ECHO: A hierarchical combination of classical and multi-agent epistemic planning problems

DAVIDE SOLDÀ, *Institute of Logic and Computation, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Wien, Austria.*

FRANCESCO FABIANO, *Department of Mathematical Physical and Computer Sciences, University of Parma, Parco Area delle Scienze 53/A, I-43124 Parma, Italy.*

AGOSTINO DOVIER, *Department of Mathematics, Computer Science and Physics, University of Udine, Via delle Scienze 206, I-33100 Udine, Italy.*

Abstract

The continuous interest in Artificial Intelligence (AI) has brought, among other things, the development of several scenarios where multiple artificial entities interact with each other. As for all the other autonomous settings, these *multi-agent* systems require orchestration. This is, generally, achieved through techniques derived from the vast field of *Automated Planning*. Notably, arbitration in multi-agent domains is not only tasked with regulating how the agents act, but must also consider the interactions between the agents' information flows and must, therefore, reason on an *epistemic* level. This brings a substantial overhead that often diminishes the reasoning process's usability in real-world situations. To address this problem, we present ECHO, a *hierarchical* framework that embeds *classical* and *multi-agent epistemic* (epistemic, for brevity) planners in a single architecture. The idea is to combine (i) classical; and(ii) epistemic solvers to model efficiently the agents' interactions with the (i) 'physical world'; and(ii) information flows, respectively. In particular, the presented architecture starts by planning on the 'epistemic level', with a high level of abstraction, focusing only on the information flows. Then it refines the planning process, due to the classical planner, to fully characterize the interactions with the 'physical' world. To further optimize the solving process, we introduced the concept of *macros* in epistemic planning and enriched the 'classical' part of the domain with *goal-networks*.

Finally, we evaluated our approach in an actual robotic environment showing that our architecture indeed reduces the overall computational time.

Keywords: Hierarchical Planning, Multi-Agent Epistemic Planning, Answer Set Programming, Answer Set Planning

1 Introduction

In recent years, the field of cognitive robotics experienced significant progress, both from the academic (see [7] for a detailed survey) and the industrial point of view (e.g. [11, 36, 50]). While most of this attention is directed to frameworks designed for scenarios where only a single entity interacts with the environment, multi-agent settings are gaining more and more attention [33, 43]. However, most of the tools envisioned for multi-agent systems do not focus on modelling the agents' information flows, i.e. they do not consider the *epistemic* aspects of the autonomous reasoning

Vol. 33, No. 8, © The Author(s) 2023. Published by Oxford University Press.

Advance Access Publication on 21 July 2023 https://doi.org/10.1093/logcom/exad036

process. In fact, the intrinsic difficulty of dealing with these concepts [10] often brings a far too great overhead so that 'real-world' tools could actually exploit them. Nonetheless, as hinted in [5], considering the knowledge/beliefs of the entities involved in a multi-agent scenario should be the natural option as it is the most accurate representation of the interactions between independent agents. This is why, starting from a real-world setting with two *Franka Emika manipulators* as agents, we decided to model a framework that could arbitrate them while considering the epistemic aspects of the domain and maintaining acceptable performances. After the first iteration, we then generalized the architecture to be able to deal with different scenarios that can be formally described with specific action languages.

In particular, we envisioned and developed ECHO (**E**pistemi**C** **H**ierarchical s**O**lver): a hierarchical planning framework that coordinates a set of autonomous agents. This set-up is able to reason on complex multi-agent epistemic concepts while dealing with the computational issues that arise from the inherent complexity of epistemic reasoning. This is achieved by combining techniques from the Multi-agent Epistemic Planning (MEP) field and the more efficient classical planning approaches. Due to this combination, our architecture can benefit from the generality derived from MEP solvers and the efficiency of the classical ones. Broadling, our planner considers both an epistemic and a classical description of the planning problem and combines the two solving techniques in a hierarchical way. As we will see later (Section 3) in much greater detail, the architecture first solves the problem with greater abstraction using the epistemic solver. Then, the classical resolution process is used to refine the world-altering actions suggested by the epistemic counterpart.

While ECHO is an architecture that is independent of the specific planners, in this work we present a specific version of our tool. In particular, we decided to define two distinct classical planners, both implemented through the *multi-shot Answer Set Programming* (`ASP`) paradigm [23], to tackle the refined reasoning part. The former is a standard solver that employs a search without any heuristics, while the latter exploits the idea of *goal-networks*, as introduced in [46], to take care of the classical planning problems. The idea is that the goal-network one is used when we have extra information on the domain, as often happens, which can help in selecting the right decomposition of the current goal in a set of partially ordered sub-goals. Instead, to tackle the MEP problem, we employed *EFP* [17], an off-the-shelf reasoner that can comprehensively reason on epistemic domains.

