

Planning and Execution of Robot Tasks Based on a Platform-Independent Model of Robot Capabilities

Jennifer Buehler and Maurice Pagnucco¹

Abstract. The diversity of robotic architectures is a major factor in developing platform-independent algorithms. There is a need for a widely usable model of robot capabilities which can help to describe and reason about the diversity of robotic systems. We propose such a model and present an integrated framework for task planning and task execution using this model. Existing planning techniques need to be extended to support this model, as it requires 1) generating new objects during planning time; and, 2) establishing concurrency based on data flow within the robotic system. We present results on planning and execution of an object transportation task in simulation.

1 Introduction

In most real-world environments, accurate predictions of the outcomes of a robot's actions are virtually impossible. For such robotic systems to operate in real-time, **approximate** models and algorithms are required which help to *estimate* the outcome. The diversity of robotic systems further complicates making such predictions, as every robot contributes different capabilities. There is a need to conceptualise robot capabilities and formalise a platform-independent model which can be used to describe and reason about any robot's capabilities, and which can provide a means to *estimate* a robot's performance without the need to know platform-specific details. Such a model can not only efficiently communicate robot capabilities across diverse platforms but can also prove useful to algorithms requiring an efficient means for performance prediction. For example, *task planning* usually involves a search through a state space, in which prediction of the robot's execution plays a central role for measuring plan quality. Such a planner would greatly benefit from better execution prediction of actions, even if it is only approximate.

In this paper, we present our platform-independent model of robot capabilities and show how this model can be used in a *planner* to find a combination of capabilities to solve a task. This model also serves as an interface to *execute* such a plan, which we show using our *ROS* (*Robot Operating System*) implementation of this interface. A *planner* commonly requires a definition of the *planning domain*. Such domains are mostly designed to suit particular tasks or experiments. Instead, with our capability model, we provide a more flexible, platform-independent planning domain which can be used for a variety of *different* robot tasks—for example, operating in a search and rescue scenario, or in a kitchen environment. Any robot providing a description of its capabilities through this model, along with a platform-specific implementation of the capability interfaces, can be seamlessly integrated into our planning system.

This paper contributes a platform-independent model of robot capabilities which can provide execution estimates at three levels of approximation accuracy. We propose an integrated approach for task planning and execution using this model. In order to support this capability model in a planner, existing planning techniques need to be extended: (1) to support generation of new objects at planning time, we need to instantiate actions during planning; and, (2) in order to define concurrency based on data flow within the robotic system, we need to extend PDDL to allow predicates within over all states and account for this in the implementation of the planner. Our results show that the resulting plan is executable on a robot using our ROS interfaces for the capability model.

We will first introduce our capability model and then detail the use of this model in our temporal planner. Related work will be referred to when it becomes relevant. We finish with a description of our ROS interfaces and present a series of experiments evaluating our planner and the execution on a robotic system in simulation.

2 Robot Capabilities

Diversity does not only emerge from differences in hardware but also from distinct robot software architectures and algorithms. Most previous work on robot capabilities integrates robot *functionalities* or *resources* (e.g., sensors, actuators) into a framework, mostly with the purpose of computing a *utility* value to express the robot's suitability for a task, e.g., [2], or to form robot coalitions [13]. A capability can also be a simple *subtask*, for which each robot learns their suitability [7]. While such concepts have been identified as factors for robot capabilities, no work to date has been devoted to formalising a general model for robot capabilities which unifies all these relevant aspects and can be used in various frameworks. We propose a platform-independent model of robot capabilities which integrates hardware and software aspects to provide execution estimates and which also serves as an interface for task execution.

Most research relating to robot capabilities revolves around determining a robot's *intrinsic* capabilities, which captures what a robot can do in general, e.g., lift an object. *Extrinsic* factors take task specific details into account, i.e., lifting a particular rock of certain size and weight. Extrinsic factors other than the common simple metric distance to the task have not received significant attention; it is difficult to estimate factors like task-completion time for various tasks [7]. We argue that, even though estimates may only be approximate, more expressive execution estimates would significantly reduce the complexity and increase the robustness of algorithms in most realistic applications where robots contribute a variety of capabilities. We propose a systematic approach to incorporate extrinsic task details for all possible tasks, by allowing robot capabilities to be *pa-*

¹ School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW, 2052, Australia. Email: {jenniferb, morri}@cse.unsw.edu.au

parameterised according to a specific task. We define a *capability* as a simple functional element which can be part of many different tasks, inspired by Zuech and Miller [16, p. 163]:

“There are a limited number of task types and task decompositions [...] with only a few different types of **reach, grasp, lift, transport, position, insert, twist, push, pull, release**, etc. A list of *parameters* with each macro can specify *where* to reach, *when* to grasp, *how far* to twist, *how hard* to push...”

