Executing Underspecified Actions in Real World Based on Online Projection

Gayane Kazhoyan and Michael Beetz*
{kazhoyan, beetz}@cs.uni-bremen.de

***Abstract*— Plan execution on real robots in realistic environments is underdetermined and often leads to failures. The choice of action parameterization is crucial for task success. In this paper, we present a mechanism for a robot that is acting in a real-world environment to think ahead of time with fast plan projection and, thereby, choose action parameterizations that are predicted to lead to successful execution. For finding causal relationships between action parameterizations and task success, we provide the robot with means for plan introspection and propose a systematic and hierarchical plan structure to support that. We evaluate our approach by showing how a PR2 robot, when equipped with the proposed system, is able to choose action parameterizations that increase task execution success rates and overall performance of fetch and place actions in a real world setting.**

I. Introduction

There have been remarkable demonstrations of autonomous mobile manipulation robots performing everyday activities such as folding clothes [1] or washing dishes [2]. However, the robot control programs for executing these tasks are only applicable in the specific settings that they are implemented for. To enable the robots escape laboratory settings and enter unstructured environments such as human households, they need to be able to autonomously manipulate a big variety of objects in a multitude of task contexts, while dealing with differences in the environments and constant failures due to inaccuracies in sensors, actuators and the world representation.

To generalize robot control programs towards different objects, tasks, environments and robot platforms, we introduced the concept of *entity descriptions* [3]. These are abstract underspecified symbolic descriptions of task-relevant entities (objects, locations, actions, etc.) that are being grounded during execution into robot's environment through perception and reasoning. During grounding they are augmented with symbolic and subsymbolic data that specializes them to the environment at hand. Here is an example action description:

```
(an action (type transporting)
           (object (an object (type spoon)))
           (target (a location (right-of bowl-1))))
```

which gets augmented with motion parameters such as where to look for the object, with which arm to transport, with which trajectories, etc.:

```
(an action (type transporting)
           (object (an object (type spoon)))
           (target (a location (right-of bowl-1)))
           (search-location location-in-drawer)
           (arm right)
           (grasp-trajectory pose-1 pose-2 ...)
           ...)
```

*The authors are with the Institute for Artificial Intelligence, University of Bremen, Germany.

There can be different groundings for the same entity description, which result in different outcomes, including a variety of failures that can happen. For example, for transporting an object from A to B, the chosen search location, grasp pose, robot base locations, arm to use, grasping force, placing location etc. determine if the action will be successful or, on the contrary, if the object will slip out, be out of reach, or if the trajectory will result in the robot colliding with the environment or knocking objects over (see Figure 1).

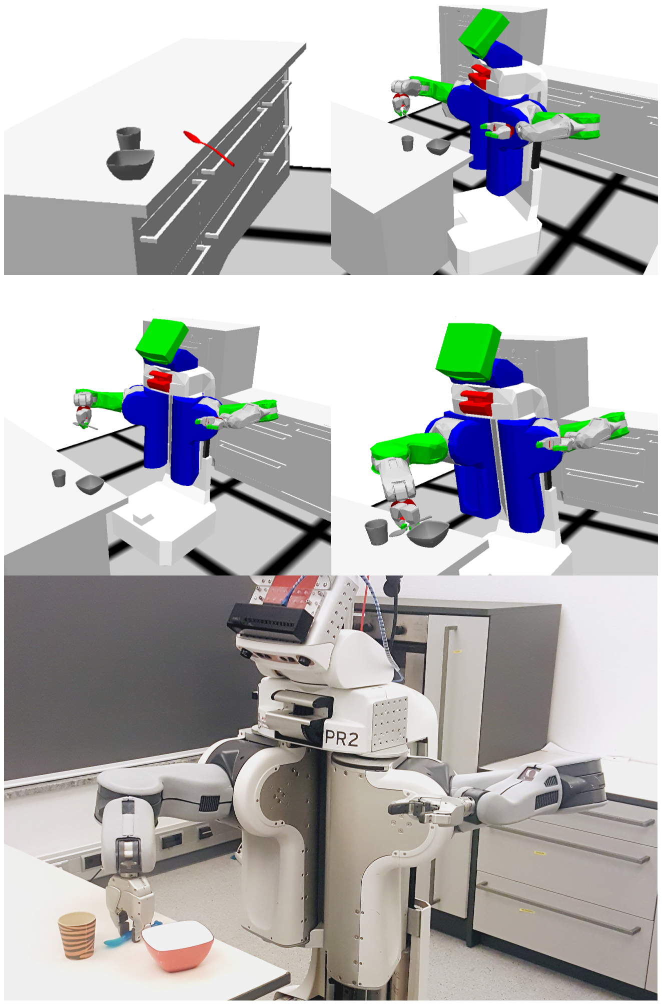

Fig. 1. Projecting a plan for the *"put a spoon right of the bowl"* action:
(top-left) step 1: choose placement location – pose unstable,
(top-right) step 2: choose another pose, navigate – base in collision,
(middle-left) step 3: relocate, try placing – placement pose unreachable,
(middle-right) step 4: relocate and try again – successful,
(bottom) step 5: execute chosen parameterization on the real robot.

