A formalization of multi-agent planning with explicit agent representation

Alessandro Trapasso
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome, Italy
trapasso@diag.uniroma1.it

Sofia Santilli
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome, Italy
sofiasantilli1998@gmail.com

Luca Iocchi
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome, Italy
iocchi@diag.uniroma1.it

Fabio Patrizi
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome, Italy
patrizi@diag.uniroma1.it

ABSTRACT

We present a formalization of multi-agent planning problems in which agents are explicitly represented. In contrast with previous formalisations, we discuss the advantages of representing agents explicitly and show the implementation in the Unified Planning formalism and some practical examples. The proposed formalism is equivalent to other models, in particular to Multi Agent Planning Domain Definition Language (MA-PDDL), and can thus be compiled into it and solved by existing multi-agent planning solvers. Moreover, we present a further extension to define more complex multi-agent problems with explicit models of other agents.

KEYWORDS

Multi-agent planning, MA-PDDL

ACM Reference Format:
Alessandro Trapasso, Sofia Santilli, Luca Iocchi, and Fabio Patrizi. 2023. A formalization of multi-agent planning, with explicit agent representation. In *Proceedings of ACM SAC Conference (SAC'23).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3555776.3577753

1 INTRODUCTION

Multi-Agent Planning (MAP) is the planning problem in a domain where many autonomous agents coexist and act. Two main variants of MAP can be identified: one where the agents cooperate towards a common goal while possibly maintaining privacy about certain information, and another where each agent acts towards its own (private) goal.

In order to automatically solve MAP problems, a specification language must be defined, which can be taken as input by a solver. Obviously, the characteristics such a language needs to feature depend on the variant of the problem under investigation. In this paper, we target the established language Multi-agent Planning Domain Definition Language (MA-PDDL) [6], which allows for modelling both the cooperative and the competitive variants.

Another important distinction in MAP concerns the solution approach. There are essentially two approach families: *centralized* and *distributed*. In the former, a central planner takes care of synthesizing a single plan which assigns each action to some agents, in such a way that the execution of the plan allows for reaching the desired goal(s). With this approach, in order to carry out the task, the planning engine needs to access the information related to every agent. Thus, while privacy can be enforced among agents, private information needs to be shared with the planner. In the latter approach, agents plan separately, with the aim to either contribute to the achievement of the common goal (in the collaborative case) or to reach their private goals (in the competitive case). Here, we deal with the former variant, where agents act collaboratively.

While MA-PDDL is an established formalism to model MAP, we identified a drawback in the choice of modelling agents as objects that may introduce semantic confusion and could prevent extensions to more complex specifications.

This paper presents a formal model for MAP that makes the agent concept explicit and clearly distinguished from objects. Our formal model is equivalent to other models (in particular, to MA-PDDL) and can thus be compiled in domain and problem specifications used by existing MAP solvers. However, the advantage of using a formal model with explicit representation of agents is in the capability of defining more complex MA problems with explicit models of other agents.

2 RELATED WORK

In recent years there has been an increase in research activity by the planning community in MA cooperative planning. The International Planning Competition (ICP) is one of the most important competitions involving the world of planning and has contributed to making significant progress in the world of planning. The first edition took place in 1998, in which Planning Domain Definition Language (PDDL) [8] became a standard de facto language for single-agent planning. Subsequent editions of the ICP have continued to improve the language by introducing new extensions. Until a few years ago, there was still no de facto standard for deterministic MAP. More recently, in the 2015 CoDMAP competition [3], MA-PDDL [6] has become a standard de facto language for MAP.

SAC'23, March 27 –March 31, 2023, Tallinn, Estonia
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577753

All the planners participating in the competition used MA-PDDL as the input language. Here we use one of such planners, namely FMAP [11], to solve the MA-PDDL problems generated by our system. MA-PDDL is a MA extension of PDDL3.1 [5], can be used for modelling MAP activity and allows the definition of both factored and unfactored. In the factored case, each planning agent uses its planning sub-task, i.e., for each agent, we provide two MA-PDDL specifications: one with the domain description and one with the problem description. In the unfactored case, we have a single specification that defines the domain of all agents and another single specification that defines the problem of all agents. Compared to the classic PDDL, there are only two additional aspects: (i) the agents take part in the planning activity, (ii) predicates, objects and implicitly actions can be specified as private. Other characteristics of MA-PDDL remained unchanged with respect to PDDL or have been adapted to represent MA domains. The agents in MA-PDDL are objects (or constants) to which it is possible to associate actions, objectives and (definitions of metrics, and utilities). More specifically, in MA-PDDL, we can define agent types as objects and associate such types with actions. Agent instances are constant of an agent type. The association of actions to agent types is unambiguous as objects can only have an associated action schema with the same name and arity. The `:private` block is present in both agent domain predicates and agent problem objects and indicates which predicates and objects are hidden from other agents. In this way, only the agent knows its own private predicates and objects.

Although MA-PDDL contains all the necessary features to model MAP problems, the design choice of considering agent types as objects allows for defining domains with unclear semantics. For example, from the definition of the domain types, it is not possible to distinguish agent types from other objects. When hierarchical types are used, we can have predicates admitting either an agent or an object. For example, in the famous MAP domain `depot`, an agent type is named `place` (often confused with the place in which the agent is), and predicate `at` is used to denote both the position of an agent and the position of other objects.

In addition to possible semantic confusion, considering agents as objects can also bring some inconvenience for plan execution when binding PDDL symbols to action and predicate implementations. For example, in a mobile robotic application, a predicate `at` referring to the position of the robot in an environment will likely be implemented by a localization module, while a predicate `at` referring to the position of an object in the environment could be the result of processing images captured with a camera. As these predicates have completely different meanings and implementations, it would be desirable to capture the difference in the planning domain, in such a way that, at execution time, it becomes clear which implementation each predicate refers to.

Finally, MA-PDDL does not allow to model mental states of agents and knowledge of one agent about other agents. For this kind of extensions, explicit models of agents are necessary.