To such *physical* capabilities we further add *computational* capabilities such as localisation, path planning and object recognition, and *sensing* capabilities such as vision. With this definition, a capability *abstracts* from hard- and software specifics at a medium level of granularity. Platform-specific details remain at the low level of capability implementations for the individual robot. For example, on an abstract level it is not important *how* a robot grasps an object (e.g., which finger movements) but only *what* it can *probably* grasp. More complex tasks are made up of several such simple capabilities.

Formally, a capability $C = \{P, E, I, O, hw, pr\}$ has preconditions P , effects E , input data I and output data O , hardware dependencies hw and parameters pr . P and E are required for planning. For example, an effect of REACH is that the manipulator is at- $Pose(?manip, ?pose)$, which is also the precondition for GRASP. Goals for planning are also specified with conditions. For example, *picking up an object* is specified by the goal that the object is not ($grounded(object)$) —an effect of LIFT.

Input and output data I, O are used to describe the data flow between capabilities, which is a natural process in robotic algorithms. For example, a 3D-Point-Cloud is produced as output O from VISION and required as input I for LOCALISATION. Such data input and output requirements constrain how capabilities can be connected in order to solve a task, and therefore can be seen as a special kind of preconditions and effects within the planner. A similar constraint is used in [13] and [14] to connect *schemas* or *activities* with matching information types. Each capability also has a list of hardware dependencies hw which efficiently captures the resources required for a task. This can be useful in evaluating plan quality.

One central concept in our capability model is the “parameters” pr which formalise the capability’s extent relating to *extrinsic factors*. Such parameters are used to *approximate* a robot’s capability properties. For example, areas that a robot can REACH are approximated with spherical shapes around the manipulators (see Figure 1); terrain on which a robot can MOVE may be described by indices of “terrain roughness” with assigned average speeds; object sizes it can GRASP may be approximated with a bounding volume; LIFT will be assigned weight ranges, and so on. While such parameters by no

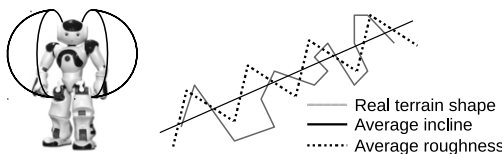


Figure 1. Left: Parameter space for Reaching. Right: Possible way to express approximate terrain shape.

means allow for accurate predictions, they still provide a much better estimate than considering only intrinsic capabilities (which simply assume the robot has or does not have the capability). Because of the simplicity of this approximation, a robot can *learn* its parameters for a capability. Such approximate information is not only useful to share

knowledge about a robot’s capabilities, and to estimate a robot’s eligibility for a specific task, it also decreases the search space in the task planner (because inapplicable robot actions are detected efficiently). Another advantage of this representation is that it accounts for cases in which only approximate information is available (e.g., only approximate terrain or object shape is known at planning time). Approximation parameters have to be chosen individually for each capability. However, as argued above, there are only a limited number of such capabilities which can be combined to form more complex ones. This makes an individual treatment feasible. Still, we need to choose simple representations for parameters, since large parameter spaces will impede the application of learning algorithms.

Levels of estimation accuracy— The model of Robot Capabilities can provide estimates of task solution qualities at three levels of accuracy. At the **Basic Level**, mere presence (or absence) of capabilities required for a task is taken into account and extrinsic details are ignored. At the middle level, which we name the **Approximation Level**, a *rough estimate* is given on *how well* a robot meets the task-specific requirements. At the third level, more accurate estimates are given based on the result of more elaborate planning algorithms which are highly platform-dependent, therefore we will refer to this as the **On-Board Level**. Task planning or evaluation of the probability of task success can be done at each of these levels. One major strength of this representation is that a compact model describing a robot’s capabilities at the Basic and Approximation levels can be communicated across different robot platforms, and each robot potentially can determine team-mates’ approximate eligibility for a task locally without the need to know platform specifics. The On-Board Level involving more elaborate planning will have to be evaluated locally by the robot in question, therefore details cannot be shared directly across the team and execution estimates have to be requested from the robot in question.