Robot control programs written with entity descriptions profit from a clean separation between the control flow and decision making. The control flow specifies the ordering of actions and failure handling behaviors. Decision making performs the reasoning necessary to find action parameterizations that lead to successful task execution. Introducing entity descriptions into programs makes them more scalable and generalizable towards different contexts as well as more compact and readable. However, the effort of specializing the program to the specific execution environment is thereby shifted from the programmer onto the robot itself.

Consider a robot setting a table in a human household environment. To successfully perform the task, the robot might need to answer a question such as: "If I grasp the cup from the handle with my left arm, will I be able to place it at the designated location without having to regrasp?" and based on the answers choose the arm and grasp pose to use. To make this possible, the robot requires foresight. We present an approach and an implemented system[1] that can be used by a robot at runtime to evaluate by the means of fast plan projection how well a plan with a particular parameterization will perform. The parameters of actions can be optimized by executing multiple runs of the same part of the plan with different parameterizations in the projection environment and choosing the best ones according to a cost function. The cost function can be based on the number of occurred failures, on lengths of trajectories, etc. Projection results are easily integrated into the robot's plan to execute the optimized part of the plan in the real world right away.

To enable the robot to find causal relationships between action parameterizations and task success, i.e. to infer which parameters resulted in successful execution and which triggered failures, and to be able to measure execution performance, the robot needs to have means for introspection. To implement that, we store plan-relevant information during real-world and projected execution into a data structure and provide the robot or its programmer with an interface for querying and reasoning with this knowledge. Plans have to be well-structured such that relevant information — action outcomes, failures that happened, parameterizations used, order of action execution etc. — is available and easily retrievable to support introspection. We propose systematic and hierarchical plan structure, based on the *entity descriptions* concept [3], and demonstrate how such a plan structure is advantageous for introspection on the example of fetch and place plans.

The novel contributions of this paper are, thus, as follows:

- an approach to structure real-world robot plans that enables easy introspection, and an implementation of fetch and place plans that have such a structure;
- means to generate different behaviors for the same symbolically-represented mobile manipulation action through sampling over parameter distributions;
- mechanisms to incorporate projection-based inference of best action parameters online into the robot executive.

[1]Open-source code and tutorials: `http://cram-system.org`

We evaluate our approach by showing how a PR2 robot is able to choose action parameterizations that increase task execution success rates and overall performance of fetch and place actions in a real world setting by using our system.

II. Related Work

In classical AI, *symbolic* plan projection is applied to predict the future state of the world. Projection is considered on the basis of axiomatized models of actions, which are atomic entities that have preconditions and effects. For example, an atomic grasping action is assumed to have an effect of an object necessarily being in hand after its execution, if certain preconditions have been met. State-space planners, such as STRIPS [4] and more recent HTN-based planners such as SHOP2 [5], search through state transition systems with atomic transitions to find a sequence of actions, which is predicted to lead to the goal state. These kind of approaches work very well in constrained domains. However, in the domain of mobile manipulation for realistic environments, there is a large number of action parameters and most of them are subsymbolic. Unfortunately, classical AI planners cannot handle such complex domains. As a workaround, they abstract away from motions. However, in real world, grasping trajectories, reachability, occlusions, world dynamics etc. are crucial for successful action execution. In our system, we consider all of the above-mentioned and project our plans all the way down to the low-level motion parameters.

To enable classical planners to work in continuous domains, one often combines them with motion planners. In task and motion planning, one can solve a planning problem in a given environment by using an IK solver and a collision detection engine [6], [7], [8]. With our projection mechanism, the robot can solve the task of how to move its body to accomplish a task successfully and use semantic reasoning to infer that. Task being successful can include not having collision or reachability failures, but we are not limited to only motion planning. For example, we can infer where to place an object such that it does not fall of the supporting surface, where to place the robot such that it can see the object with no occlusions in the way, where to direct the robot's gaze to ensure successful perception etc. Additionally, we can execute different runs of the same action by varying any of its symbolic or subsymbolic parameters, including the standard motion planning parameters such as which grasp pose to use, but also semantic parameters, e.g., in which container to look for the object.

As opposed to deterministic action planners, in classical AI prediction of the future has also been considered based on probabilistic methods. One of the first works in that direction is by Hanks [9], where he argues that the classical AI approach of constraining the search space of possible outcomes by simplifying the world state and action representations may not generate accurate enough projection results to be practically applicable. Instead, he suggests to consider comprehensive world and action representations but restrict the search space to only "important" or "significant" outcomes. Continuing this line of work, Beetz and McDermott [10] present a plan revision technique that improves the behavior of agents by eliminating probable execution failures. They apply plan transformation rules to forestall the most probable failures based on running a small number of execution samples in projection. The work demonstrates advanced techniques but the application domain of the system is a delivery robot in a 2D grid world, whereas in the real world it is very difficult to construct a realistic probabilistic model of robot's actions and their effects.