This work aims at highlighting the difference between agents, which actively operate in the environment, and objects, which are passive entities. Agents are different from objects and also from the conceptual and knowledge-representation perspectives. This distinction between agents and objects becomes even more marked when considering human-aware planning settings, where humans, who are definitely not objects, acting in the environments are modelled as agents with significantly different characteristics wrt those of other artificial agents.

In this paper, we present a novel formalization of MAP with explicit agent representation, which is equivalent to MA-PDDL but allows for clear semantics of the specification of the planning domains and problems, reducing possible causes of semantic confusion. Such an equivalent formalization allows for compiling MAP problems into MA-PDDL and for using available MAP solvers.

We also discuss some extensions of this formalization to take into account agent individual abilities or constraints, mental states, and knowledge about other agents.

3 MULTI-AGENT MODELLING

In this section, we present the MAP framework with explicit agent modelling. We stress that the framework is not meant as a new MAP language but, rather, as a conceptual model to be possibly compiled into MA-PDDL (or any other expressive-enough language).

3.1 Formalism and problem specification

A *planning domain* is, as standard, a pair $\mathcal{D} = \langle F, A \rangle$, where F is a set of propositional fluents and A a set of actions. A *planning problem* is a tuple $\Pi = \langle \mathcal{D}, I, G \rangle$, where $\mathcal{D}$ is a planning domain, and I and G, respectively, the initial state and goal description, expressed as boolean formulas over fluents from F.

We assume an *environment* $\mathcal{E}$ modelling the common environment the agents act in and retaining the information available to all agents. Such information is formally captured by a set of *environment fluents* $\mathcal{E}.F$. The set of agents acting in $\mathcal{E}$ is $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$. Each agent $\lambda_i \in \Lambda$ has an associated set $\lambda_i.F$ of *agent(-specific) fluents*, which model the state of λ_i, and a set of *agent(-specific) actions* $\lambda_i.A$, representing all the actions available to the agent. Agent-specific fluents can be shared among other agents in the team if they are marked as *public*. Conversely, *private* fluents are not available to other agents. We denote with $\lambda_i.P \subseteq \lambda_i.F$ the subset of public fluents of agent λ_i and with $\Lambda_{-i}.P = \bigcup_{j=1,\ldots,n, j \neq i} \lambda_i.P$, the set of all public fluents of all other agents except λ_i.

Every agent has a different perspective on world dynamics, which depends on its own fluents and actions, on the environment fluents, and on the public fluents of other agents. This is captured by the notion of *agent domain*, i.e., a pair $\mathcal{D}_i = \langle F_i, A_i \rangle$, where $F_i = \mathcal{E}.F \cup \lambda_i.F \cup \Lambda_{-i}.P$ and $A_i = \lambda_i.A$. On top of this, every agent can define an *agent problem*, which is a tuple $\Pi_i = \langle \mathcal{D}_i, I_i, G_i \rangle$, where I_i and G_i model the initial state and the goal for the agent, and are expressed as formulas over the fluents F_i of $\mathcal{D}_i$.

From the set of domains $\mathcal{D}_i = \langle F_i, A_i \rangle$ of all agents, a *multi-agent planning domain*, or *MA planning domain*, is obtained: $\mathcal{D}_{MA} = \langle F_{MA}, A_{MA} \rangle$, where $F_{MA} = \bigcup_{i=1}^n F_i$ and $A_{MA} = \bigcup_{i=1}^n A_i$. From a collection of agent problems $\Pi_i = \langle \mathcal{D}_i, I_i, G_i \rangle$, we obtain a *multi-agent planning problem* (*MA planning problem*) $\Pi_{MA} = \langle \mathcal{D}_{MA}, I_{MA}, G_{MA} \rangle$, where $\mathcal{D}_{MA}$ is the MA planning domain obtained from all agent domains $\mathcal{D}_i$ and $I_{MA} = \bigwedge_{i=1}^n I_i$ and $G_{MA} = \bigwedge_{i=1}^n G_i$. Essentially, Π_{MA} is the problem obtained by joining the problems of all the agents. Notice that since the agents act in the same environment, by the definitions of agent planning domain and MA planning domain, it follows that $\mathcal{D}_{MA}$ includes all the environment fluents $\mathcal{E}.F$, in addition to all agents'.

Intuitively, each $f \in F_{MA}$ is a propositional fluent whose value depends on the current state of the modelled world. Environment fluents are associated to environment's and not agents' properties, while agent-specific fluents are associated to individual agents and, while possibly named in the same way, may take different values for different agents. Agent-specific fluents are not significant when not associated to any agent.

For instance, a fluent *door_open* $\in \mathcal{E}.F$ might be used to model whether a door is open (fluent true) or closed (false); its current value does not depend on the state of the agents (although it might be affected by the actions they execute). Such fluent, as well as its value, is accessible to all agents, which thus share the knowledge about the state of the door. As to agent-specific fluents, consider two different agents λ_i and λ_j and assume both have a fluent, say *at*, modelling their current position. Formally, the agents are associated to fluents $\lambda_i.at$ and $\lambda_j.at$, which may take different values in some configurations of the world, depending on the position of the agents in the configuration. If predicate *at* is not associated to any agent, it is not significant.

The agents in Λ are assumed to be the only actors in the world; in particular, the environment cannot perform any action. This is modelled by associating no action to $\mathcal{E}$. When some agent λ_i performs an action $a \in \lambda_i.A$, it affects the state of the world by changing the values of the fluents describing the domain.

As typical in planning, actions and fluents can be parameterized with a set of objects O. For instance, to express that agent λ holds a box, we can use the expression: $\lambda.holds(box)$. Since we assume that the number of objects is finite, i.e., there exist only finitely many valuations for *box*, this parameterization is just a syntactic shortcut to define one distinct fluent for every parameter assignment.