At the Basic Level, robots which do not have the required capabilities for a task can be ruled out. At the Approximation Level, task-specific (extrinsic) details are considered in an *approximate* way in order to compare different robot’s likely performance. The robot’s capability *parameters* are used to obtain this approximation. Overall, verifying that a capability meets the requirements of the *task details* (evaluating the *extrinsic* capability) involves matching a capability **specification** (e.g., the range a robot can reach) to a task-specific **instantiation** (i.e., the point to reach). Matching of specification and instantiation on the Approximation level is quite straightforward and efficient—e.g., checking if a value lies within a range, or a point lies within a simple shape. The On-Board level requires more elaborate calculations, mostly invoking other costly and platform-specific algorithms, e.g., using motion planners to determine whether a point is reachable by the robot. This greatly improves the accuracy of the execution prediction but it also comes at a cost of computing the result, which can be prohibitive if it has to be done for many node expansions in a planner.

3 Planning with Capabilities

We encode our model of robot capabilities in the *Planning Domain Definition Language (PDDL)* to be used in a domain-independent planner. The “domain” is the capability model, with each capability defined as an action. *Automated planning* is a well-researched field. A popular and successful approach has been to use forward-search in the state space, e.g., *SAPA* [4], *Temporal Fast Downward (TFD)* [5], *COLIN* [3], and others. For a more extensive review of previous

work, we refer to our earlier paper [1]. Few approaches in robotics use a domain-independent planning system. Temporal Fast Downward (TFD) is extended in [8], building on their previous continual planner. Also [15] uses the TFD/M planner, designing a PDDL domain for the particular task of exploration and transportation. Instead, with our capability model we are able to design a PDDL domain which is general for *all* robotic tasks, including *extrinsic details* and describing possible actions a robot can take in *any* environment.

The aim is to minimise the time it takes the robot(s) to complete the tasks, therefore a *temporal planner* is required. PDDL2.1 [6] introduces *durative actions* to account for time, including durative conditions and effects which apply throughout the execution of the action and are specified within the *over all* definition. Continuous effects in PDDL model change over time by affecting *numeric variables*. Such effects are mostly linear (e.g., [3]), although non-linear effects have been proposed [11].

We develop a planner to find a combination of capabilities required to complete a task. As we will discuss in the following, we need to extend current planning techniques to support this capability model.

Concurrency requirements due to data flow—In robotics, concurrency is strongly tied to data being produced in one software module and consumed by another. For example, robot navigation requires the robot's current position. It does not matter where the data comes from—it could come from the robot's localisation, or even from another robot. This observation is also made in ASyMTRe [13] which connects action schemas by matching their data input/output in their coalition formation algorithm, however not explicitly encoding *required* concurrency. While other approaches encode concurrency by specifying particular actions required to run concurrently (e.g., [9]), in robotics it is more natural to base this on *required data flow*. It does not matter which particular action generated the data. To encode this in PDDL, we need continuous effects which allow us to express **certain data continuously being produced** by an action. For example, LOCALISATION will continuously output the robot's current position. This can be expressed with a predicate of the form `gen-data(?robot ?data)`. For this to be possible, we need to extend PDDL to allow predicates within *over all*. We shall call such continuous predicate effects and conditions *persistent predicate effects and conditions*. They can encode concurrency of actions, simply by requiring a persistent predicate effect of one action A to unify with the persistent predicate precondition of another action B, thereby establishing that A and B need to run concurrently. We can also allow **cycles in the data flow**, which can be required in a robotic domain, for example to express that localisation, path planning and navigation have to run concurrently and exchange data cyclically: navigation outputs odometry information which is used by localisation to improve the accuracy of the current robot position; the path planner has to continuously update the path based on this current robot position; and, this path is needed again by navigation, closing the cycle (note that this is a simplified example, as other capabilities like vision are also required for this task). While cycles might not be required (i.e., localisation works without odometry data), the final plan needs to represent the correct data flow among capabilities, to achieve correct plan execution (see Section 4). To support cyclic dataflow, we apply all persistent predicate effects of all running actions at the same time to the current planning state and subsequently check whether all continuous predicate conditions are fulfilled—if not, the planning state is invalid and the planner needs to backtrack.