In robotics, full-fledged dynamics simulators have been used to improve execution. Rockel et al. [11] show a system where simulation is integrated into the planner, such that the latter can choose the appropriate action and parameters based on simulation. This allows the robot to learn a new skill such as balancing an object on a tray. Kunze et al. [12] present a temporal projection system that translates naive physics problems into parameterized simulation tasks with support of first-order representation reasoning over the execution results. With this system the robot can estimate parameters of actions, e.g., for manipulating an egg. Abelha et al. [13] use a simulator to estimate how a particular tool performs in a given task: they wary the parameters of the action of using a tool to estimate the best parameterization based on a "task function". The difference between the aforementioned works and our approach is that they concentrate on simulations of short time span tasks, whereas our approach projects over multiple plan steps and can infer a full set of parameters at once, e.g., for a complete mobile fetch and place sequence including opening / closing containers. From a practical perspective, traditional simulation-based approaches are computationally expensive and have a low real-time factor, whereas our projection is very fast w.r.t. the pace of action execution (see Section VI). Finally, using simulation for reasoning requires to implement the data exchange between the planner and the simulator, whereas we can project any code segment to infer any parameters with no extra configuration effort.

The closest related work that deals with large time span plan projection that goes all the way down to low-level motion parameters is by Mösenlechner et al. [14]. The system described in [14] can be used for offline plan projection and manual introspection. In this paper, we equipped the robot with a plan projection mechanism such that it can optimize its own plans autonomously by running multiple runs of projection during execution and integrating results of projection-based reasoning back into the executive. Additionally, in order to generate different variations of the same plan for choosing parameters that are likely to succeed, the plans are written with underspecified action descriptions, which allow for sampling from the distributions of any of their symbolic or subsymbolic parameters. Finally, we project plans that run in the real world, which are of much higher complexity than those only used in simulation: real-world plans contain concurrent behaviors to monitor the environment and react to changes therein as well as comprehensive failure handling strategies This requires an approach to structure even such complex plans in a way that allows easy introspection, which we present in the next section.

To authors' best knowledge, the projection mechanism presented in this paper is the first mechanism, which goes all the way down to the low-level motion parameters and allows an autonomous robot to use it online to improve its own mobile manipulation plans.

III. Plan Architecture

Let us consider an example plan for real-world applications, e.g., a *fetch* plan. It consists of sequentially executing four other subplans – *go*, *move-neck*, *perceive*, *pick-up* – which can generate failures of 5 different types that *fetch* has to be able to handle. Some of the failures relevant for fetch and place tasks are listed in Table I.

perception-object-not-found	perception system returned no matching object
navigation-pose-unreachable	navigation trajectory is blocked
navigation-pose-in-collision	navigation goal results in a collision with the environment
navigation-goal-not-reached	navigation controller finished but goal was not reached
neck-goal-unreachable	look goal tries to twist robot's neck
manipulation-pose-unreachable	no IK solution exists for pose
manipulation-goal-not-reached	manipulation controller finished but goal was not reached
manipulation-pose-in-collision	manipulation trajectory generates a collision with the environment
gripper-closed-completely	gripper closed completely although an object was expected to be grasped

TABLE I
COMMON FAILURES FROM THE FETCH AND PLACE DOMAIN

The default failure recovery strategy of *fetch* is: if the object could not be fetched due to any of the possible failures, relocate the robot's base to a new location for reaching the object and retry the *fetch* plan again. If *fetch* cannot handle a failure locally, it throws an *object-unfetchable* failure to the higher level of the plan hierarchy.

To enable convenient performance introspection, plans have to be well structured, i.e. be modular, explicit and transparent. In our fetch and place plans that is achieved by separating the control flow from the reasoning necessary to ground abstract action descriptions into the environment. The knowledge required to execute the plan successfully in a given environment is inferred through reasoning rules for grounding entity descriptions. For example, the *fetching* action has seven parameters, and each one has a reasoning rule, which is used to generate different parameter values (see Table II). Thus, the rules define the search space of plan projection, from which the sampling is done.

robot_base_location(ReferenceLocations, Robot, Constraints, BaseLoc)
arm(Object, Robot, Arm)
grasp_pose(ObjectType, GraspPose)
gripper_opening(ObjectType, Distance)
reaching_trajectory(ObjectType, Arm, GraspPose, ObjectPose, Traj)
grasping_force(ObjectType, Force)
lifting_trajectory(ObjectType, Arm, GraspPose, ReachTrajectory, Traj)

TABLE II
REASONING RULES FOR INFERRING PARAMETERS OF A FETCH ACTION

The inference of the missing parameters of an underspecified action description happens during action grounding, namely, within the body of the *action_grounding* rule. The implementation of *action_grounding* is defined by the plan designer, such that each action has a set of reasoning rules associated with it. For example, for the *fetching* action, *action_grounding* is implemented as follows:

```
1   action_grounding(Action, [fetch, NewAction]) :-
2     property(Action, [type, fetching]),
3     property(Action, [object, Object]),
4     property(Object, [type, ObjectType]),
5     property(Object, [location, ObjectLoc]),
6     robot(Robot)
7     RobotLoc = [a, location,
8                 [reachable-for, Robot],
9                 [location, ObjectLoc]],
10    arm(Object, Robot, Arm),
11    grasp_pose(ObjectType, GraspPose),
12    gripper_opening(ObjectType, Dist),
13    ... ,
14    augmented_description(Action,
15      [[robot-location, RobotLoc],
16       [grasp, GraspPose],
17       [arm, Arm],
18       [gripper-dist, Dist],
19       [reach-traj, Trajectory],
20       [grasp-force, Force] ...],
21      NewAction).
```

The code snippet reads as following: for any action description *Action* (line 1), which has a property *[type, fetching]* (line 2) and a property *[object, Object]* (line 3), such that the *Object* has properties *[type, ObjectType]* and *[location, ObjectLoc]* (lines 4-5), the location for the robot to stand to pick up the object is defined as *[a, location, [reachable-for, Robot], [location, ObjectLoc]]* (lines 7-9), the arm with which to perform the fetching is defined through the *arm* rule (line 10), the grasp pose is inferred based on the *grasp_pose* rule (line 11), etc. The inferred parameters are then added to the original underspecified action description to create a new augmented fully-specified description *NewAction* (lines 14-20), which is then passed to the *fetch* plan (line 1).

