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
{jenniferb,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 `reach(?pose)` to action instance `reach(new-grasp-pose)`. However, the common approach in current planners is to pre-generate all possible action instances with pre-defined objects, e.g., `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 `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 *utility* value which expresses a robot's "expected quality of task execution" (Korsah, Stentz, and Dias 2013). Capabilities are related to some kind of *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 *subtask*, for which each robot learns their suitability (Fua and Ge 2005). Other research also formalizes capabilities relating to robot *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 *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 capability can be part of many different tasks, either (a) representing a primitive *physical action* as, for example, REACH, GRASP, PUSH, or (b) a *sensing* activity which produces data (e.g., VISION), or (c) is an *algorithm* or *computational* entity as PATH-PLANNING, 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 = {P,E,I,O,hw,pr}` has *preconditions* `P` and *effects* `E`, as common for actions in planning. For example, an effect of REACH is that the manipulator is `at-Pose(?manip-id,?pose)`, which is also the precondition for GRASP. The effect of GRASP is that the object is `held(?manip-id,?object)`, which is also the precondition for LIFT, which in turn has effect `not(grounded(?object))`. Goals are specified with conditions. For example, *picking up an object* is specified by the goal that the object is `not(grounded(object-id))`. A solution is to connect REACH, GRASP and LIFT. However, this is not sufficient to reflect the data flow. In this example, we need to compute a specific *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 *planning counterpart* which can appear earlier in the plan, i.e., GRASP-PLAN for GRASP. Now, the correct action sequence is GRASP-PLAN, REACH, GRASP and LIFT, with data being passed between GRASP-PLAN and 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 *schemas* or *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 *parameters* (`pr`) specify approximate execution constraints which further help ruling out a robot's eligibility for a specific task. For example, areas that a robot can REACH are approximated with spherical shapes around the manipulators; 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; weights it can LIFT with a maximum bound, and so on.

Planning with Capabilities

We implemented a decision epoch planner similar to *SAPA*. In *SAPA*, preconditions Pre_s and Pre_e are required to be true at the starting/ending point t_s / t_e of the action of duration D_a, while **persistent conditions** Pre_p should hold true throughout some duration $d \leq D_a$. Effects Eff_s, Eff_d Eff_e occur at the start or end time points t_s, t_e, or at predefined time points t_d within the action duration. We adopt a similar approach, extending it by supporting **persistent effects** Eff_p which constantly happen throughout the *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 SAPA algorithm to **support cyclic dependencies** in the data flow.

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

- E: An *event queue* containing all (ground) actions that are currently being concurrently executed;
- t: time elapsed since the start of the plan;
- $Facts$: a set of predicate and function values which hold at time t. This is essentially the current state of the world;
- τ: a set of propositions which describe the persistent action preconditions of *all* actions in Q. This is a collection of predicates and function values;
- γ: a set of propositions which describe the persistent action effects of *all* actions in Q; and,
- 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 S which actions are applicable. An action A is applicable to be started in state S if and only if:

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

If an action is eligible to start in S, we apply the effect Eff_s on $Facts$ and add it to the event queue E. The persistent conditions and effects are added to τ and γ.

Concurrency of actions Concurrency can be represented with *persistent preconditions* and the *persistent effects* by including propositions that symbolise data flow, e.g., `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. `p1` and `p2` are *persistent effects* of actions `B` and `A`. The persistent precondition of `B` (`A`) is the availability of data object `p2` (`p1`). We connect persistent effects to persistent preconditions to form a valid data flow. In the example, `p1` connects `B` to `A`, and `p2` connects `A` to `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 *persistent effects* we need to extend PDDL to allow effects within `over all`.

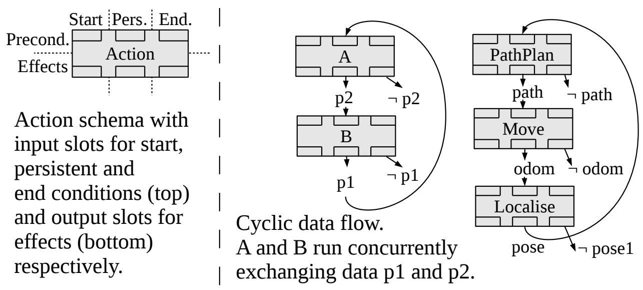
Start
Pers.
End.
Precond.
Action
Effects
Action schema with input slots for start, persistent and end conditions (top) and output slots for effects (bottom) respectively.
A
p2
¬ p2
B
p1
¬ p1
Cyclic data flow.
A and B run concurrently exchanging data p1 and p2.
PathPlan
path
¬ path
Move
odom
¬ odom
Localise
pose
¬ pose1

Figure 1: Cyclic dependencies of persistent condition/effects.