Data generation during planning—Another important property of robotic domains which needs to be considered in the planner is the

handling of infinite domains. Commonly, planning is based on a predefined number of objects in the planning problem. These objects are used to instantiate all actions in all possible ways, thereby obtaining a number of *ground actions* which are used during search. The main reason for the use of such fully-grounded models is that is easier to derive heuristics. However, the downside to this common approach is that it does not allow for new objects during planning time. For application to robotics this is an important property, as complex data is generated during the robot's operation. For example, consider action GRASP. Before the grasp is possible, the robot needs to REACH to the object. The standing position *S* to where the robot has to navigate before it can REACH to the object depends on the particular robot and is the result of a planning action REACH-PLAN. Also, the position *R* where to REACH is not necessarily equal to the object's position *O*—it depends on the manipulator properties for grasping and is the result of GRASP-PLAN. This example is illustrated in Figure 2. The

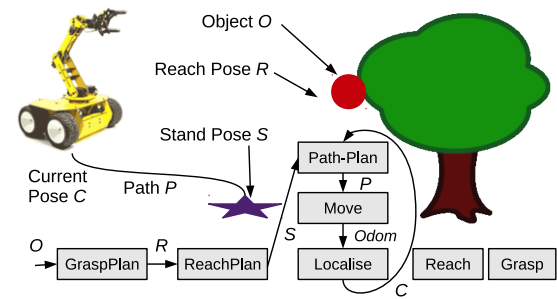


Figure 2. Plan to grasp an object. Arrows represent data flow.

robot has to navigate from *C* to *S* before being able to REACH and then GRASP. Coordinates *C*, *S*, *R*, the odometry information *Odom*, and the path *P* are computed during planning time by the actions. Essentially, such coordinates are equivalent to new *objects* introduced during search. Taking the common approach to pre-define all objects within the planning problem in advance would mean enumerating all possible reaching positions and navigation targets and all paths between all positions. This would clearly make the search intractable, due to the infinite characteristics of the domain. Hence, we need to consider only the data (the “objects”) which become relevant as the plan evolves. To allow generation of new objects, we have to support **on-line action instantiation**, which means we can’t pre-generate a set of ground actions before planning but need to instantiate the actions *during* planning. The resulting plan will be passed to the execution module and should contain these new objects within the action parameters, e.g., in `Plan=[act1(<pose>), act2(<pose>)]`, `<pose>` could be generated by `act1` and used by `act2`.

The only recent planner we are aware of which binds variables during planning time is *Optop* [10], a regression-based planner. SHOP2 [12] also binds variables during planning time but it does so only in an internal step which evaluates the precondition and does not bind action parameter’s variables. The extra effort to bind variables during planning makes on-line action instantiation algorithms slower than others and application of popular techniques to derive heuristics is complicated. However, the added support of infinite domains such as numbers (e.g., an action `act(?a -number)`) and generation of new objects during planning is well worth the effort to investigate and improve on-line action instantiation methods further.

To realise on-line action instantiation, we instantiate only those actions which are eligible in the current planning state (at each node

expansion). All variables which appear in the `:parameters` of the action must also appear in the `:(pre)condition`. To determine whether an action is eligible, we need to (1) instantiate its precondition *Pre* using a *variable substitution* *s*, and (2) find a proof of *Pre* | *s* (*Pre* “instantiated with” *s*) with the current planning state, which is a ground proposition set *Facts*. We do this proof using *unification*. Unification in the common definition means that two expressions’ variables can be substituted in such a way that the expressions are equivalent. We introduce **subset unification**, meaning that the expressions don’t have to be exactly the same but one expression is a subset of the other. For example, in the two expressions

```
E1=and ((atom1(?v1, o2))
```

```
E2=and ((atom1(o1, o2) (atom2(o3,o2))
```

E1 is a subset of *E2* for either of the substitutions $s1=[?v1|o1]$, $s2=[?v1|o3]$. Since *E1* and *E2* are conjunctions, then $E2 \models E1$ with (*s1* or *s2*), meaning we can *prove* *E1* from *E2*. *Facts* is always a conjunction and, if we convert *Pre* to Disjunctive Normal Form (DNF), we can find *alternative* proofs for each conjunct *P* \in *Pre*_{DNF}.

The result of such a proof is a list of variable substitutions $S = [s_0..s_n]$, such that $\forall s_i \in S : Facts \models Pre \mid s_i$. With *S* we obtain all proofs of *Pre*, each $s_i \in S$ instantiating the action such that it is eligible in the planning state.

No currently available planner supports both of these requirements: (1) concurrent actions based on data flow requirements including cycles; and, (2) on-line action instantiation. Therefore, we implemented a planner similar to *SAPA*, adding these extensions. This paper focuses on the robot capability aspects of the planner, we refer to [1] for a detailed discussion of the planning algorithms.