If the *fetch* plan encounters a failure, it asks for the next grounding of the *fetching* action. This results in the reasoning rules giving the next suitable set of action parameters, with which the plan is retried. In this paper, we go through the different groundings of the same action not in the real world, which would result in, e.g., the robot physically repositioning itself, but in projection, such that only one attempt, which has been predicted to succeed by projecting into the future, is executed in the real world.

IV. Online Plan Projection

The projection library that we took as a base for implementing our online plan projection is the one described in [14]. It contains functionality for setting up a projection environment and a 3D world [15], where one can execute robot plans. It uses Bullet physics engine[2] to represent the 3D state of the world and to do physics simulation, OpenGL's GLUT library[3] to do visibility reasoning and to visualize the world state, KDL-based[4] inverse kinematics solver to do reachability reasoning, and other external and internal tools. Visualizations shown in Figure 1 are screenshots of this 3D environment. In projection, all the motions of the robot are not continuous, as in traditional simulators, but discrete, so the robot goes through key poses of motions by "teleporting". This is the level of abstraction sufficient for our plan projection framework for making realistic predictions about action outcomes: physics-based methods provide fine-grained information sufficient to perform geometric reasoning. Opposite of the precision requirement, projection also should not significantly delay execution, i.e. it should be much faster than realtime. Hence, projection does not ensure the correctness of trajectories generated by the motion planner to connect the inferred via points. In case the motion controllers throw a failure, those are handled by well-designed failure recovery strategies, specified in the plan (see related discussion in Section VII).

To be able to use the projection library, we provided it with two equivalent implementations of all the low-level motions that the robot can execute – one for the real robot and one for projection. We have plugged these low-level motions into our plans, in order to be able to project complete plans. To be able to switch between execution in the real world and in projection, we made sure that the real-world data of the robot and data produced in projection mode, which are both generated concurrently, is published to other software components of the robot on different channels to avoid "sleepwalking": the robot should not move in the real world, when projecting actions in its "imagination".

We use the same geometric world for robot's world state representation and for projection. Due to this tight integration, it is easy to initiate projection with the current world state of the robot at any point in time and manipulate it for projecting into the future, then reset it back to the original state representing the real world once projection is over. This is important for implementing online plan projection, as the robot should be able to start projection runs at any designated point in the code.

To run projection online for finding plan parameterizations that lead to successful task execution, we have implemented the *with-projected-task-tree* construct. It is wrapped around the segment of the robot control program that we would like to project. For example, let us consider a transporting action (see Figure 2).

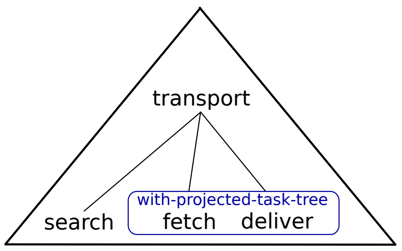

Fig. 2. Task tree of *transporting* action

We would like to project its plan with different parameterizations of the *fetching* and *delivering* actions and choose the one that leads to successful execution. The *searching* action

[2]http://bulletphysics.org/
[3]https://www.opengl.org/resources/libraries/glut/
[4]http://www.orocos.org/wiki/orocos/kdl-wiki

is not being projected, such that the projection is performed after the object has been found: we need to know the exact location of the object in order to calculate correct positioning of the robot base and the manipulation trajectories.

The signature of *with-projected-task-tree* is as follows:

```
(with-projected-task-tree
   parameters-to-infer
   number-of-projection-runs
   cost-function-to-compare-results
 code-to-project-and-execute)
```

The transporting plan is, therefore, defined as follows:

```
(def-plan transport (?object ?search-location)
 (perform (an action
              (type searching)
              (object ?object)
              (location ?search-location)))

  (with-projected-task-tree
     (?fetch-robot-location ?arm ?grasp
      ?deliver-robot-location ?obj-target-location)
     *number-of-projection-runs*
     #'pick-best-parameters-by-distance

    (perform
     (an action
         (type fetching)
         (object ?object)
         (robot-location ?fetch-robot-location)
         (arm ?arm)
         (grasp ?grasp)))

    (perform
     (an action
         (type delivering)
         (object ?object)
         (robot-location ?deliver-robot-location)
         (target ?obj-target-location)))))
```

The code segment with fetching and delivering actions will be executed in projection **number-of-projection-runs** times and the different executions will be compared with the *pick-best-parameters-by-distance* cost function. Finally, the five parameters (*?fetch-robot-location*, *?arm*, *?grasp*, *?deliver-robot-location* and *?obj-target-location*) of the best run will be used when executing the same code segment on the real robot.

