007) 3711–3716.

[4] Diankov, R., Kuffner, J.J.: OpenRAVE: A Planning Architecture for Autonomous Robotics. Technical report, CMU-RI-TR-08-34, The Robotics Institute, CMU (2008)

[5] Sucan, I., Chitta, S.: MoveIt! http://moveit.ros.org

[6] Hauser, K., Latombe, J.C.: Integrating task and PRM motion planning: Dealing with many infeasible motion planning queries, ICAPS Workshop on Bridging the Gap Between Task and Motion Planning (2009)

[7] Koenig, S.: Creating a Uniform Framework for Task and Motion Planning: A Case for Incremental Heuristic Search, ICAPS Works. on Action and Motion Planning (2010)

[8] Lozano-Perez, T., Kaebling, L.: Integrated Task and Motion Planning in Belief Space (2013)

[9] Nedunuri, S., Prabhu, S., Moll, M., Chaudhuri, S., Kavraki, L.E.: SMT-Based Synthesis of Integrated Task and Motion Plans for Mobile Manipulation. In: ICRA. (2014)

[10] Hauser, K., Ng-Thow-Hing, V.: Randomized Multi-Modal Motion Planning for a Humanoid Robot Manipulation Task. IJRR (2011)

[11] Kaelbling, L., Lozano-Pérez, T.: Integrated Robot Task and Motion Planning in the Now. CSAIL Technical Report (2012)

[12] Stilman, M., Kuffner, J.J.: Planning Among Movable Obstacles with Artificial Constraints. In: WAFR. (2006)

[13] Ayan, N.F., Kuter, U., Yaman, F., Goldman, R.P.: HOTRiDE: Hierarchical ordered task replanning in dynamic environments. In: ICAPS Workshop on Planning and Plan Execution for Real-World Systems. (2007)

[14] Gaschler, A., Petrick, R.P., Kröger, T., Knoll, A., Khatib, O.: Robot task planning with contingencies for run-time sensing. In: ICRA Workshop on Combining Task and Motion Planning. (2013)

[15] Olawsky, D., Krebsbach, K., Gini, M.: An analysis of sensor-based task planning. Technical report (1995)

[16] Bhattacharya, S., Michael, N., Kumar, V.: Distributed coverage and exploration in unknown non-convex environments. In: DARS. Springer (2013) 61–75

[17] Bekris, K.E., Tsianos, K.I., Kavraki, L.E.: Safe and distributed kinodynamic replanning for vehicular networks. Mobile Networks and Applications **14**(3) (2009) 292–308

[18] Marino, A., Parker, L.E., Antonelli, G., Caccavale, F.: A decentralized architecture for multi-robot systems based on the null-space-behavioral control with application to multi-robot border patrolling. Journal of Intelligent & Robotic Systems **71**(3-4) (2013) 423–444

[19] Stiffler, N.M., O'Kane, J.M.: A Sampling Based Algorithm for Multi-Robot Visibility-Based Pursuit-Evasion, IEEE Intl. Conf. on Intelligent Robots and Systems (2014)

[20] Saha, M., Sanchez-Ante, G., Latombe, J.C., Roughgarden, T.: Planning multi-goal tours for robot arms. Int. J. Robotics Research **25**(3) (March 2006) 207–223

[21] Kimmel, A., Dobson, A., Littlefield, Z., Krontiris, A., Marble, J., Bekris, K.E.: PRACSYS: An Extensible Architecture for Composing Motion Controllers and Planners. In: SIMPAR, Tsukuba, Japan (11 2012)

[22] Koenig, N. and Hsu, J. and Dolha, M. - Willow Garage, Gazebo: http://gazebosim.org/

[23] Kröger, T.: Opening the door to new sensor-based robot applications—The Reflexxes Motion Libraries. In: ICRA. (2011)

[24] Willow Garage, Robot Operating System (ROS): http://www.ros.org/wiki/

[25] Bullet Physics Engine. http://bulletphysics.org

[26] van den Berg, J., Lin, M., Manocha, D.: Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In: IEEE ICRA. (2008)