External modules—On the Basic Level, we can perform the search using only the planning state *Facts* (a list of propositions). However, on the Approximation level, more complex operations have to be performed, e.g., action *REACH* requires a point-in-shape test to check whether the robot can potentially reach a point. PDDL is not expressive enough to define such operations. TFD/M [5] addresses this lack of PDDL expressiveness by introducing *external modules* in which function calls to external libraries can be specified within PDDL. We also use external modules but, instead of defining library calls within PDDL, we leave the PDDL definition unchanged and integrate calls to external modules *per action* in the planner itself: (1) to obtain the action duration; (2) to check for the precondition; and, (3) to apply the effects on internal data structures of the external module. Depending on which level we plan on, calculations of varying complexity are performed within the external module at the occasions (1)–(3). For example, when planning on the Approximation level, in step (2) the external module looks up the robot’s capability parameter in a database and matches it to the task to check the precondition. Each capability has its own implementation of the external module’s interface, which is dynamically assigned to the PDDL action before planning commences. We also keep an internal data structure for *all* data which is generated within the modules. For example, position data $p=(x, y, z)$ is indexed by an *integer id* and kept in a look-up table for all 3D coordinates. While it would be possible to keep such data structures within the planning state *Facts*, e.g., by a predicate $pose(id, x, y, z)$, this would complicate the PDDL definition and slow down the on-line action instantiation algorithm, because more variables need to be bound. The external module data is passed between planning states, so external modules can access data created in previous steps—e.g., *REACHPLAN* can access reach pose *R* (calculated in *GRASPPLAN*) by its index in the table.

PDDL domain for capabilities—We can now define a PDDL domain in which *actions produce data* and *required concurrency* is specified with persistent predicate conditions and effects. We illustrate this with two example capabilities: *LOCALISE* and *PATHPLAN*.

In addition to the traditional PDDL “objects” we now have *numbers* in action parameters. We define a data type *ID* which is derived from the PDDL type *number* and is used for indices to external module data. During on-line action instantiation, action parameters are bound to actual objects or numbers in the current planning state. This is done during the proof of the precondition, as discussed earlier. For example, consider the *LOCALISE* action:

```
:durative-action Localise
:parameters(?robot ?o -ID ?p -ID)
:condition (and
  (at start (and (odometry ?robot ?o)
                 (currPose ?robot ?p)))
  (over all (dataGen ?robot odomType ?o)))
:effect ( (over all (and
  (atPose ?robot ?p)
  (dataGen ?robot poseType ?p) ) ) )
```

The precondition predicate `(odometry ?robot ?o)` will be instantiated with a robot object and this robot’s odometry data index, while `(currPose ?robot ?p)` binds *p* to that same robot’s world position index, yielding an action substitution like, for example, $s=[?robot|robot1, ?o|1, ?p|5]$, numbers 1 and 5 being indices into the external module look-up tables. Localisation requires continuously updated odometry data (which comes from the *MOVE* action). This is represented in the *over all* condition `(dataGen ?robot odomType ?o)`, which is a *persistent predicate condition* indicating data generation of the odometry data type. `odomType` is a constant defined within the PDDL domain which helps to keep the predicate `dataGen` more general, so it can be used for various data types. Localisation also provides continuously updated data for the robot’s current position. This is represented in the *over all* effect `(dataGen ?robot poseType ?p)`, a *persistent predicate effect*. This position data is concurrently required by action *PATHPLAN* which plans a path from the robot’s location *?p* to another target location *?trg*:

```
:durative-action PathPlan
:parameters(?rob ?p -ID ?trg -ID ?path -ID)
:condition (and
  (at start (and (currPose ?rob ?p)
                 (naviPose ?trg)
                 (= (idGen pathType ?rob) ?path)))
  (over all (dataGen ?rob poseType ?p)))
:effect (and (
  (at start(increase (idGen pathType ?rob) 1))
  (over all(dataGen ?robot pathType ?path)))
```

Again, all variables are bound at the *at start* condition to objects or values within the planning state. Predicate `naviPose` would have been previously set by an action identifying targets that the robot can/should navigate to. A *new path* is created every time *PATHPLAN* is started, and it is continuously updated during the execution. The function `idGen` is used as an index counter for newly generated data (similar to `dataGen`, it uses a constant to bind the data type). The counter is increased every time a new data entry is created, which happens in the action’s start effect. The external module uses this index to store the resulting path in the external module data. Finally, the persistent predicate precondition `(dataGen ?rob poseType ?p)` is only fulfilled if a concurrent action produces updated data, which is the case if *LOCALISE* runs concurrently. Both actions need to be extended to fully represent the semantics but this simplified example suffices to illustrate the idea.