In order to be able to find the best run, i.e., in our case, to implement the *pick-best-parameters-by-distance* cost function, we need to have access to all the relevant parameters of both *fetching* and *delivering* actions. Thus, we use performance introspection tools. As opposed to the typical model-based approach to AI planning, where control routines are modeled in a purely symbolic way, our system represents the control routines in a subsymbolic way but, at the same time, such that it would be possible to *symbolically* infer consequences of executing a plan.

V. Performance Introspection

The main data structure in which plan-relevant information is stored during execution is the task tree [16] (see Figure 3). The nodes of the tree correspond to *tasks*. A task is a representation of the runtime state of an annotated segment of the robot control program that is semantically meaningful in the context of plan execution and is important for introspection purposes. The most common task is the representation of an *action description* that is performed within the plan. Every node in the task tree contains a unique path that is used

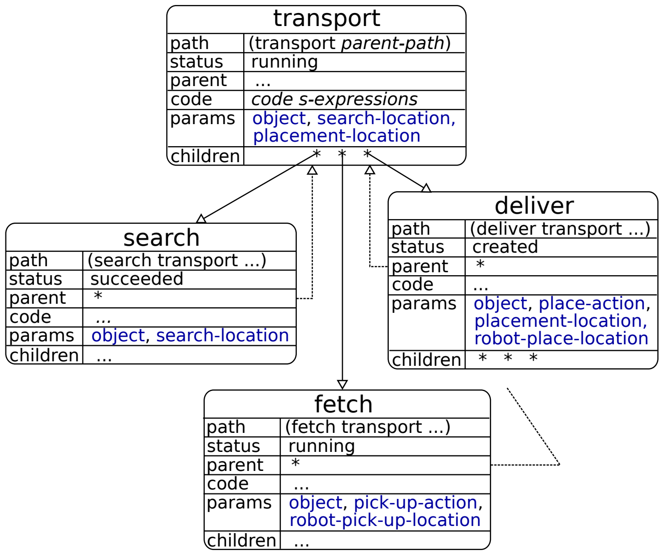
transport
path | (transport parent-path)
status | running
parent | ...
code | code s-expressions
params | object, search-location, placement-location
children | * * *
search
path | (search transport ...)
status | succeeded
parent | *
code | ...
params | object, search-location
children | ...
deliver
path | (deliver transport ...)
status | created
parent | *
code | ...
params | object, place-action, placement-location, robot-place-location
children | * * *
fetch
path | (fetch transport ...)
status | running
parent | *
code | ...
params | object, pick-up-action, robot-pick-up-location
children | ...

Fig. 3. Diagram of the task tree data structure

for indexing and searching, a status (*succeeded* or *failed*), pointers to the parent node and children nodes, the code expressions of the task, the parameters with which it has been called, information about its failures etc. The task tree is automatically generated at runtime while tasks are being executed. To access the task tree and to reason on it, an API consisting of first-order logic predicates is defined. The ones relevant for this paper are listed in Table III.

task(_, Task)	Binds *Task* to any task of the current task tree
task(SubtreePath, Task)	Binds *Task* to any task of subtree defined with *SubtreePath*
task_path(Task, Path)	Gives the unique path of the task node defined with *Task* and binds it to *Path*
task_outcome(Task, Outcome)	Binds the result of *Task* to *Outcome*
task_failure(Task, Failure)	If *Task* failed, binds its failure object to *Failure*
task_created_at(Task, Time)	Binds the timestamp of creation of *Task* to *Time*
task_started_at(Task, Time)	Binds the timestamp of when *Task* started execution to *Time*
task_ended_at(Task, Time)	Binds the timestamp of when *Task* execution ended to *Time*
action_subtask(SubtreePath, _, Task, Action)	Binds all tasks from *SubtreePath* corresponding to action descriptions to *Task* and their action description to *Action*
action_subtask(SubtreePath, ActionType, Task)	Binds all tasks from *SubtreePath* corresponding to action descriptions of type *ActionType* to *Task*
action_task_previous_sibling(SubtreePath, Task, ActionType, PrevTask)	For an action task *Task* in *SubtreePath* finds the previous action task of type *ActionType* and binds it to *PrevTask*
action_task_next_sibling(SubtreePath, Task, ActionType, NextTask)	Binds the next action of type *ActionType* of an action task *Task* in *SubtreePath* to *NextTask*

TABLE III
Predicates for accessing task tree data

These predicates can be used by the robot as building blocks for answering questions such as “What was the last action I was trying to perform?”, “Which parameters did I use?”, “Was the action successful?”, “Where was I standing at that moment?”, “What were the failures?” etc. For example, if a placing action failed, the robot could crawl the task tree for the picking up action that preceded the failed placing action to see if the source of failure could have been that the object was picked up in a wrong way. As the input arguments of plans are stored in the task tree, the robot can access all the action parameterizations that it used during execution and reason about them. To keep introspection queries simple and straightforward it is crucial for the task tree to be well structured. This is achieved automatically if the plans are designed in a structured and systematic way, as is, for example, the case with our fetching and delivering plans.