The MA initial state I_{MA} and the goal description G_{MA} are terms based on environmental fluents and on agent-specific fluents. The output plan will be given in terms of agent-specific actions. The solution of a MA problem is a MA plan represented as a combination of agent-specific actions that, when executed (according to a specific execution model) from a state represented by I_{MA}, reaches a state represented by the goal description G_{MA}.

In general, for the same problem, there may exist many solution plans, possibly obtained by serializing in different ways sets of actions which do not affect executability of each other (but, together, may affect the executability of other actions needed to reach the goal). If, for instance, the goal requires to have a door and a window open, the order in which actions *open_door* and *open_window* are executed is not important. In these situations, instead of providing a single, rigid, solution, it is convenient to return a compact representation of a set of plans. To this end, a Partially Order Plan (POP, [13]) can be used, which is essentially a Directed Acyclic Graph representing the (partial) ordering constraints over the actions to be executed [1]. In this paper, we adopt this representation. It is immediate to see that sequential plans are special cases of POP.

[1]In fact, POPs also contain causal links, not relevant here.

3.2 Compilation to MA-PDDL

In our formalism, agents are represented explicitly. The formalism is implemented in the Unified Planning (UP) Framework for the AIPlan4EU project[2]. The main objective of the project is to make planning technology accessible to practitioners, companies, PMI and innovators, to facilitate the use of the planning technology in real scenarios. The UP library allows for creating a single-agent problem and extend it to a MA one by simply defining the environment and the agents, and associating fluents and actions to the agents. Algorithms 1 and 2 allow for compiling a MA problem Π_{MA} into a MA-PDDL specification. We next describe the algorithms.

MA-PDDL allows for specifying problems and domains in either *factored* or *unfactored* way. The latter requires writing a pair of files corresponding to the problem and the domain for each agent, while the latter requires only a single domain file and a single problem file. Our algorithms use the factored representation.

The algorithms take as input a MA problem represented in the UP formalism and transform the agents into objects, as required by the MA-PDDL formalism. Agent-specific actions are compiled into MA-PDDL actions with the agent type set as the action parameter. Agent-specific fluents are compiled into private predicates that have as their parameter the type of the agent. Environment fluents are compiled into domain predicates or functions.

Algorithm 1 implements the planning domain compilation process. For simplicity, we consider only boolean predicates and instantaneous actions, but generalizing the process to functions and other actions classes (e.g., durative or sensing actions) is straightforward. The main steps of the algorithm are the following:

- U is the set of MA problem's user types, i.e., the user-defined types of objects. User types are organized hierarchically, as there may exist subtypes; types without a parent are simply compiled as *objects* (lines 11 - 15).
- Environment fluents are compiled as predicates; agent-specific fluents are compiled into predicates, according to their privacy flag (lines 16 - 30).
- Each agent-specific action is compiled into an action. Function *agent_spec* (not reported here) produces, for an agent-specific fluent, the corresponding MA-PDDL expression specifying the involved types; for instance, fluent $at(?loc)$ is compiled as $(at\ ?agent\ ?loc)$ (lines 31 - 43).

Algorithm 2 implements a planning problem's compilation.

- O is the set of objects in the MA problem. Domain objects have two parameters, name and type, while agent objects also have a third parameter, the agent owning the object. The algorithm outputs a list of such objects, together with their parameters. Agent objects are specified using keyword ":private" (lines 8 - 13).
- I is the set of fluents' initial values. For an initial value $i \in I$, by $i.f$ we denote the fluent f that the initial value refers to. Initial values can be *agent-specific*, meaning that the fluent they refer to is so. For this class of initial values, the algorithm produce an MA-PDDL specification, including the fluent, the agent and the list of involved objects. For non-agent-specific initial values, instead, the agent is omitted (lines 14 - 19).

[2]https://www.aiplan4eu-project.eu/

- Similarly to the initial values, also goals can be agent-specific. Agent-specific goals are identified with the keyword *:agent* and produce an output similar to that of initial values. Function *agent_spec* (not reported) extract the goals from the goal formula G. (line 20).

Algorithm 1 MA-PDDL domain compilation for agent (λ_i)

```
1:  Environment: $\mathcal{E}$
2:  Agent-specific fluents: $\lambda_i.F$
3:  Subset of agent-specific public fluents: $\lambda_i.P$
4:  Other agents public fluents: $\Lambda_{-i}.P$
5:  Agent-specific actions: $\lambda_i.A$
6:  $\Pi_{MA}$: MA planning problem
7:  $U$: set of UserType in $\Pi_{MA}$
8:
9:  write: "(define ("$\Pi_{MA}$.name") (:requirements$" $\Pi_{MA}$.req ")"
10: write: "(:types"
11: for all Type $t \in U$, SubType $s \in U$ do
12:     write: $t$ "- object"
13:     write: $s$ "-" $s.type$
14: end for
15: write: ")"
16: write: "(:function"
17: for all IntType \or RealType $f \in \mathcal{E}.F \cup \Lambda_{-i}.P$ do
18:     if $f \in \Lambda_{-i}.P$ then write: "(" $f.name$ "- ?agent -" $\lambda_i.object.type$ "-" $f.type.name$ ")";
19:     elif $f \in \mathcal{E}.F$ then write : "(" $f.name$ "-" $f.type.name$ ")";
20: end for
21: write: "(:predicate"
22: for all BoolType $f \in \mathcal{E}.F \cup \Lambda_{-i}.P$ do
23:     if $f \in \Lambda_{-i}.P$ then write: "(" $f.name$ "- ?agent -" $\lambda_i.object.type$ "-" $f.type.name$ ")";
24:     elif $f \in \mathcal{E}.F$ then write : "(" $f.name$ "-" $f.type.name$ ")";
25: end for
26: write: "(:private"
27: for $f$ in $\lambda_i.F$ do
28:     write: "(" $f.name$ "- ?agent -" $\lambda_i.object.type$ "-" $f.type.name$ ")"
29: end for
30: write: ")))"
31: for all $a$ in $\lambda_i.A$ do
32:     write: "(:action" $a.name$
33:     for all $ap$ in $a.parameters$ do
34:         write: ":parameters ( ?a - agent" $a.type.name$ $ap.name$ "-" $ap.type.name$ ")"
35:     end for
36:     for $pr$ in $a.precondition$ do
37:         write: ":precondition ("$agent\_spec(pr)$ ")"
38:     end for
39:     for $e$ in $a.effect$ do
40:         write: ":effect ("$agent\_spec(e)$ ")"
41:     end for
42: end for
43: write: "))"
```