4 Execution of capabilities

We define interfaces for all robot capabilities such that they can be encapsulated in a ROS node and interact with other capabilities via ROS messages. Data flow between capabilities is connected via ROS topics. While we can't provide the full specification of our interfaces within the space of this paper, we shall give an overview.

Each *Capability* provides a set of **ROS services**, which return some information about the capability, or a success flag. This includes requests about the capability's *parameter* and a request to *Start* or *End* execution of this capability. Two important services are *GetApproxEstimate* and *GetEstimate* to request capability execution estimates on the Approximation or On-Board Level for some given task details. Each capability also provides a **ROS Action** *Execute* which executes the capability.

Each *Robot* supporting the capability model provides a service *GetCapabilities* which lists all available capabilities and the topic names under which the services/actions are available. Further, an action *ExecutePlan* takes a sequence of commands (*Start* / *End* / *Execute* for a capability). This plan can be the result of the planner.

Overall, the aim of this capability model along with the defined ROS interfaces is to provide a standardised, simple yet powerful high-level interface to various robot architectures. We can translate the final plan computed by the planner into a sequence of commands which the ROS action *ExecutePlan* can execute.

5 Experiments

In a first set of experiments, we evaluate some performance characteristics of our planner. While we can't compare to other planners (because none supports our capability model), we can measure the number of node expansions it takes for the search to find a valid plan. We then test plan execution on our implementation of Robot Capabilities using the ROS interfaces and the *Gazebo* simulator.

For the experiments, we use 11 actions: *LOCALISE*, *PATHPLAN*, *MOVE*, *GRASPPLAN*, *GRASP*, *RELEASE*, *REACHPLAN*, *REACH*, *PLACEPLAN*, *PLACE* and *LIFT*. Of these, *LOCALISE*, *PATHPLAN* and *MOVE* have cyclic concurrency requirements in their interacting data flow, as illustrated in Figure 2. *PLACEPLAN* determines an arm pose to place an object, similar to *GRASPPLAN*.

The first set of experiments is set in a simple 10x10 grid world, in which robots have to transport objects from their random initial position to a random target. Robots can move vertically and horizontally in this grid world. The aim of these experiments is to examine the effect on the planner performance when varying the number of robots and objects. We plan on the Approximation Level, but robots are homogeneous as we want to measure unbiased planning performance and randomly generated capability parameters would randomise the results. However, planning on the Approximation Level allows us to estimate task time based on distance to the task. All robots move at 1 tile per second. Execution of a capability takes 1 second, except for *MOVE* (which depends on the path calculated in the *PATHPLAN* action) and the planning actions, which take no time. We use informed search. At this stage of our research we are still using a very simple domain-specific heuristic: the maximum distance of any object to its target location. This is admissible because it assumes all objects are transported in parallel. While improved heuristics is the subject of our future work, this simple heuristic should still reflect performance trends of the planner. We compare the results of an A* and Best First search: node expansions *Exp*, the task time *Task-t* it takes for all robots to transport all objects, and the planning time *Plan-t* in

seconds. We show a superset of our experiments in [1]. 100 problem instances are generated and results averaged. Table 1 displays the

Table 1. Results for the object transportation task.

#Robots – #Objects	A*			Best First		
	#Exp	Task-t	Plan-t	#Exp	Task-t	Plan-t
1 – 1	78	18.1	0.18	78	18.1	0.17
1 – 2	239	35.2	0.84	167	37.0	0.55
1 – 3	690	51.1	5.21	303	52.7	1.86
1 – 4	1738	66.9	25.46	494	74.4	5.79
1 – 5	4688	79.6	116.21	920	89.8	19.86
5 – 1	132	15.0	2.54	85	20.9	1.30
6 – 1	137	14.5	4.13	86	19.8	1.98
7 – 1	165	14.8	7.58	88	20.8	2.92
8 – 1	205	14.3	14.17	88	19.8	4.01
9 – 1	210	13.8	21.39	91	19.3	6.03
10 – 1	259	13.4	37.56	91	19.5	8.24

results, showing that even for only one robot, the number of node expansions increases significantly with more objects being introduced. The reason for this is that, in order to find an optimal plan, the robot has to try transporting objects in all possible orders (and for each order, it has to try all actions at each node expansion). The planner is not only solving the planning problem but is implicitly dealing with the Travelling Salesman Problem, finding the optimal path to transport all objects. In the A* case, where performance drops are most evident, the optimal solution to the TSP has to be found by the planner. In contrast, increasing the number of robots for transportation of only one object does not dramatically impact performance. This is because the robots compete for the same object, and the closest robot to the object is found in a relatively early planning step. From this we can conclude that our planner works well for various numbers of robots but only few location-dependent tasks. This is enough for our purposes, as our aim is a planner which returns action sequences for one or few robots possibly cooperating on one task.