Let us consider the introspection queries that compare different parameterizations of the *tranport* plan that resulted in successful execution, based on a certain cost function, e.g., a function that compares lengths of trajectories the robot would have to execute:

```
successful_fetch_and_deliver_params(ParentTaskPath,
                                  PickNavAction, PickAction,
                                  PlaceNavAction, PlaceAction) :-
    action_subtask(ParentTaskPath, fetching,
                   FetchTask, FetchAction),
    task_path(FetchTask, FetchTaskPath),
    action_subtask(FetchTaskPath, picking-up,
                   PickTask, PickAction),
    task_outcome(PickTask, succeeded),
    action_task_previous_sibling(FetchTaskPath,
                                 PickTask,
                                 navigating,
                                 PickNavTask),
    action_subtask(FetchTaskPath, navigating,
                   PickNavTask, PickNavAction),
    action_subtask(ParentTaskPath, delivering,
                   DeliverTask),
    task_outcome(DeliverTask, succeeded),
    task_path(DeliverTask, DeliverTaskPath),
    action_subtask(DeliverTaskPath, placing,
                   PlaceTask, PlaceAction),
    task_outcome(PlaceTask, succeeded),
    action_task_previous_sibling(DeliverTaskPath,
                                 PlaceTask,
                                 navigating,
                                 PlaceNavTask),
    action_task(DeliverTaskPath, navigating,
                PlaceNavTask, PlaceNavAction).
```

We extract the fetching and delivering tasks from the task tree and make sure that their outcomes are *succeeded*. If not, the rule fails and we do not get any parameter bindings, which means that the projection run was not successful. Next, as *picking up* is a subaction of *fetching*, we extract the picking up task from the fetching subtree and the action description corresponding to that task. As we are applying introspection after execution has finished, we can access all the parameters of the *picking up* action, including the arm that was used, the grasp pose, even the trajectories. Once we have the picking up action, we find the navigating action that last preceded it. That action contains the location description that was used to position robot's base. Similarly, we can do the same for the *placing* action, which is a subaction of *delivering*. Having the navigation locations that preceded the *picking up* and *placing* actions, we can, for example, approximate the distance driven during the *transporting* action.

Thus, with a small number of queries we can access all the parameterizations of the general plan that were used to ground it into the environment at hand.

VI. Experimental Analysis

We evaluated our approach on a breakfast table setting scenario with a PR2 robot. The scenario included fetching 5 different objects and bringing them to the table. We executed it 10 times without our system and 10 times with it. In an effort to reduce the randomness factor in execution we constrained the initial as well as goal locations of objects to be constant in all the runs. The initial configuration we chose was random, with the constraint that the objects should be at least 2 cm away from each other and not be completely out of reach of the robot. The setup is shown in Figure 4. The robot transports the objects one by one in the following order: *milk*, *cup*, *cereal*, *bowl*, *spoon*.

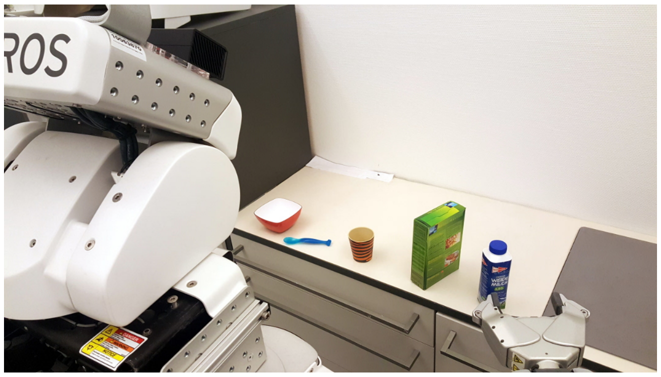
Fig. 4. Experimental setup – initial configuration

The first action in the *transport* plan is the searching action, so the robot searches for the object of a specific type on the surface of the counter. As it does not know where exactly the object is and as its field of view is limited to the sensor's image size, it samples random poses on the surface, navigates to a location from where the pose is visible, and moves its head to point at it. Then it calls the perception system [17]. If perception fails, the robot picks a different pose on the surface and retries.

Once the object has been found, next in the plan are the *fetching* and *delivering* actions wrapped into *with-projected-task-tree* as shown in Figure 2. If projection is disabled, the robot samples a location to stand to reach the object, drives to the location, samples an arm and a grasp pose to use and tries to reach. If a manipulation failure occurs, the robot samples a different location to stand, drives there and retries. This backtracking behavior is time consuming and leaves an impression of incompetent behavior. Additionally, if the object placing orientation is difficult to achieve with a certain grasp and the robot is unlucky to sample that particular grasp, *deliver* will completely fail.

If projection is enabled, the robot executes four runs of projection. We refer to [10] for the proof of an argument that even with a small number of randomly generated execution scenarios it is nonetheless very likely that the probable failures will be eliminated. Projection is used to choose the following four parameters: the arm to grasp with, the grasp pose and the locations of the robot base for picking up and placing the objects. Action parameterizations in successful projection runs are evaluated based on a heuristic approximation of distances that the robot would have to move and, thus, the best run is chosen. Chosen plan parameters are then used in the real world to execute the *fetching* and *delivering* actions.