Algorithm 2 MA-PDDL problem compilation for agent (λ_i)

```
1:  $U$: set of UserType in $\Pi_{MA}$
2:  $O$: set of Objects in $\Pi_{MA}$
3:  $I$: set of initial values in $\Pi_{MA}$
4:  $G$: goal formula of $\Pi_{MA}$
5:
6:  write: "(define (problem" $\Pi_{MA}.name$ "- problem)")
7:  write: "(:domain $\Pi_{MA}.name$ "- domain)"
8:  write: "(:objects"
9:  for $o \in O$ do
10:     if $o \in \lambda_i.O$ then write: "(:private" $\lambda_i.name$ $o.name$ "-" $o.type.name$ ")";
11:     elif $o \in O \setminus \lambda_i.O$ then write: $o.name$ "-" $o.type.name$;
12: end for
13: write: ")"
14: write: "(:init "
15: for $i \in I$ do
16:     if $i.f \in \lambda_i.F$ then write: "(" $i.f.name$ $i.\lambda_i.name$ $i.objects$ ")";
17:     elif $i.f \notin \lambda_i.F$ then write: "(" $i.f.name$ $i.objects$ ")";
18: end for
19: write: ")"
20: write: "(:goal " $agent\_spec(G)$ "))"
```

3.3 Examples

To illustrate the use of our MA planning formalism implemented in the Unified Planning (UP) framework, we present three examples: a simple domain, a benchmark used in IPC competitions, and the formalization of a real industrial problem.

Multi-robot loader. We first describe a simple single-agent problem *robot loader* and its extension to a MA problem *multi-robot loader*. The goal of the problem is to transport a cargo (whose position is denoted by the fluent *cargo_at*) from location *l2* to location *l1*. The concept of location is modelled by the *UserType Location*, *l1* and *l2* are two objects with this type. The fluents *robot_at* and *cargo_at* indicate the positions of the robot and of the cargo. The fluent *cargo_mounted* models the concept of having the cargo mounted on the robot and the fluent *is_connected* indicates that there is an edge between two locations. The actions are *move*, *load*, and *unload*, with preconditions and effects that model the dynamics of the system. Action *move* has two parameters of type *Location* indicating the current position of the robot l_from and the intended destination of the movement l_to, while actions *load* and *unload* have only one parameter of type *Location*. A possible sequential plan, generated for example, by classical planners like *fast-downward* [2], *TAMER* [12] or *Pyperplan* [1], concatenates *move*, *load*, and *unload* actions to achieve a final state in which the cargo is in the desired location, i.e., $cargo\_at(l1)$ holds.

A snippet of UP code is shown below

```
#Robot loader
Location = UserType("Location")
robot_at = Fluent("robot_at", BoolType(), position=Location)
cargo_at = Fluent("cargo_at", BoolType(), position=Location)
cargo_mounted = Fluent("cargo_mounted")
is_connected = Fluent("is_connected", BoolType(), l1=Location, l2=
       Location)
move = InstantaneousAction("move", l_from=Location, l_to=Location)
l_from = move.parameter("l_from")
l_to = move.parameter("l_to")
move.add_precondition(is_connected(l_from, l_to))
move.add_precondition(robot_at(l_from))
move.add_precondition(Not(robot_at(l_to)))
move.add_effect(robot_at(l_from), False)
move.add_effect(robot_at(l_to), True)
load = InstantaneousAction("load", loc=Location)
...
unload = InstantaneousAction("unload", loc=Location)
...
l1 = Object("l1", Location)
l2 = Object("l2", Location)
problem = Problem("robot_loader")
problem.add_fluent(robot_at)
...
problem.add_action(move)
...
problem.add_object(l1)
problem.add_object(l2)
problem.set_initial_value(robot_at(l1), True)
problem.set_initial_value(cargo_at(l2), True)
...
problem.add_goal(cargo_at(l1))
```

From the above single-agent specification, we can easily describe a MAP problem, by explicitly defining agents. In this example, we will show how to extend the domain with two robots. The *Multi-robot loader* problem will be specified by extending the *Robot loader* problem, reusing fluent, action and object specifications.

In the MAP problem, the agents are named $robot1$ and $robot2$. The fluents *is_connected* and *cargo_at* are environment fluents, while *robot_at* (renamed into *at*) and *cargo_amounted* are agent-specific fluents. The actions *move*, *load* and *unload* are also agent-specific. The association between agents and agent-specific fluents and actions is obtained in UP through methods *add_fluent* and *add_action* of the *Action* class. To specify the initial value of the planning problem, we use the operator *Dot*, which allows associating an agent-specific fluent with an agent. For example, *Dot(robot1,at(l1))* denotes fluent $at(l1)$ for agent $robot1$, i.e., $robot1.at(l1)$, while The goal of the MAP problem is still to have the cargo in the desired location. In this problem, three locations are considered and a possible sequential plan concatenates *move*, *load*, and *unload* actions of both agents. The UP code snippet for the MA extension of the problem is reported below.

```
# Multi-robot loader
problem = MultiAgentProblem("multi-robot_loader")
at = Fluent("at", Location)  # replaces robot_at
problem.ma_environment.add_fluent(is_connected)
problem.ma_environment.add_fluent(cargo_at)
# agent definition
robot1 = Agent("robot1", problem)
robot1.add_fluent(at)
robot1.add_fluent(cargo_mounted)
robot1.add_action(move)
robot1.add_action(load)
robot1.add_action(unload)
robot2 = Agent("robot2", problem)
...
problem.add_agent(robot1)
problem.add_agent(robot2)
# New objects
l3 = Object("l3", Location)
problem.add_objects([l1, l2, l3])
# Initial state
problem.set_initial_value(is_connected(l1, l2), True)
...
problem.set_initial_value(Dot(robot1, at(l1)), true)
problem.set_initial_value(Dot(robot2, at(l2)), true)
problem.set_initial_value(cargo_at(l3), True)
...
problem.set_initial_value(Dot(robot1, cargo_mounted), False)
problem.set_initial_value(Dot(robot2, cargo_mounted), False)
# Goal
problem.add_goal(cargo_at(l1))
```

