A Framework for Task Planning in Heterogeneous Multi Robot Systems based on Robot Capabilities

Jennifer Buehler and Maurice Pagnucco

School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
{jennerb,morri}@cse.unsw.edu.au

Abstract

In heterogeneous multi-robot teams, robustness and flexibility are increased by the diversity of the robots, each contributing different capabilities. Yet platform-independence is desirable when planning actions for the various robots. We propose a platform-independent model of robot capabilities which we use as a planning domain. We extend existing planning techniques to support two requirements: generating new objects during planning; and, required concurrency of actions due to data flow which can be cyclic. The first requires online action instantiation, the second a small extension of the Planning Domain Definition Language (PDDL): allowing predicates in continuous effects. We evaluate the planner on benchmark domains and present results on an example object transportation task in simulation.

Heterogeneous multi-robot systems are characterized by the robots’ diversity, each contributing different capabilities. One important aspect of such systems is task planning, which involves finding sequences of actions required by one or several robots to solve a task. Most commonly, a planning domain is designed to fit a set of specific tasks or certain experiments. This hinders portability of such robot systems into new domains. We define a planning domain based on robot capabilities which is flexible enough to express different types of robot tasks in different environments (household, rescue, etc) and then use this domain for task planning.

Actions producing data are a natural property of robotic algorithms interacting with each other. For example, the Robot Operating System (ROS) builds on standardized message exchange between programs. From the planning perspective, each robot algorithm or program may be seen as an action. To reflect data flow between robot actions, the following factors need to be considered in the planner: (1) An action needs to be able to generate new data which can then be passed to another action. For instance, grasp planning will yield a position for the arm’s end effector, which is then used to compute the arm’s reaching motion. To allow passing data between actions, variables need to be instantiated during planning, i.e., binding action \texttt{reach(?pose)} to action instance \texttt{reach(new-grasp-\textit{pose}).} However, the common approach in current planners is to pre-generate all possible action instances with pre-defined objects, e.g., \texttt{reach(pose-1), reach(pose-2), etc.} This requires pre-computing all possible new data objects (pose-1 etc.). In the robotic domain, this can be prohibitive. That is, all possible object positions need to be considered for computing all grasp poses, as an object could have been moved earlier in the plan. (2) Data flow between concurrently executing actions has to be clearly represented, and can include cycles. For example, localisation and navigation run in parallel and share positional and odometry data. Such data flow dependencies imply that concurrency be required, i.e., due to the data flow requirements, localisation has to run concurrently with navigation. In the robotic domain, it seems natural to encode concurrency with data flow requirements.

The aim of this paper is to propose a platform-independent model of robot capabilities which can be used for planning and execution of robot tasks. This paper contributes an extension of existing planning techniques to support this robot domain. We address the abovementioned requirements by: (1) developing an algorithm to achieve online action instantiation; and, (2) handle required concurrency by extending existing techniques, proposing a small extension to the Planning Domain Definition Language (PDDL): allowing predicates in \textit{over all} effects.

This paper is structured as follows. First, we review relevant aspects and work. Then, we describe our robot capabilities and our temporal planner, and finish by presenting results on a series of experiments evaluating our planner.

Relevant aspects and background

Capability model Most work on formalizing robot capabilities focusses on the computation of a \textit{utility} value which expresses a robot’s “expected quality of task execution” (Korsah, Stentz, and Dias 2013). Capabilities are related to some kind of \textit{resource}, e.g., sensors/actuators, processing capacities (He and Ioerger 2003; Chen and Sun 2010), and/or software modules (Parker and Tang 2006). A capability can also be a simple \textit{subtask}, for which each robot learns their suitability (Fua and Ge 2005). Other research also formalizes capabilities relating to robot \textit{components} to infer what a robot can do (Kunze, Roehm, and Beetz 2011) or how to decompose a task into simple ‘skills’ (Huckaby and Christensen 2012). The Object Interaction Language (OIL) (Sutton et al. 2010) describes a “functionality” by \textit{code snippets}.
Task planning The planning problem is NP-complete (Erol, Nau, and Subrahmanian 1995), therefore all planning algorithms must resort to search. There is a plethora of algorithms addressing this problem (Malik Ghallab and Paolo Traverso 2004). A popular and successful approach has been to use forward-search in the state space, often with a planning graph based heuristic. Examples are Fast Forward (Hoffmann and Nebel 2001), Fast Downward (Helmert 2006), TLPlan (Bacchus and Ady 2001), SAPA (Do and Kambhampati 2003), Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009), LPG (Gerevini and Serina 2002), Crikey (Coles et al. 2009) and COLIN (Coles et al. 2012). None of these supports online action instantiation or cyclic dependencies in concurrent actions’ data flow.