To support cyclic dependencies, we need to adopt a new interpretation of the *wait* action: actions can be in one of three states: `start(s)` , `running(r)` and `end(e)`. When starting an action A, it is flagged as `start` (A_s) and added to the queue E. Time only moves forwards once *wait* is applied—it "activates" all actions A_s in E and switches their flags from `start` to `running` (A_r). *Wait* is only applicable if all actions A_s and A_r in E have their persistent conditions fulfilled. To check this, we first apply all the persistent effects γ_s, γ_r (of all A_s, A_r) on $Facts$ and then check if all the persistent conditions τ of all actions in E are fulfilled with $Facts$. 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 *wait* action), and not after one another, which allows cyclic dependencies. When *wait* has found that an action A_r 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 *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 *Optop* (McDermott 1999), 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 given s (abbreviated $Pre \mid s$) with the current proposition set $Facts$. We do this proof using *unification*. When searching for all s such that $Facts \models Pre \mid s$, we obtain not only the proof of 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 **subset unification**, meaning that the expressions do

Algorithm 1 Subset unification

```
 1: function UNIFYPREDICATE($p_i$, facts, givenS)
 2:     S ← []
 3:     neg ← isNegated($p_i$)
 4:     if neg then $p_i$ ← negate($p_i$)
 5:     for all ps ∈ givenS do
 6:         $p_{i,inst}$ ← applySubstitution($p_i$,ps)
 7:         extensions ← []
 8:         for all fact ∈ facts do
 9:             matchSub ← match($p_{i,inst}$,fact)
10:             if $p_{i,inst}$ = fact ∨ matchSub ≠ [] then
11:                 $ps_{ext}$ ← ps ∪ matchSub
12:                 extensions ← extensions ∪ $ps_{ext}$
13:         if neg ∧ extensions = [] then
14:             S ← S ∪ ps
15:         else if ¬ neg ∧ extensions ≠ [] then
16:             S ← S ∪ extensions
        return S
```

not 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 as a subset of `E2` for either of the substitutions `s1=[?v1/o1], s2=[?v1/o3]`. If `E1` and `E2` are conjunctions, then $E2 \models E1 \mid s1 \vee s2$, 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 $P \in Pre_{dnf}$. Given the conjuncts P and $Facts$, we can find possible variable substitutions $S_i = [s_1, ...s_n]$ for propositions $p_i \in P$, such that $Facts \models p_i \mid \forall s \in S_i$. Algorithm 1 sketches a function which returns possible substitutions (or the empty set) for a *predicate* p_i such that $Facts \models p_i$ (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 $givenS$ (results of calling $unifyPredicate$ for propositions $p_j \mid j < i$ earlier in the conjunct P). Each substitution $ps \in givenS$ may have to be *extended* to match the current predicate p_i. The function $match$ in line 9 returns the variable substitution $matchSub$ to make two predicates equal. If a unification of p_i with a $fact \in facts$ is possible, $matchSub$ is added to ps (line 11), and the extended ps is added to the result set S (line 12). Using the closed world assumption, negated predicates should find no unification with the $Facts$. Hence, we only keep 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 **actions produce data**. The example of planning a grasp 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 **external modules**, while the specification in PDDL only encodes the *symbolic action* and the *symbolic information generation* (i.e., the grasp planning and the manipulator position). In the planner, we represent symbolic information by generating an **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 modules 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-instantiatiate such actions

		Offline		Online	
Domain	# Exp	# Act	Time	# Act	Time
Briefcase$_4$	6738	49	1.7	3	1.1
Briefcase$_5$	7856	71	6.6	3	3.6
Ferry$_{n5,l5}$	25156	300	4.9	3	6
Ferry$_{n6,l5}$	134543	363	22	3	42
Gripper$_6$	8957	2100	9.2	3	2.7
Gripper$_7$	23689	2783	31.9	3	5.8
Gripper$_8$	60213	3600	100.5	3	14.9
Gripper$_9$	149322	4563	306	3	41.4
Grid$_{x5,y5,k1,l1}$	325	-	-	5	7.3
Log$_{a1,s2,c2,p3}$	2900	67500	23.7	6	2.3
Log$_{a1,s2,c3,p2}$	202117	29376	82	6	5.6

Table 1: Results on FF domains. Node expansions (# Exp), (generated) actions (# Act) and search time in seconds.

(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 peformances 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

#Robots – #Objects	AStar # Exp	AStar Task time	Best First # Exp	Best First 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.

$standPose$ to reach to $armPose$. LOCALISE, PATHPLAN and MOVE bring the robot to a target position $standPose$. PLACEPLAN 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 well. We want to see how many nodes have to be expanded until a plan is found and how long it takes all robots to transport all objects (total task time). We compare results of an optimal A-Star search with a Greedy Best First search. Task-independent heuristic computation is part of our future work, therefore the heuristic we currently use is simply the maximum 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 duration of all manipulation actions is 1 second but the MOVE action depends on the path calculated in the PATHPLAN action, which for these experiments returns the Manhattan distance. We generated 100 such problem instances and averaged 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 concurrent 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 example 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 performance is not heavily impacted by having several robots competing for the same object. The total task time decreases with

#Robots – #Objects	AStar # Exp	AStar Task time	Best First # Exp	Best First 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
1 – 1	79	20.0	78	20.0
1 – 2	234	36.7	175	37.6
1 – 3	679	49.9	303	54.0
1 – 4	1738	66.9	494	74.4
1 – 5	4688	79.6	920	89.8

Table 3: Result set 2 for object transportation task.