Depot. Depot is a standard benchmark for MAP and MA-PDDL solvers. This domain is more complex than the previous one, as different types of agents work together to achieve the goal. This domain was designed for the *AIPS 2002 planning competition*[7] and combines logistics domains and blocks domains, well known in the literature. The Depot domain (rewritten in MA-PDDL) [4] is part of the 10 benchmarks used in the 2015 CoDMAP competition [3]. There are two types of agents, *driver* and *place*. The *driver* agents drive the *trucks* to transport the crates between the warehouse, with the help of hoists present in each warehouse. The agent's depots and distributors are place-type, as each place has control over a *hoist*.

The actions are:

- **drive**: move the truck from one place to another.
- **load**: load a crate on a truck via a hoist.
- **unload**: unloads a crate from a truck via a hoist.
- **lift**: lifts a crate placed on a pallet or other crate.
- **drop**: drop a crate onto a pallet or another crate.

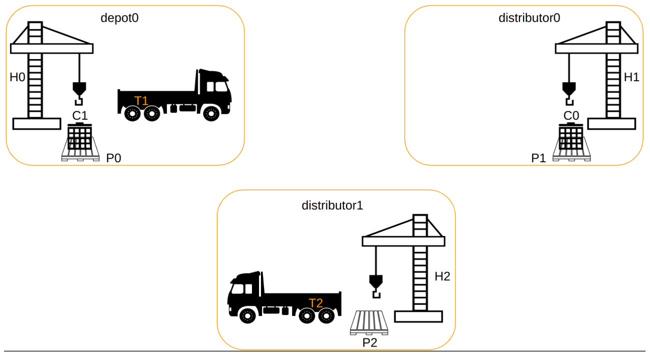

Figure 1: Initial state - Depot

In the *initial state*, *truck1* driven by *driver1*, is at *depot0* and *truck2* driven by *driver2*, is at *distributor1*. The driver driving the truck is free to move between all positions. The global goal is to have *crate0* on *pallet2* and *crate1* on *pallet1*, so we need to move *crate0* to *distributor1* and *crate1* to *distributor0*.

In our example, we changed the representation of the Depot domain by representing agents explicitly but keeping it equivalent to the representation in MA-PDDL. To make the agents explicit, we instantiated the agents: (*depot0*, *distributor0*, *distributor1*, *driver0*, and *driver1*, according to our formalism.

In MA-PDDL Depot representation, the agents are defined as object types associated with the action. As already mentioned, in our formalism, agents are not objects. Moreover, we use a more general notation in which each agent is named individually. For example, the *drive* action is parametric to both the explicit agents *driver0* and *driver1*.

Below is a snippet of the MA-PDDL specification of the Depot problem.

```
#MA-PDDL unfactored (depot problem)                            1
(:init                                                         2
  (driving driver0 truck0)                                     3
  (driving driver1 truck1)                                     4
  (at pallet0 depot0)                                          5
  (clear crate1)                                               6
  (at pallet1 distributor0)                                    7
  (clear crate0)                                               8
  (at pallet2 distributor1)                                    9
  (clear pallet2)                                              10
  (at truck0 distributor1)                                     11
  (at truck1 depot0)                                           12
  (at hoist0 depot0)                                           13
  (available depot0 hoist0)                                    14
  (at hoist1 distributor0)                                     15
  (available distributor0 hoist1)                              16
  (at hoist2 distributor1)                                     17
  (available distributor1 hoist2)                              18
  (at crate0 distributor0)                                     19
  (on crate0 pallet1)                                          20
  (at crate1 depot0)                                           21
  (on crate1 pallet0)                                          22
)                                                              23
```

In the initial state of Depot's problem, we can see that the objects representing agents also indicate their location. For example, in *(line 5 of MA-PDDL problem)*, *pallet0* has as initial position *depot0*, which is an object representing the location of *pallet0*, but also the agent executing actions. In our formalism, we avoid this kind of semantic confusion, since agents are entities different from objects and cannot be used in their place.