Configuration plans are generated in (Di Rocco et al. 2013), similar to ASyMTRe (Parker and Tang 2006). Time is represented by constraints over the “activities”, which have to be re-defined for each specific task. We aim at a more general, domain-independent approach which suits many tasks.

Few approaches in robotics use a domain-independent planning system. Temporal Fast Downward (TFD) is extended in (Keller, Eyerich, and Nebel 2010), building on their previous continual planner. The special feature is their “global memory” in the planner, i.e., if the same action fails several times, it is not considered at the re-planning stage. Also (Wurm et al. 2013) use the TFD/M planner to coordinate heterogeneous teams of robots in exploration and transportation tasks. The planning is done using symbolic actions which are specified in PDDL by simple propositions, specifically relating to the transportation task. Essentially this designs the PDDL domain for a particular task, as the exploration and transportation task. Instead, we aim at designing a PDDL domain which is general for all robotic tasks.

Temporal planning with concurrency To find a plan for robots to co-operate towards a goal, time is an essential factor, as the aim is to minimise the time it takes the robot(s) to perform the task(s). PDDL2.1 (Fox and Long 2003) introduces durative actions to account for time. Most existing temporal planners build on PDDL’s durative actions.

Planning systems for real-world applications also need the ability to handle concurrency. One very popular approach to handle concurrency during planning is Decision Epoch Planning, as used in TLPlan (Bacchus and Ady 2001), SAPA (Do and Kambhampati 2003) and Optop (McDermott 2003). This idea introduces a wait action which simulates the passing of time. In every state there is a set of active process instances (concurrently running actions). The wait action moves time forwards until the next relevant time point, e.g., when a process ends. Concurrency is interleaved, meaning the actual starting/ending of actions happens sequentially and only the actual execution of several active processes happens concurrently. Decision epoch planners are temporally incomplete for some planning problems requiring complex temporal interactions, though (Cushing et al. 2007) proposes a solution for this limitation. Our robot tasks do not require such complex interactions at this stage and we also chose to implement our planner based on Decision Epoch Planning. However, we need to adapt the concept to support our data flow requirements.

MA-PDDL (Kovacs 2012) encodes concurrency requirements in PDDL. They use a generalisation of the concurrent action lists in (Boutilier and Brafman 2001). Essentially, this approach lists specific actions which have to run in parallel. However, in robotics an algorithm typically needs data as input. It does not matter which action generated it. For example, the localisation algorithm does not care whether the 3D point cloud comes from a laser scan or a stereo camera—the data could even come from another robot. This observation is also made in ASyMTRe (Parker and Tang 2006) which connects action schemas by matching their data input/output in their coalition formation algorithm. We connect capabilities’ data input/output using a temporal planner instead.

Information generation during planning In PDDL, it is not possible to specify that performing an action causes new objects to be created. For example, the result of an action GRASP_PLAN (which plans the grasp of an object) will result in a position \( p = (x, y, z) \) to which the manipulator is to be moved. This new data object, unique to the specific grasping problem, is then passed to the REACH(?position) action by binding ?position to \( p \). Passing information from plan step to plan step has been recognized as a relevant factor (McDermott 1978) but has received little attention. To pass a new data object to an action, actions have to be instantiated at planning time (online). However, the common approach is to avoid variables altogether by substituting them in all possible ways before the search. The only recent planner that binds variables during planning time is Optop (McDermott 2003), which is a regression-based planner. The extra effort to bind variables during planning makes it slower than other algorithms but instead it is able to handle infinite domains such as numbers (for example, an action act(?a -number) would not be allowed in planners which enumerate all possible actions in advance). Unfortunately, their regression-based algorithm is incomplete due to binding variables such that the maximum number of goal conjuncts are satisfied. We develop an algorithm for action instantiation during forward-search without such restrictions in variable bindings.