Our perception system has about 2 cm precision for the objects in the experimental setup, which tends to improve when the robot gets closer to the object. Due to that, the fetching plan reperceives the object directly before grasping. We were faced with two alternatives: either perceive the object once, project, then reuse generated trajectories even if the result of reperception varies highly from the previous result, or not reuse the trajectories but rather only predict the arm and the grasp pose that is most likely to succeed. We went in favor of the second, as reusing trajectories proved to be very prone to misgrasping errors.

As the aim of projection is to find the action parameterization that leads to successful execution, we chose our evaluation criteria to be the success rate of actions and the number of failures that happen. Table IV shows statistics from the first run of the system without using projection.

Object	**milk**	**cup**	**cereal**	**bowl**	**spoon**	**Total**
Runtime	411.6	207.2	142.4	170.9	229.2	1181.4
Arm used	left	right	right	right	right	
Grasp used	front	front	back	top	top	
Success	no	yes	yes	yes	yes	4 of 5
Num. fail.	52	22	14	3	4	95

TABLE IV
EXPERIMENTAL RESULTS OF ONE SCENARIO RUN WITHOUT PROJECTION

In Table IV it can be seen that the delivering action of the milk object failed and created 52 manipulation failures, which resulted in applying failure handling strategies that relocate robot's base. The average number of failures that happened in the 10 scenario runs that did not use projection, as seen in Table V, is 55.45 failures per run.

Object	**milk**	**cup**	**cereal**	**bowl**	**spoon**	**Total**	**Per obj.**
Num. fail.	22.25	9.0	11.8	5.4	7.0	55.45	11.09
Success rate	75%	100%	100%	100%	80%	91%	91%

TABLE V
RESULTS AVERAGED OVER TEN SCENARIO RUNS WITHOUT PROJECTION

Experimental results of one scenario run with the projection system enabled are shown in Table VI. It can be seen that the number of failures is very small. There is one collision failure that happened when transporting the *spoon* object. However, it was expected to be 0 if the parameterization was predicted to generate successful behavior. As mentioned before, we only infer the arm to use and the grasp pose, which should lead to successful execution, and do not reuse the trajectory generated in projection. As the trajectory generation algorithm is randomized, even slight changes in object pose can result in no valid trajectory being found. The collision failure happened when picking up the spoon because the perception system changed the pose estimate of the object significantly enough for the inverse kinematics solver to fail for the new pose with the given arm and grasp pose. This poses an issue for the projection mechanism if the perception results are inconsistent with respect to object orientations, which is the case in our perception system: the axes of the object pose can flip randomly. In that case, e.g., a front grasp that was supposed to be ideal for the current world state becomes unreachable and a back grasp should be chosen instead. This situation happened in one of the 10 runs of the scenario with projection, where the grasp for the milk object failed although a valid parameterization was successfully inferred.

Object	**milk**	**cup**	**cereal**	**bowl**	**spoon**	**Total**
Proj. time	47.9	25.0	23.3	12.4	15.5	
Infer. time	31.9	5.6	4.2	3.3	3.8	
Runtime	193.8	155.6	151.5	132.5	160.7	823.2
Arm used	right	right	right	right	right	
Grasp used	front	front	back	top	top	
Proj. success	2 of 4	4 of 4	4 of 4	4 of 4	4 of 4	
Success	yes	yes	yes	yes	yes	5 of 5
Num. failures	0	0	0	0	1	1

TABLE VI
EXPERIMENTAL RESULTS OF ONE SCENARIO RUN WITH PROJECTION

Table VII shows the average number of failures that happen in real world when using fast plan projection.

Object	**milk**	**cup**	**cereal**	**bowl**	**spoon**	**Total**	**Per obj.**
Num. fail.	6.9	2.22	0.5	0.0	0.2	9.82	1.96
Success rate	80%	100%	100%	100%	100%	96%	96%

TABLE VII
RESULTS AVERAGED OVER TEN SCENARIO RUNS WITH PROJECTION

Based on experimental data, we conclude that our system improves the success rate of fetch and deliver plans from 90% to 96%, which is not substantial since the robustness of the evaluation scenario is already considerably high. However, we additionally decrease the amount of manipulation-related failures and, therefore, times when robot physically backtracks, from 55.45 per run to 9.82, which is more than a 500% improvement (see Figure 5).

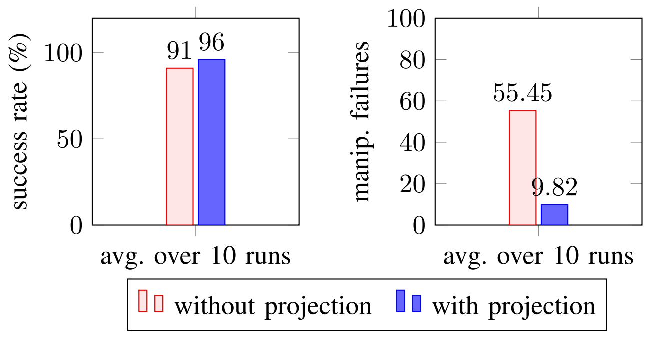

Fig. 5. Success rate and number of manipulation failures comparison between executions without and with projection.