#Robots – #Places	AStar # Exp	AStar Task time	Best First # Exp	Best First 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
2 – 3	317	28.7	231	35.2
5 – 3	1106	26.3	312	31.5
10 – 3	1517	21.0	385	26.1
2 – 5	604	26.0	342	31.9
5 – 5	1578	24.5	391	29.1
10 – 5	2652	19.3	553	30.1

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

the number of robots because the closest robot will transport the object but the number of node expansions does not increase 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 action sequence for the robot which is best suited (according to execution prediction) for the task.

In the last experiment, we assume robots cannot move 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 "placing 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 planning 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 generate 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 results of this experiment. 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 more optimal place positions.

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

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency a forward chaining approach. In *Proceedings of the 17th International Joint Conference on Artificial intelligence*, 417–424. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Boutilier, C., and Brafman, R. I. 2001. Partial-order planning with concurrent interacting actions. *Journal Of Artificial Intelligence Research* 14:105–136.

Chen, J., and Sun, D. 2010. An online coalition based approach to solving resource constrained multirobot task allocation problem. In *2010 IEEE International Conference on Robotics and Biomimetics*, 92–97.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence* 173(1):1–44.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2012. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research* 44(1):1–96.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, 1852–1859. Morgan Kaufmann Publishers Inc.

Di Rocco, M.; Pecora, F.; Sivakumar, P. K.; and Saffiotti, A. 2013. Configuration planning with multiple dynamic goals. In *2013 AAAI Spring Symposium Series*.

Do, M., and Kambhampati, S. 2003. SAPA: a multi-objective metric temporal planner. *Journal Of Artificial Intelligence Research* 20:155–194.

Dornhege, C.; Eyerich, P.; Keller, T.; Trg, S.; Brenner, M.; and Nebel, B. 2012. Semantic Attachments for Domain-Independent Planning Systems. In *Towards Service Robots for Everyday Environments*, number 76 in Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg. 99–115.

Erol, K.; Nau, D. S.; and Subrahmanian, V. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(12):75–88.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, 130–137. AAAI Press.

Fox, M., and Long, D. 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fua, C., and Ge, S. 2005. COBOS: Cooperative backoff adaptive scheme for multirobot task allocation. *IEEE Transactions on Robotics* 21(6):1168–1178.

Gerevini, A., and Serina, I. 2002. LPG: A Planner Based on Local Search for Planning Graphs with Action Costs. In *Proceedings of 6th International Conference on AI Planning Systems*, 13–22. AAAI Press.

He, L., and Ioerger, T. R. 2003. A quantitative model of capabilities in Multi-Agent systems. In *Proceedings of International Conference on Artificial Intelligence*, 730–736.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14(1):253–302.

Hoffmann, J. 2001. The FF domain collection, http://fai.cs.uni-saarland.de/hoffmann/ff-domains.html.

Huckaby, J., and Christensen, H. 2012. A Taxonomic Framework for Task Modeling and Knowledge Transfer in Manufacturing Robotics. In *Proceedings of the 8th International Workshop on Cognitive Robotics*, 94–101.

Keller, T.; Eyerich, P.; and Nebel, B. 2010. Task planning for an autonomous service robot. In *KI 2010: Advances in Artificial Intelligence*, number 6359 in Lecture Notes in Computer Science. Springer Berlin Heidelberg. 358–365.

Korsah, G. A.; Stentz, A.; and Dias, M. B. 2013. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research* 32(12):1495–1512.

Kovacs, D. 2012. A multi-agent extension of PDDL3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition*.

Kunze, L.; Roehm, T.; and Beetz, M. 2011. Towards semantic robot description languages. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, 5589–5595. IEEE.

Malik Ghallab, D., and Paolo Traverso. 2004. *Automated planning: Theory and practice*. San Francisco, CA: Morgan Kaufmann Elsevier.

McDermott, D. 1978. Planning and acting. *Cognitive Science* 2(2):71–109.

McDermott, D. 1999. Using regression-match graphs to control search in planning. *Artificial Intelligence* 109(12):111–159.

McDermott, D. V. 2003. Reasoning about Autonomous Processes in an Estimated-Regression Planner. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, 143–152.

Parker, L., and Tang, F. 2006. Building multirobot coalitions through automated task solution synthesis. *Proceedings of the IEEE* 94:1289 – 1305.

Sutton, D.; Klein, P.; Otte, M.; and Correll, N. 2010. Object Interaction Language (OIL): An intent-based language for programming self-organized sensor/actuator networks. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 6113–6118.

Wurm, K. M.; Dornhege, C.; Nebel, B.; Burgard, W.; and Stachniss, C. 2013. Coordinating heterogeneous teams of robots using temporal symbolic planning. *Autonomous Robots* 34(4):277–294.

Zuech, N., and Miller, R. K. 1989. *Machine Vision*. Springer.