Defining Robot Capabilities

We define capability in a manner that abstracts from hardware and software, allowing a robot to learn its constraints relating to the capabilities. A capability is a simple functional element which can be part of many different tasks, inspired by Zuech and Miller (Zuech and Miller 1989, 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...”

With this definition, a capability abstracts from underlying architectures at a medium level of granularity. For example, to pick up an object, any robot needs to REACH for it, GRASP and then LIFT it. It is not important how a robot grasps an object (e.g., which finger movements) but only what it can probably grasp. The overall idea is that a ca-
pability can be part of many different tasks, either (a) representing a primitive \textit{physical action} as, for example, \textsc{reach}, 
\textsc{grasp}, \textsc{push}, or (b) a \textit{sensing} activity which produces data (e.g., \textsc{vision}), or (c) is an \textit{algorithm} or \textit{computational} entity as \textsc{path-planning}, \textsc{stereo-matching}, etc.

A complete ontological discussion of the individual capabilities cannot be given in the space of this paper. Therefore, we shall only briefly describe the relevant aspects for task planning. Because we define our domain in PDDL, the notation for variables (?var) is used in the following. A capability \( C = \{ \text{?var} \} \) is used in the following. A capability \( C = \{ \text{?var} \} \) has preconditions \( \text{?pre} \) and effects \( \text{?eff} \), as common for actions in planning. For example, an effect of \textsc{reach} is that the manipulator is at-\text{Pose}(\text{?manip-id}, \text{?pose}), which is also the precondition for \textsc{grasp}. The effect of \textsc{grasp} is that the object is held(\text{?manip-id}, \text{?object}), which is also the precondition for \textsc{lift}, which in turn has effect not(grounded(\text{?object})). Goals are specified with conditions. For example, picking up an \textit{object} is specified by the goal that the object is not(grounded(\text{?object-\text{id}})). A solution is to connect \textsc{reach}, \textsc{grasp} and \textsc{lift}. However, this is not sufficient to reflect the data flow. In this example, we need to compute a specific \textit{pose} where to reach so the object can be grasped. This is a computational process related to the grasp action: the grasp planning. Each physical action may have a computational \textit{planning counterpart} which can appear earlier in the plan, i.e., \textsc{grasp-plan} for \textsc{grasp}. Now, the correct action sequence is \textsc{grasp-plan}, \textsc{reach}, \textsc{grasp} and \textsc{lift}, with data being passed between \textsc{grasp-plan} and \textsc{reach}. To capture the data flow, capabilities have input (I) and output data (O) (e.g., images, maps, poses, etc.), similar to ASyMTRe (Parker and Tang 2006) and (Di Rocco et al. 2013) which connect \textit{schemas} or \textit{activities} with matching information types. Data inputs/outputs serve as additional constraints on how capabilities can be connected. Furthermore, hardware dependencies (hw) are listed for each capability. Each device is indexed by a unique number (ID) on a robot. By using this ID in an action’s parameter, we can ensure consistency in hardware use, i.e., the same manipulator is used for reaching and grasping. The resulting plan efficiently captures the hardware devices required, which can be useful in evaluating plan quality. Capability \textit{parameters} (px) specify approximate execution constraints which further help ruling out a robot’s eligibility for a specific task. For example, areas that a robot can \textsc{reach} are approximated with spherical shapes around the manipulators; terrain on which a robot can \textsc{move} may be described by indices of “terrain roughness” with assigned average speeds; object sizes it can \textsc{grasp} may be approximated with a bounding volume; weights it can \textsc{lift} with a maximum bound, and so on.