In the 10 times we ran the scenario with projection-based reasoning disabled, execution time varied between 696.4s and 1181.4s. There is a vast number of factors that affect the runtime of execution on a robotic system, including the efficiency of computational processes, power of the underlying hardware, time optimality of robot's controllers etc., as well as the amount of physical backtracking that happens due to, e.g., perception failures or suboptimal action parameterization samples. Due to such high variety, we do not consider 10 execution runs to be enough to provide statistically meaningful empirical estimates of execution time. Considering that one scenario run takes about 15min, a large-scale evaluation might not be feasible. However, below we give a short report on observed runtime to give a general intuition of our implementation efficiency for practical reasons. In the 10 runs of the scenario with projection enabled, 196 action projections have been executed. The average projection time per one run of transporting one object was 6.2s, and 4.1s if excluding all the non-successful runs. This can be considered sufficiently fast with respect to the pace of action execution. The hardware used for projection was a laptop with 8GB RAM, an 8 core i7 CPU and an NVIDIA GeForce GT 650M graphics card. The average runtime of a full scenario run was 877.28s without projection and 823.85s with projection.

VII. Conclusion, Discussion and Future Work

In this paper we presented the fast plan projection mechanism, which can be used to specialize a general plan towards the environment and task at hand by choosing action parameterizations that are predicted to lead to successful task execution. We showed how carefully designed plan structure can benefit plan introspection and how to apply introspection tools to choose parameterizations of executed actions that were predicted to succeed in the projection environment. We demonstrated how the results are easily integrated into robot's executive module such that the optimized part of the plan can be executed right away in the real world. Finally, we evaluated our approach by showing how a PR2 robot is able to use the system to choose action parameterizations that increase task execution success rates and decrease failure rates of fetch and deliver actions in a real world setting.

In the evaluation section, we mentioned one limitation of our approach, which is a general limitation that any system that thinks ahead of time based on the current world state has: if the world state representation is inaccurate, projection has a higher chance of producing action parameterizations, which do not lead to successful task execution when transferred onto the real world. We address this problem by integrating our projection results only as a suggestion for the planner, and if suggested parameterization fails, execution continues with its default failure handling routines, trying to find a better parameterization without the help of projection-based reasoning. A similar limitation is the danger of the world state changing while projection performs its inference. In our application scenarios, which happen in semi-controlled environments, external influences and, therefore, unexpected world state changes happen with a sufficiently low frequency compared to the runtime of the projection-based inference.

One important assumption that has to be made about the projection mechanisms is that the probability distribution of failing in projection is similar to the real world. In our fetch and deliver scenarios, we use a high-precision model of the environment, which makes geometric reasoning and basic world dynamics sufficiently realistic. However, some failures such as an object slipping away from the gripper or a shiny object causing misdetection, are not represented in our system, and that is a limitation. In future, we are planning to learn failure models to use in our projection environment based on large-scale data collected from real world experiments.

Acknowledgments

This work was supported by DFG Collaborative Research Center *Everyday Activity Science and Engineering (EASE)* (CRC #1320).

References

[1] S. Miller, J. Van Den Berg, M. Fritz, T. Darrell, K. Goldberg, and P. Abbeel, "A geometric approach to robotic laundry folding," *The International Journal of Robotics Research*, vol. 31, no. 2, 2012.

[2] K. Okada, T. Ogura, A. Haneda, J. Fujimoto, F. Gravot, and M. Inaba, "Humanoid motion generation system on hrp2-jsk for daily life environment," in *IEEE International Conference Mechatronics and Automation*, 2005.

[3] G. Kazhoyan and M. Beetz, "Programming robotic agents with action descriptions," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[4] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[5] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of artificial intelligence research*, vol. 20, 2003.

[6] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning." in *IJCAI*, 2015, pp. 1930–1936.

[7] L. P. Kaelbling and T. Lozano-Pérez, "Integrated task and motion planning in belief space," *The International Journal of Robotics Research*, vol. 32, no. 9-10, 2013.

[8] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 639–646.

[9] S. Hanks, "Practical temporal projection," in *AAAI*, vol. 90, 1990.

[10] M. Beetz and D. McDermott, "Fast probabilistic plan debugging," *Recent Advances in AI Planning*, 1997.

[11] S. Rockel, Š. Konečnỳ, S. Stock, J. Hertzberg, F. Pecora, and J. Zhang, "Integrating physics-based prediction with semantic plan execution monitoring," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[12] L. Kunze, M. E. Dolha, E. Guzman, and M. Beetz, "Simulation-based temporal projection of everyday robot object manipulation," in *The 10th International Conference on Autonomous Agents and Multiagent Systems*, 2011.

[13] P. Abelha and F. Guerin, "Learning how a tool affords by simulating 3d models from the web," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[14] L. Mösenlechner and M. Beetz, "Fast temporal projection using accurate physics-based geometric reasoning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[15] L. Mösenlechner and M. Beetz, "Parameterizing Actions to have the Appropriate Effects," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011.

[16] L. Mösenlechner, N. Demmel, and M. Beetz, "Becoming action-aware through reasoning about logged plan execution traces," in *International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[17] M. Beetz, F. Balint-Benczedi, N. Blodow, D. Nyga, T. Wiedemeyer, and Z.-C. Marton, "RoboSherlock: Unstructured Information Processing for Robot Perception," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.