Comparing the results of A* and Best First search, as expected we can see how the heuristic causes the planner to be too greedy and find suboptimal solutions but instead saving a significant amount of node expansions. Given our simplistic heuristic, the results can still be significantly improved, which we hope to achieve in future work.

In the next experiment, we introduced some heterogeneity in the robots and want to test planner characteristics when robot *cooperation* is required. Robots are now only able to navigate certain terrain. The grid is split into two halves with different terrain shapes. The generated objects have to be transported to the other side. Now, robots have to *cooperate* in the task. Robots can place objects at the boundary of the terrain, where a robot from the other side can pick it up. However, not all tiles are suitable for placing objects. We vary the number of such randomly generated “placing tiles” at the terrain boundary. Such placing positions could be automatically detected in real environments, e.g., flat surfaces. Again, we want to study the effects on the planning performance. Results are shown in Table 2. As expected, the search space increases with more place positions, because robots have more choices. At the same time, overall task runtime decreases, as robots find improved place positions.

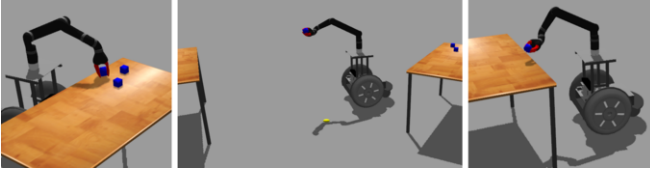
We expect planning time to decrease significantly with a better heuristic, which is a well-known factor in making the planning problem viable. While our code can still be optimised further, we believe the heuristic is the main factor in making the planner a bit slow for large instances of robots and objects. The development of a heuristic for on-line action instantiation is a main part of our future work.

Finally, we want to verify that the interfaces of our Robot Capabilities are suitable to execute a plan on a robotic system. We im-

Table 2. Object transportation task with the need for cooperation.

#Robots – #Places	# Exp	A*		#Exp	Best First	
		Task-t	Plan-t		Task-t	Plan-t
2 – 1	171	29.2	0.78	169	33.2	0.83
5 – 1	298	26.8	9.25	158	30.5	3.87
10 – 1	996	23.9	265.38	206	31.6	40.63
2 – 3	317	28.7	2.44	231	35.2	1.56
5 – 3	1106	26.3	56.94	312	31.5	13.69
10 – 3	1517	21.0	1774.72	385	26.1	160.24
2 – 5	604	26.0	6.20	342	31.9	3.28
5 – 5	1578	24.5	81.95	391	29.1	26.67
10 – 5	2652	19.3	1143.98	553	30.1	168.34

plement each capability within a ROS node and exchange all information via ROS messages. In this experiment, we also use capability VISION3D, which outputs a *PointCloud* to be used as input for LOCALISATION. We also include OBJECTDETECTION (taking a *PointCloud* and yielding a filtered *PointCloud*) and OBJECTRECOGNITION (taking the filtered *PointCloud* and outputting an *ObjectType*). We use our own algorithms for navigation, localisation, object detection/recognition (at this stage using gazebo data to help) and use the ROS *MoveIt!* packages (<http://moveit.ros.org/>) for path and motion planning, and *Graspl!* (<http://wiki.ros.org/graspl>) for grasp planning. We modelled our robot, a Segway RMP100 which has a JACO Arm mounted on top (see Figure 3). We determined the range

**Figure 3.** Segway/JACO robot doing the object transportation task.

parameter for REACH (a *sphere*) by simply trying random end effector positions; similarly the GRASP parameter (*object size*) was found by trying different object sizes. We determined the *speed* of MOVING by simply driving around, recording distance travelled and time. Only one terrain shape is supported for this experiment. We limit capability parameters to these and leave other capabilities unparameterised. The aim of this experiment is for the robot to move a cube from one table to the other. Table heights and object sizes can be varied. The task fails if the cube is lost (we don't implement recovery plans yet). We first generate a plan on the Basic Level and then check if it is ruled out by the parameters on the Approximation Level. This is mostly the case when an object is not within the parameter space of REACH or GRASP. If the plan is ruled out, it should not be executable. This is important, as a planner which misses viable plans is not desirable. We executed 30 random plans deemed not viable by the Approximation Level. In none of the scenarios did the robot succeed, as expected. Next, we planned on the Approximation level and compared predicted to actual execution time. Out of 30 runs, 21 failed, mainly because: (a) the position of the object or the placing position was not actually reachable with the required hand orientation; or (b) the grasp failed, as we use only a simple version of the grasp planner at this stage; or (c) the object slipped out of the robot's hand at some stage. Part of the failure can be attributed to the physics simulator but because the Approximation level is optimistic, it is expected to be over-confident about plan viability. In all succeeded cases, execu-