### Planning with Capabilities

We implemented a decision epoch planner similar to \textsc{SAPA}. In \textsc{SAPA}, preconditions \texttt{Pre}_{\text{a}} and \texttt{Pre}_{\text{c}} are required to be true at the starting/ending point \( t_0, t_e \) of the action of duration \( D_a \), while \textit{persistent conditions} \texttt{Pre}_{\text{p}} should hold true throughout some duration \( d \leq D_a \). Effects \texttt{Eff}_{\text{s}} \texttt{Eff}_{\text{d}} \texttt{Eff}_{\text{e}} occur at the start or end time points \( t_0, t_e \), or at predefined time points \( t_d \) within the action duration. We adopt a similar approach, extending it by supporting \textit{persistent effects} \texttt{Eff}_{\text{p}} which constantly happen throughout the \textit{whole} execution duration \( D_a \) of the action (similar to the persistent precondition which needs to hold throughout the action execution). Additionally, we need to modify the \textsc{SAPA} algorithm to support \textit{cyclic dependencies} in the data flow.

We keep the following data structures for describing a search \( S = (E, t, \text{Facts}, \tau, \gamma, \text{Ext}) \):

- \( E \): An \textit{event queue} containing all (ground) actions that are currently being concurrently executed;
- \( t \): time elapsed since the start of the plan;
- \( \text{Facts} \): a set of predicate and function values which hold at time \( t \)
- \( \tau \): a set of propositions which describe the persistent action preconditions of all actions in \( Q \)
- \( \gamma \): a set of propositions which describe the persistent action effects of all actions in \( Q \); and,
- \( \text{Ext} \): External module data (detailed on page 5).

Using this state description, we perform a forward search through the state space, expanding each node by evaluating for each state \( \text{S} \) which actions are applicable. An action \texttt{A} is applicable to be started in state \( \text{S} \) if and only if:

1. All starting preconditions \texttt{Pre}_{\text{a}} are fulfilled by \texttt{Facts};
2. The same action \textit{instance} \texttt{A} is not already in \( Q \) (nb. the same action can run twice but with different parameters);
3. The external planning module determines \texttt{A} to be eligible \( S \) (as discussed on page 5); and,
4. The starting effect \texttt{Eff}_{\text{s}} does not interfere with any of the conditions in \( \tau \) or persistent effects in \( \gamma \). Interference of proposition \texttt{P}_i \in \texttt{Eff}_{\text{s}} means:
   - \( P_i \) is a predicate and its negation is contained in \( \tau \); or,
   - \( P_i \) is a function expression and it is writing the same function value as another proposition in \( \gamma \).

If an action is eligible to start in \( S \), we apply the effect \texttt{Eff}_{\text{s}} on \texttt{Facts} and add it to the event queue \( E \). The persistent conditions and effects are added to \( \tau \) and \( \gamma \).

#### Concurrency of actions

Concurrency can be represented with \textit{persistent preconditions} and the \textit{persistent effects} by including propositions that symbolise data flow, e.g., \texttt{data(?d - Image2D)}. Data consumption (persistent precondition) and production (persistent effect) are matched by the data object and thereby connect concurrent actions. Figure 1 shows example capabilities, represented as a box with input and output slots for conditions and effects. \texttt{p1} and \texttt{p2} are \textit{persistent effects} of actions \texttt{B} and \texttt{A}. The persistent precondition of \texttt{B} (\texttt{A}) is the availability of data object \texttt{p2} (\texttt{p1}). We connect persistent effects to persistent preconditions to form a valid data flow. In the example, \texttt{p1} connects \texttt{B} to \texttt{A}, and \texttt{p2} connects \texttt{A} to \texttt{B}, forming a cyclic data flow. A more specific example is the data flow between moving, localising and path planning (which continuously corrects the path). For \textit{persistent effects} we need to extend PDDL to allow effects within over all.
To support cyclic dependencies, we need to adopt a new interpretation of the \textit{wait} action: actions can be in one of three states: \texttt{start}(s), \texttt{running}(e) and \texttt{end}(e). When starting an action \(A\), it is flagged as \texttt{start}(A) and added to the queue \(E\). Time only moves forwards once \textit{wait} is applied—it "activates" all actions \(A_S\) in \(E\) and switches their flags from \texttt{start} to \texttt{running}(A_s). \textit{Wait} is only applicable if all actions \(A_S\) and \(A_e\) in \(E\) have their persistent conditions fulfilled. To check this, we first apply all the persistent effects \(\gamma_s, \gamma_r\) of all \(A_S, A_e\) on \texttt{Facts} and then check if all the persistent conditions \(\tau\) of all actions in \(E\) are fulfilled with \texttt{Facts}. If they are not, we have reached a dead end and the planner has to backtrack, possibly starting actions which provide the missing persistent effects. Only once all persistent conditions are met, can we advance time. The key point is that persistent effects and preconditions of concurrent actions will be added at the same time (at the \textit{wait} action), and not after one another, which allows cyclic dependencies. When \textit{wait} has found that an action \(A_e\) has ended, it will be flagged \(A_e\). At each node expansion, if there are any such ended actions, ending them is mandatory before starting any other actions or \textit{wait}. Some actions may have undefined durations: they end once their persistent preconditions are not fulfilled any more (e.g., Localisation and Path Planning end once the Moving has ended).