A snippet of UP code for the Depot problem is shown below. The agents are explicitly described and associated to actions and fluents, allowing for a description with clear semantics. In this case, agents and places are clearly separated and a `Dot` operator is used to denote agent-specific fluents. For example, the predicate representing the position of agent `depot0` at location `place0` is denoted with `Dot(depot0, pos(place0))`

```
#MultiAgent-Depot
place = UserType("place")
locatable = UserType("locatable")
...
at = Fluent("at", BoolType(), locatable=locatable, place=place)
pos = Fluent("pos", BoolType(), place=place)
...
drive = InstantaneousAction("drive", x=truck, y=place, z=place)
x = drive.parameter("x")
y = drive.parameter("y")
z = drive.parameter("z")
drive.add_precondition(at(x, y))
drive.add_precondition(driving(x))
drive.add_effect(at(x, z), True)
drive.add_effect(at(x, y), False)
lift = InstantaneousAction("lift", p=place, x=hoist, y=crate, z=
      surface)
...
drop = InstantaneousAction("drop", p=place, x=hoist, y=crate, z=
      surface)
...
load = InstantaneousAction("load", p=place, x=hoist, y=crate, z=
      truck)
...
unload = InstantaneousAction("unload", p=place, x=hoist, y=crate,
      z=truck)
...
problem = MultiAgentProblem("depot")
depot0 = Agent("depot0", problem)
distributor0 = Agent("distributor0", problem)
distributor1 = Agent("distributor1", problem)
driver0 = Agent("driver0", problem)
driver1 = Agent("driver1", problem)
driver0.add_action(drive)
driver1.add_action(drive)
depot0.add_action(lift)
...
distributor0.add_action(lift)
...
distributor1.add_action(lift)
...
problem.ma_environment.add_fluent(at, default_initial_value=False)
...
driver0.add_fluent(driving, default_initial_value=False)
depot0.add_fluent(pos)
driver1.add_fluent(driving, default_initial_value=False)
driver1.add_fluent(pos)
depot0.add_fluent(lifting, default_initial_value=False)
depot0.add_fluent(available, default_initial_value=False)
depot0.add_fluent(pos)
distributor0.add_fluent(lifting, default_initial_value=False)
distributor0.add_fluent(available, default_initial_value=False)
distributor0.add_fluent(pos)
distributor1.add_fluent(lifting, default_initial_value=False)
distributor1.add_fluent(available, default_initial_value=False)
distributor1.add_fluent(pos)
problem.add_agent(depot0)
...
truck0 = Object("truck0", truck)
place0 = Object("place0", place)
...
problem.add_object(truck0)
problem.add_object(place0)
...
problem.set_initial_value(Dot(depot0, pos(place0)), True)
problem.set_initial_value(at(truck1, place0), True)
...
problem.add_goal(on(crate0, pallet2))
problem.add_goal(on(crate1, pallet1))
```

As already mentioned, our formalism is equivalent to MA-PDDL and Algorithms 1 and 2 are used to generate MA-PDDL specifications and solve the problem with MA-PDDL solvers. An example of a solution provided by *FMAP*[10] planner (integrated into the UP library) is the Partial Order Plan (POP) shown in Figure 2.

```
with OneshotPlanner(problem_kind=problem.kind) as planner:
    result = planner.solve(problem)
    print("Adjacency list:", result.plan.get_adjacency_list)
```

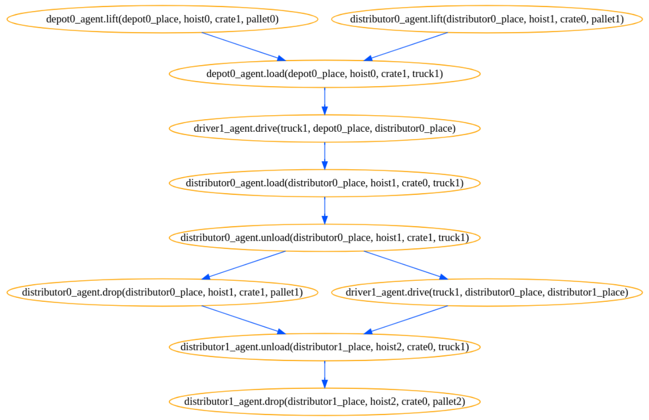
depot0_agent.lift(depot0_place, hoist0, crate1, pallet0)
distributor0_agent.lift(distributor0_place, hoist1, crate0, pallet1)
depot0_agent.load(depot0_place, hoist0, crate1, truck1)
driver1_agent.drive(truck1, depot0_place, distributor0_place)
distributor0_agent.load(distributor0_place, hoist1, crate0, truck1)
distributor0_agent.unload(distributor0_place, hoist1, crate1, truck1)
distributor0_agent.drop(distributor0_place, hoist1, crate1, pallet1)
driver1_agent.drive(truck1, distributor0_place, distributor1_place)
distributor1_agent.unload(distributor1_place, hoist2, crate0, truck1)
distributor1_agent.drop(distributor1_place, hoist2, crate0, pallet2)

Figure 2: Partial Order Plan - MA-Depot example

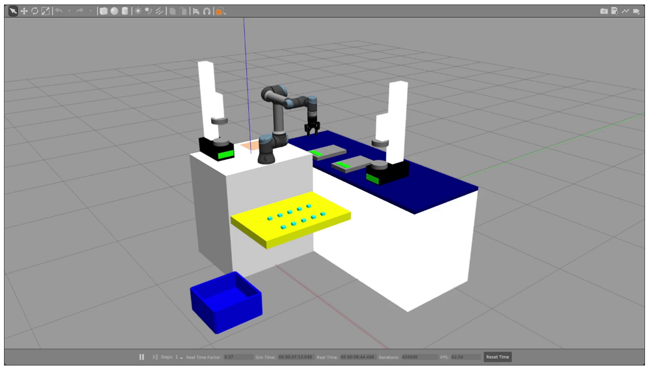

Figure 3: Simulated environment for robotic quality tests

Robotic automatic quality tests. The last example presented comes from a real industrial scenario. Here robots are employed to support people in the development of automated quality tests that, for statistical significance, need to be run on a high number of samples (thousands), since they are used in order to guide scientists in the selection and evolution of new products. In particular, we considered quality tests for laundry pouches operated by a robotic arm. In addition to the robotic arm, the scenario contains some measurement devices to be used to take quality measures about the pouches. The system's goal is to perform all the measures about all the pouches that are available in a cabinet drawer in the shortest time. A gazebo simulation environment has been setup to speed up development and testing (see Figure 3).

The formalization of this planning problem ended up in a domain composed by 8 actions, 14 predicates, and 28 objects, including locations, two states of the gripper (reset or active), four postures of the gripper (indicating how much the gripper is opened) and two modalities for grasping a pouch (horizontal or vertical).