tion time was 192 percent higher than predicted. This can mostly be attributed to the navigation prediction being based on a straight line path, not considering turns. We expect actual execution time to be even higher as soon as obstacles are introduced. It would be possible to improve the results by calculating at least a rough path on the Approximation Level for MOVE. Planning on the On-Board Level is not viable in reasonable time, due to the complexity of the algorithms involved. However, a plan made on the Approximation Level can be *evaluated* on the On-Board level. Taking the prediction of such evaluation, 11 of the 30 plans were ruled out. When executing the remaining 19 plans, 11 failed, mainly due to the object slipping out of the hand. Execution time when successful was only 35 percent higher than predicted by the On-Board level, mainly due to path corrections and the navigation taking time to get unstuck. This last set of experiments verifies that the data flow, precondition and postconditions of our Robot Capability Model, are viable to execute a plan on a real robot. In future work we want to model parameters for more robot models and run more complex scenarios.

6 Conclusion

We present a model of robot capabilities which supports a variety of robotic tasks and can be used as a planning domain. To integrate this model with a domain-independent planner, we present a novel approach to instantiate actions during planning and to support concurrent actions based on data flow including cyclic dependencies. Results show that the planner can produce a plan in an acceptable number of node expansions. Our ROS implementation of the capability interfaces can successfully execute the plan on a robotic system.

REFERENCES

- [1] J. Buehler and M. Pagnucco, 'A Framework for Task Planning in Heterogeneous Multi Robot Systems Based on Robot Capabilities', in *28th AAAI Conference on Artificial Intelligence (AAAI)*, (2014).
- [2] J. Chen and D. Sun, 'An online coalition based approach to solving resource constrained multirobot task allocation problem', in *International Conference on Robotics and Biomimetics*, pp. 92–97, (2010).
- [3] A. J. Coles, A. I. Coles, M. Fox, and D. Long, 'COLIN: planning with continuous linear numeric change', *JAIR*, 1–96, (2012).
- [4] M. Do and S. Kambhampati, 'SAPA: a multi-objective metric temporal planner', *Journal Of Artificial Intelligence Research*, 155–194, (2003).
- [5] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, 'Semantic Attachments for Domain-Independent Planning Systems', in *Towards Service Robots for Everyday Environments*, 99–115, (2012).
- [6] M. Fox and D. Long, 'PDDL2.1: an extension to PDDL for expressing temporal planning domains', *JAIR*, 61–124, (2003).
- [7] C.-H. Fua and S. Ge, 'COBOS: cooperative backoff adaptive scheme for multirobot task allocation', *IEEE T-RO*, 1168–1178, (2005).
- [8] T. Keller, P. Eyerich, and B. Nebel, 'Task planning for an autonomous service robot', in *KI 2010*, 358–365, Springer, (2010).
- [9] D.L. Kovacs, 'A multi-agent extension of PDDL3.1', in *3rd Workshop on the International Planning Competition (IPC)*, (2012).
- [10] Drew V. McDermott, 'Reasoning about autonomous processes in an estimated-regression planner', in *ICAPS*, pp. 143–152, (2003).
- [11] M. Molineaux, M. Klenk, and D. Aha, 'Planning in Dynamic Environments: Extending HTNs with Nonlinear Continuous Effects', in *AAAI*, (2010).
- [12] D. Nau, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman, 'SHOP2: an HTN planning system', *JAIR*, 379–404, (2003).
- [13] L.E. Parker and F. Tang, 'Building multirobot coalitions through automated task solution synthesis', *IEEE*, 1289–1305, (2006).
- [14] M. Di Rocco, F. Pecora, P. Sivakumar, and A. Saffiotti, 'Configuration planning with multiple dynamic goals', in *AAAI Spring Symp.*, (2013).
- [15] K. Wurm, C. Dornhege, B. Nebel, W. Burgard, and C. Stachniss, 'Coordinating heterogeneous teams of robots using temporal symbolic planning', *Autonomous Robots*, 277–294, (2013).
- [16] Nello Zuech and Richard K. Miller, *Machine Vision*, Springer, 1989.