This implementation implies the following limitation: we cannot allow two concurrent actions which write the same function value, else the function value is not well-defined.

**Online action instantiation** At each plan state \(S\), we instantiate only actions which are eligible in the current planning state. Just as in \textit{Optop} (McDermott 1999), all variables which appear in the \texttt{parameters} of the action must also appear in the \texttt{:condition}. To determine whether an action is eligible, we need to (1) instantiate its precondition \texttt{Pre} using a variable substitution \(s\), and (2) find a proof of \texttt{Pre} given \(s\) (abbreviated \texttt{Pre \lvert s}) with the current proposition set \texttt{Facts}. We do this proof using \texttt{unification}. When searching for all \(s\) such that \texttt{Facts \models Pre \lvert s}, we obtain not only the proof of \texttt{Pre}, but also all possible variable substitutions \(s\) which instantiate the action such that it is eligible in the planning state. Unification in the common definition means that two expressions’ variables can be substituted in such a way that the expressions are exactly equal. We introduce \texttt{subset unification}, meaning that the expressions do not have to be exactly the same but one expression is a subset of the other. For example, in the two expressions
\[
E_1 &= \text{and}((\text{atom1}(?v1, o2))
E_2 &= \text{and}((\text{atom1}(o1, o2)) (\text{atom2}(o3, o2))
\]
\(E_1\) is as a subset of \(E_2\) for either of the substitutions \(s_1=\{?v1/o1\}, s_2=\{?v1/o3\}\). If \(E_1\) and \(E_2\) are conjunctions, then \(E_1 \equiv E_2 \equiv E_1 \lor E_2\), meaning we can prove \(E_1\) from \(E_2\). \texttt{Facts} is always a conjunction and, if we convert \texttt{Pre} to Disjunctive Normal Form (DNF), we can find \texttt{alternative} proofs for each conjunction \(P \in \texttt{Pre}_{dnf}\). Given the conjunctions \(P\) and \texttt{Facts}, we can find possible variable substitutions \(S_i = \{s_1...s_n\}\) for propositions \(p_i \in P\), such that \(\texttt{Facts} \models p_i \text{ } \forall s \in S_i\). Algorithm 1 sketches a function which returns possible substitutions (or the empty set) for a \texttt{predicate} \(p_i\) such that \texttt{Facts} \models p_i (functions and numerical, quantified and conditional expressions are also supported but cannot be detailed within this space). The function takes a list of given substitutions \texttt{givenS} (results of call to \texttt{unifyPredicate} for propositions \(p_i\) \text{ } j < i\) earlier in the conjunction \(P\)). Each substitution \(ps \in \texttt{givenS}\) may have to be \texttt{extended} to match the current predicate \(p_i\). The function \texttt{match} in line 9 returns the variable substitution \texttt{matchSub} to make two predicates equal. If a unification of \(p_i\) with a \texttt{fact \in facts} is possible, \texttt{matchSub} is added to \texttt{ps} (line 11), and the extended \texttt{ps} is added to the result set \(S\) (line 12). Using the closed world assumption, negated predicates should find no unification with the \texttt{Facts}. Hence, we only keep \texttt{ps} if we do not find a disproof with it (line 14).