Here some relevant aspects taken into account during the formalization are described. First, two different predicates were used to express the position of a pouch: when the pouch is resting in a drawer or on a measurement device or in the bin, the predicate '*pouchRestIn ?loc*' is used; when the pouch is in a location while carried by the gripper, '*pouchAt ?loc*' is employed. A different predicate '*at ?loc*' expresses the location occupied by the gripper. Afterwards, the motion actions of the gripper from one location to another were distinguished into three types, since they had different preconditions and effects: '*goto_grasp*' when the opened gripper has to go in a suitable position for grasping a pouch resting in a location, '*pickup*' when the gripper, already closed on a pouch, has to lift it and '*goto*' for the remaining displacements. Still, concerning these motion actions, some predicates were introduced, in order to classify the locations and allow the gripper to appropriately move between them, avoiding collisions. An example is '*dropPos ?x ?y*', where '*x*' is the location the gripper needs to be at when it wants to drop the pouch on a device '*y*'. Furthermore, particular attention was given to the grasping movement and three predicates were introduced for managing it. Once the gripper has reached the proper grasping position, it changes its posture, by closing itself on the pouch, making the predicate '*touched*' true. The lifting action follows, making '*picked*' true. Finally, a sensing action checks if the grasping was successful and if so, makes the '*grasped*' predicate true. It is also relevant to pay attention if the grasping happened horizontally or vertically with respect to the pouch. This is done through the predicate '*graspMode ?modality*', which constitutes a precondition for some motion actions. In fact, some devices require the grasp to have been previously performed horizontally to avoid the pouch's misplacing when the gripper puts it on the device.

There are two possible ways of modelling the domain: it can be treated as a single or MA problem. In the former, the planner will return a solution represented as a sequence of actions: a sequence of robot pick-and-place and measurement actions that reaches a goal state in which all the pouches have been correctly tested. A UP code snippet is presented below. The sequential plan generated by a classical planner contains a sequence of 21 actions to process each pouch. This sequence contains two measuring actions and actions for moving the robot arm and the gripper.

```
Location = UserType("Location")
pouchIn = Fluent("pouchIn", BoolType(), device=Location)
at = Fluent("at", BoolType(), position=Location)
restLoc = Fluent("restLoc", BoolType(), position=Location, device=
      Location)
reset = Fluent("reset", BoolType(), device=Location)
measuredAt = Fluent("measuredAt", BoolType(), device=Location)
measure=InstantaneousAction("measure", device=Location, rest=
      Location)
device = move.parameter("device")
rest = move.parameter("rest")
measure.add_precondition(pouchIn(device))
measure.add_precondition(at(rest))
measure.add_precondition(restLoc(rest, device))
measure.add_precondition(reset(device))
move.add_effect(measuredAt(device), True)
move.add_effect(reset(device), False)
...
goto=InstantaneousAction("goto", from=Location, to=Location)
...
movegripper_grasp=InstantaneousAction("movegripper_grasp", from=
      Location, to=Location)
...
loc1 = Object("loc1", Location)
loc2 = Object("loc2", Location)
device1 = Object("device1", Location)
problem = Problem("quality_test")
problem.add_fluent(pouchIn)
...
problem.add_action(measure)
...
problem.add_object(loc1)
...
problem.set_initial_value(at(loc1), True)
problem.set_initial_value(restLoc(loc2, device1), True)
...
problem.add_goal(measuredAt(device1))
```

However, this approach does not guarantee an optimal throughput, since sensors usually take time to perform measurements during which the robot remains idle, while it could operate on other pouches. A possible solution, with little effort in coding, is to treat the problem as a MA one, not considering only the robot as an agent, but also the measurement devices. It is sufficient to attribute the measurement actions to the corresponding devices and all the other actions to the robotic agent and to distribute the predicates between the agents' predicates and the local ones in the most appropriate way. The UP code for the MA extension of the problem is illustrated below.

```
problem = MultiAgentProblem("multi-agent_quality_test")
...
problem.ma_environment.add_fluent(pouchIn)
problem.ma_environment.add_fluent(restLoc)
problem.ma_environment.add_fluent(measuredAt)
...
robot = Agent("robot", problem)
robot.add_fluent(at)
robot.add_action(goto)
robot.add_action(movegripper_grasp)
problem.add_agent(robot)
device1 = Agent("device1", problem)
device1.add_fluent(reset)
device1.add_action(measure)
problem.add_agent(device1)
...
problem.set_initial_value(Dot(device1, reset), True)
problem.set_initial_value(Dot(robot, at(loc1)), True)
problem.set_initial_value(restLoc(loc2, device1loc), True)
...
problem.add_goal(measuredAt(device1))
```

By using a MAP planner, e.g., FMAP, it is possible to obtain a Partial Order Plan, in which robot actions and measurement operations are parallelized in order to increase the overall throughput. In particular, the measurement actions are executed in parallel with actions to move the robot arm to the position for checking and grasping the pouch after the measurement. This allows for reducing the overall execution of the plan to 19 steps instead of 21, thus also improving the system's overall throughput (processed pouches per hour).

To further improve performance, we are developing solutions allowing for higher action parallelism and introducing other robotic arms (other agents) that cooperate to achieve the desired goal.

Plan generation. In all the examples presented above, we used the UP libraries and planning engines to generate plans. This operation can be conveniently performed thanks to the ability to automatically select the most suitable planner given the features of the planning problem. In fact, when defining a UP planning domain, the structures used in the preconditions and effects formulas are analysed in order to determine the problem type. With this information, it is possible to generate plans in a unified way (see the snippet code below), with the automatic selection of a planning engine from the problem type.

```
with OneshotPlanner(problem_kind=problemP&G.kind) as planner:
    result = planner.solve(problem)
    if result.status in unified_planning.engines.results.
      POSITIVE_OUTCOMES:
        print(f"{planner.name} found this plan: {result.plan}")
    else:
        print("No plan found.")
```

As shown in the above examples, the UP formalism allows for improving the usability and effectiveness of AI planning technology. In particular, the MA planning formalization presented in this paper and its implementation in the UP allows to design, implement and test incremental planning-based solutions.