The result of our algorithm yields a list of substitutions \(S = \{s_1...s_n\}\). If not empty, each \(s \in S\) instantiates the action such that it is eligible in the current planning state.

**Application to capabilities** We can now define a PDDL domain in which \texttt{actions produce data}. The example of planning a grasping action will produce a position \(p=(x, y, z)\) where to move the manipulator. Evidently, PDDL is not expressive enough to simulate such a complex process as grasp planning. This has to be done in \texttt{external modules}, while the specification in PDDL only encodes the \texttt{symbolic action} and the \texttt{symbolic information generation} (i.e., the grasping planning and the manipulator position). In the planner, we represent symbolic information by generating an \texttt{Object ID} for it. The
PDDL definition can then abstract from the actual data (i.e., the coordinates) the external module outputs. For example, let’s consider a symbolic action to plan a grasp:

```
[action graspPlan
 :parameters (?obj ?objectPos)
 :precondition ( and
 (= (objectAtPos ?obj) ?objectPos)
 (= (idGenerator) ?armPosID) ...)
 :effect ( and
 (reachLocation ?obj ?armPosID)
 (increase (idGenerator) 1) ... )
)
```

Each time a grasp plan is performed, we increase the object ID (incrementing the value of idGenerator). We add a predicate reachLocation to symbolise data creation, which as parameter takes the object ID (?armPosID is unified with the last idGenerator value in the precondition).

The durative action REACH has reachLocation in its precondition, which will then be bound to the current ID. This way, data flow amongst actions is symbolically represented, and repeated action calls are supported due to the index generation. Detailed data is kept in the external module, indexed by the ID’s. Evidently, the illustrated graspPlan needs to be extended to fully represent its semantics, but to illustrate the idea this simplified example should suffice.

**External modules** To determine a robot’s eligibility for a specific task, we may need to evaluate conditions and compute time estimates which require more complex algorithms that cannot be represented in PDDL. TFD/M (Dornhege et al. 2012) addresses the lack of PDDL expressiveness by introducing external modules in which function calls to external libraries can be specified within PDDL. We adopt a similar approach, with the difference that we introduce external modules per action, instead of arbitrary function calls within PDDL definitions. This more naturally suits our needs and does not require changes in the PDDL specification. The external module of an action is called from within the planner to (a) retrieve the action’s duration (e.g., calculating time of navigation along a path), (b) to check for more complex preconditions (e.g., concerning capability parameters mentioned earlier), and (c) when applying effects of an action in the current planning state. The latter is mainly used to allow the external module to create data structures that have been generated by the action (e.g., storing the x/y/z coordinates indexed by the generated object ID). Such data has to be passed from one planning state to the next so the module can use it for their calculations. The reference Ext to external module data is passed along with the current state.

**Results**

We evaluate our algorithms using our implementation of the planner on a series of experiments. First, we want to compare the performance of our runtime action instantiation (online instantiation) to the traditional approach of pre-instantiating all actions in advance (offline pre-instantiation). Our planner can also do offline instantiation—but domains with numeric action parameters (like our capabilities) will not work with the offline approach, because we cannot pre-instantiate such actions (which is the reason why we needed to implement the online instantiation). However, we can evaluate the performance of offline vs. online on other classic domains like the well-known FF benchmark domains (Hoffmann 2001). We used depth-first search only in order to reflect raw runtime performances regardless of heuristics. Results are shown in Table 1. Domain name subscripts in the table list the parameters for problem generation (cf. FF domains documentation). #Act is the number of action instances which were pre-generated for the offline case and is the number of domain actions in the online case. #Time does not include the time to generate action instances in offline, which for some problems was significant (e.g., for Grid).

The main observations which can be made from this table are as follows. (a) The number of action instances can become huge in the offline case, which is clearly reflected in the search time, because all these actions have to be evaluated for applicability in each state. In particular, the results on the Gripper domain show a big difference between both approaches. (b) Sometimes the offline approach can result in better performance, as seen in the ferry domain. This domain has a larger number of predicates in the problem description (Facts in our state), which are used to instantiate domain actions, hence the larger state description affects search performance. (c) In some cases, we were not able to run the offline variant because the planner ran out of memory at the stage of pre-generating action instances, which was the case in the Grid domain (and others we tested which are not shown here). Overall, the results show that, in most cases, there is no disadvantage of action instantiation at planning time but that there can even be performance improvements, in some cases solving problems requiring too much memory for the offline approach.

Next, we ran experiments with our robot capabilities defined in PDDL. Apart from the planning actions, all are durative actions. For the experiment, 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 1. To pick an object, the robot has to GRASPPLAN to obtain a target arm position armPose for the grasp, which is passed to REACHPLAN to determine the robot’s standing pose...

<table>
<thead>
<tr>
<th>Domain</th>
<th># Exp</th>
<th>Offline # Act</th>
<th>Offline Time</th>
<th>Online # Act</th>
<th>Online Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Briefcase_4</td>
<td>6738</td>
<td>49</td>
<td>1.7</td>
<td>3</td>
<td>1.1</td>
</tr>
<tr>
<td>Briefcase_5</td>
<td>7856</td>
<td>71</td>
<td>6.6</td>
<td>3</td>
<td>3.6</td>
</tr>
<tr>
<td>Ferry_5,15</td>
<td>25156</td>
<td>300</td>
<td>4.9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Ferry_6,15</td>
<td>134543</td>
<td>363</td>
<td>22</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>Gripper_e</td>
<td>8957</td>
<td>2100</td>
<td>9.2</td>
<td>3</td>
<td>2.7</td>
</tr>
<tr>
<td>Gripper_e</td>
<td>23689</td>
<td>2783</td>
<td>31.9</td>
<td>3</td>
<td>5.8</td>
</tr>
<tr>
<td>Gripper_e</td>
<td>60213</td>
<td>3600</td>
<td>100.5</td>
<td>3</td>
<td>14.9</td>
</tr>
<tr>
<td>Gripper_e</td>
<td>149322</td>
<td>4563</td>
<td>306</td>
<td>3</td>
<td>41.4</td>
</tr>
<tr>
<td>Grid_5,2,11</td>
<td>325</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>7.3</td>
</tr>
<tr>
<td>Log_1,2,3</td>
<td>2900</td>
<td>67500</td>
<td>23.7</td>
<td>6</td>
<td>2.3</td>
</tr>
<tr>
<td>Log_1,2,3</td>
<td>202117</td>
<td>29376</td>
<td>82</td>
<td>6</td>
<td>5.6</td>
</tr>
</tbody>
</table>

Table 1: Results on FF domains. Node expansions (# Exp), (generated) actions (# Act) and search time in seconds.
standPose to reach to armPose. LOCALISE, PATHPLAN and MOVE bring the robot to a target position standPose. PLA-
CPLAN determines an arm pose to place an object, similar to GRASPPLAN. LIFT and PLACE are symbolic actions which
attach/detach the object from the robot. Unfortunately we do
not have the space to include the PDDL definitions but the
principle has been explained in the above sections. Robots
are generated in a 10x10 grid world in which they can move
vertically and horizontally. A number of objects which have
to be transported to a random target location are generated as
weel. We want to see how many nodes have to be expanded
until a plan is found and how long it takes all robots to trans-
port all objects (total task time). We compare results of an
optimal A-Star search with a Greedy Best First search. Task-
dependent heuristic computation is part of our future work,
therefore the heuristic we currently use is simply the max-
imum distance of any object to its target location, assuming
all robots move at the same speed (this is admissible because
it assumes all objects are transported in parallel). The dura-
tion of all manipulation actions is 1 second but the MOVE
action depends on the path calculated in the PATHPLAN ac-
tion, which for these experiments returns the Manhattan dis-
tance. We generated 100 such problem instances and aver-
age the results as displayed in Table 2. The number of node
expansions increases quite dramatically with more robots
and objects being introduced. This is not surprising, because
temporal planning, particularly with interleaved and concur-
current actions, is much more complex than classical planning:
Scheduling is part of the problem and, in this example, also
the Traveling Salesman Problem (TSP), which is NP-Hard,
implicitly is part of the problem. This is because the shortest
path to all objects needs to be found. Therefore, we cannot
use this planner for large instances of robots and objects.
However, this is not our aim. We require a planner to yield
action sequences for one robot or a smaller number of robots
co-operating at one task (also termed a Coalition (Korsah,
Stentz, and Dias 2013)). The number of node expansions
needed is acceptable for a small number of robots/objects,
especially considering that the heuristic used in this exa-
ample is very simple. If we compare the results of A-Star and
Best First, 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.
Table 3 shows another run of the experiment with only one
robot or one object respectively. It shows clearly that perfor-
mance is not heavily impacted by having several robots com-
peting for the same object. The total task time decreases with
the number of robots because the closest robot will transport
the object but the number of node expansions does not in-
tcrease too much. Instead, if we have one robot which has to
transport several objects, a dramatic increase in the search
space can be observed. This is because the TSP implicitly
becomes part of the problem. From this, we can draw the
conclusion that our planner works well for several robots
and few location-dependent tasks. Our planner finds an ac-
ction sequence for the robot which is best suited (according
to execution prediction) for the task.