4 MODELLING OTHER AGENTS

Modelling other agents is an important aspect for MA planning and reasoning that allows for more sophisticated reasoning abilities based not only on facts about the environment, but also on other agents' mental states and knowledge about them. In this case, the model of agent λ_i can contain an estimation, the belief or the knowledge about the model of another agent λ_j. In other words, by modelling other agents, we can distinguish between what λ_j knows or believes about the world from what λ_i knows or believes about what λ_j knows or believes about the world. For example, we can model that $\lambda_j.\phi$ is *true* in λ_j model (i.e., λ_j knows or believes that ϕ is *true*), while $\lambda_j.\phi$ is unknown in λ_i model (i.e., λ_i does not know or does not believe that λ_j knows or believes that ϕ is *true*. When modelling other agents, it is not possible to assume perfect knowledge about such models and we thus need to consider models of other agents as estimates or approximations.

Our MA formalization described in the previous section can be easily extended to represent models of other agents, as described here. The domain specification for agent λ_i that includes an estimation of the model of agent λ_j can be defined as $\mathcal{D}_i^+ = \langle \mathcal{D}_i, \hat{\mathcal{D}}_{i,j} \rangle$, where $\hat{\mathcal{D}}_{i,j}$ is an estimation (approximation) of $\mathcal{D}_j$ from agent λ_i perspective (what λ_i knows or believes about λ_j). The corresponding agent problem can be defined as $\Pi_i^+ = \langle \Pi_i, \hat{\Pi}_{i,j} \rangle$, with $\hat{\Pi}_{i,j}$ being an approximation of Π_j from λ_i perspective. With this model, agent λ_i can also reason on the (estimated) model of agent λ_j and it is possible to define novel interesting reasoning and planning problems, some of which are described here for example.

1) Planning problems considering the goals of multiple agents: a) find a plan π_i for agent λ_i (i.e., solve Π_i to achieve G_i) that does not prevent another agent λ_j to achieve its own goal $\hat{G}_j$, i.e., it exists a plan π_j such that the execution of both π_i and π_j will reach both G_i and $\hat{G}_j$; b) find a plan π_i for agent λ_i that achieves $G_i \wedge \hat{G}_j$, i.e. that achieves also λ_j goal.

2) Planning considering actions of multiple agents: a) find a joint plan π with actions in $A_i \cup \hat{A}_j$ to achieve a common goal $G_i \wedge \hat{G}_j$.

3) Plan explanations as model reconciliation [9]: a) given π_i that is not valid or optimal in $\hat{\mathcal{D}}_{i,j}$, find an update of $\hat{\mathcal{D}}_{i,j}$ that will make the plan valid or optimal.

Defining and solving such planning problems will significantly increase the reasoning and planning capabilities of MA systems and will enable more interesting problem specifications considering teams formed by humans and artificial agents, allowing for a formal specification of human-aware reasoning and planning problems.

5 CONCLUSION

In this paper, we have presented a formalization of MAP where agents are explicitly modelled and distinguished from objects (not having an active role in the domain). We described the formalism and its compilation into the well-known language MA-PDDL, thus allowing for using existing solvers. We discussed some examples where the proposed approach has been successfully used, including a relevant industrial use case. Finally, we presented an extension to model other agents that can introduce novel interesting MA problems. The formalism has been implemented in the Unified Planning formalism developed within the AIPlan4EU project and is thus available to researchers and practitioners to effectively model and solve MA problems, with an intuitive API and clear semantics.

ACKNOWLEDGMENTS

Work partly supported by projects: EU ICT-49 2021 AIPlan4EU (No. 101016442), ERC Advanced Grant WhiteMech (No. 834228), and EU ICT-48 2020 TAILOR (No. 952215).

REFERENCES

[1] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. 2020. Pyperplan. https://doi.org/10.5281/zenodo.3700819
[2] Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.
[3] Antonín Komenda, Michal Štolba, and Dániel Kovács. 2016. The International Competition of Distributed and Multiagent Planners (CoDMAP). *AI Magazine* 37 (10 2016), 109–115. https://doi.org/10.1609/aimag.v37i3.2658
[4] Antonín Komenda, Michal Štolba, Dániel Kovács, and Michal Pechoucek. 2015. *Proc. of the 3rd Workshop on Distributed and Multi-Agent Planning.* 94 pages.
[5] Dániel László Kovács. 2011. BNF definition of PDDL 3.1. Unpublished manuscript from the IPC-2011. https://helios.hud.ac.uk/scommv/IPC-14/repository/kovacs-pddl-3.1-2011.pdf
[6] Dániel László Kovács. 2012. A multi-agent extension of PDDL3.1. In *ICAPS-2012 Proc. of the 3rd Workshop on Distributed and Multi-Agent Planning.* 19–27.
[7] Derek Long and Maria Fox. 2003. The 3rd International Planning Competition: Results and Analysis. *J. Artif. Intell. Res. (JAIR)* 20 (12 2003), 1–59.
[8] Drew M. McDermott. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21, 2 (Jun. 2000), 35. https://doi.org/10.1609/aimag.v21i2.1506
[9] S. Sreedharan, T. Chakraborti, and S. Kambhampati. 2021. Foundations of explanations as model reconciliation. *Artificial Intelligence* 301 (2021).
[10] Alejandro Torreño, Óscar Sapena, and Eva Onaindia. 2018. FMAP: A Platform for the Development of Distributed Multi-Agent Planning Systems. *Know. Based Syst.* 145, C (apr 2018), 166–168. https://doi.org/10.1016/j.knosys.2018.01.013
[11] Alejandro Torreño, Óscar Sapena, and Eva Onaindia. 2018. FMAP: A platform for the development of distributed multi-agent planning systems. *Knowledge-Based Systems* 145 (2018), 166–168. https://doi.org/10.1016/j.knosys.2018.01.013
[12] Alessandro Valentini, Andrea Micheli, and Alessandro Cimatti. 2020. Temporal Planning with Intermediate Conditions and Effects. In *Proc. of the 34th Conference on Artificial Intelligence (AAAI '20).* AAAI Press, 9975–9982.
[13] Daniel S. Weld. 1994. An Introduction to Least Commitment Planning. *AI Mag.* 15, 4 (1994), 27–61. https://doi.org/10.1609/aimag.v15i4.1109