In the last experiment, we assume robots cannot move ev-
everywhere in the world because they cannot navigate certain
terrain. We divide the grid into two different terrain shapes;
one half of the grid each. Robots can place an object at the
boundary to the other terrain, so that another robot can pick
it up and bring it to its destination. Now, robots need to co-
operate in order to solve the task. For our search algorithm
to work, positions in the world have to be tagged as “plac-
ing positions”. This is necessary to restrict the search space,
otherwise robots would have to try placing objects at every
possible location in the world, which would make the plan-
ning problem intractable in continuous domains. Such place
positions could be automatically detected, e.g., positions on
tables, or on other flat surfaces. For this experiment, we gen-
erate such place positions randomly at the boundary of the
two terrains. We want to see how the number of such place
positions influences node expansions. Table 4 shows the re-
results of this experiment. As expected, the search space in-
creases with more place positions, because robots have more
choices. At the same time, overall task runtime decreases, as
robots find more optimal place positions.

| #Robots – | AStar | Best First |
| #Objects | # Exp | Task time | # Exp | Task time |
| 1 – 1 | 78 | 18.1 | 78 | 18.1 |
| 1 – 2 | 239 | 35.2 | 167 | 37.0 |
| 1 – 3 | 690 | 51.1 | 303 | 52.7 |
| 2 – 1 | 83 | 16.7 | 85 | 18.8 |
| 2 – 2 | 1876 | 20.1 | 826 | 34.4 |
| 2 – 3 | 5764 | 29.2 | 1483 | 49.8 |
| 3 – 1 | 105 | 15.7 | 84 | 20.4 |
| 3 – 2 | 3463 | 17.7 | 804 | 33.5 |

Table 2: Results for the object transportation task.

| #Robots – | AStar | Best First |
| #Objects | # Exp | Task time | # Exp | Task time |
| 1 – 1 | 78 | 18.0 | 78 | 19.9 |
| 3 – 1 | 105 | 16.1 | 84 | 19.6 |
| 5 – 1 | 132 | 15.0 | 85 | 20.9 |
| 8 – 1 | 205 | 14.3 | 88 | 19.8 |
| 10 – 1 | 259 | 13.4 | 91 | 19.5 |

Table 3: Result set 2 for object transportation task.

| #Robots – | AStar | Best First |
| #Places | # Exp | Task time | # Exp | Task time |
| 2 – 1 | 171 | 29.2 | 169 | 33.2 |
| 5 – 1 | 298 | 26.8 | 158 | 30.5 |
| 10 – 1 | 996 | 23.9 | 206 | 31.6 |

Table 4: Object transportation task with need for co-
operation.
Conclusion

We present a domain description supporting a variety of robotic tasks, with robot capabilities defined in a platform-independent way to support heterogeneous architectures. To integrate this model with a domain-independent planner, we present a novel approach to instantiate actions during planning and support concurrent actions with cyclic data flow dependencies. Results show that this planner can produce an optimal plan in an acceptable number of node expansions.

References