Finally, inspired by [6], we also propose the concept of *macros* in the MEP environment. The use of macros helps in reducing the burden of the epistemic planning process, allowing ECHO to address even more multi-agent planning scenarios. In fact, such domains present several complications, e.g. the number of possible actions, which if not addressed correctly could render the solving process unfeasible.

To summarize, in this paper we present ECHO, a hierarchical planning architecture to efficiently solve MEP problems. This, in turn, is comprised of (i) an off-the-shelf epistemic reasoner, of which we enriched the solving process with the concept of macros; and(ii) two `ASP`-based classical planners, one of which exploits accumulated domain knowledge, due to the notion of goal-networks, to guide the search process. The contributions are coded in a `python` library that is able to decode planning problems in various formats and solve them with the ECHO architecture. In particular, this API is available from the URL: http://clp.dimi.uniud.it/sw/ similar in spirit to the Unified Planning Framework of the AIPlan4EU project [1]; it is able to tackle **(1)** classical planning problems (whether enriched or not with goal-networks as defined in [46]); **(2)** MEP problems, as defined in [19] and **(3)** hierarchical MEP problems as defined in this work.

The remainder of the paper is structured as follows: (i) Section 2 provides some background to better understand the various contributions of the paper;(ii) in Section 3 we propose our contributions. In particular, we present the definitions of our architecture and of its principal components, i.e. the ASP solvers and the epistemic solver enriched with the idea of macros;(iii) Section 4, to better illustrate the behaviour and the capabilities of ECHO, presents the case study that motivated our work;(iv) in Section 5 we validate the behaviour of our framework through experimental results. Alongside the results, we also provide some explanatory examples on how to employ ECHO.(v) Finally, in Section 6, we conclude our manuscript by summarizing its main contributions and by discussing some future works.

2 Background

2.1 Classical Planning

The area of *automated planning* is one of the most prominent in Artificial Intelligence (AI). This field studies how to devise tools that help us in deciding the best course of actions to reach a given goal, an activity that is done continuously in our life. Automated planning, therefore, represents one of the most interesting aspects of AI and, consequently, has been vastly studied [30, 41, 44, 45].

In its 'basic' form, planning is usually referred to as *classical planning*. This setting represents the most known and studied variation of automated planning and has been continuously improved since its early life in the '60s. The initial research was focused on ways to formulate the problem of generating long-term plans for achieving goals, for problems of non-trivial size, in a computationally feasible manner [9]. In order to achieve such feasibility and have tractable and approachable problems, classical planning must consider constrained environments, i.e. they have to be (i) *single-agent*: only one entity is acting upon the world;(ii) *static*: the environment is not subjected to external variations but can only be modified by the acting agent;(iii) *deterministic*: each action executed by the agent must have at most one possible outcome; and(iv) *fully observable*: every action executed must be witnessed by the acting agent itself [28].

An example of a classical domain, which respects all the aforementioned conditions, is the well-known Blocksworld (**BW**) domain [29]. Blocksworld, due to its simplicity, is one of the most employed examples when it comes to explaining the basics of planning. This domain consists of a few simple elements:

- *blocks* of the same size that can be placed either on the table, or on top of another block; and
- a *mechanical arm*—i.e. the acting agent—that can move the blocks and can determine whether it is holding a block or not.

Moreover, there are some constraints that regulate **BW**:

- the mechanical arm can only hold, and therefore move, one block at the same time; and
- a block can only be placed on top of a *clear* block—a block with no blocks on top of it and that is not held by the mechanical arm—or on the table.

In this domain, we have that a single acting agent, i.e. the mechanical arm, wants to move the blocks around in order to achieve a specific disposition of those. This, very informal, specification constitutes a *planning problem* in the classical setting where the properties of the world, e.g. the status of the blocks and the arm, are defined through logical statements called *fluents*. More formally a planning problem is defined as follows:

DEFINITION 1 (Planning Problem).
A *planning problem* is a tuple $\langle D, I, G \rangle$ where:

- D is an *action domain* expressed in some language: i.e. D describes the properties of interest of the environment in which the agents are acting upon and also specifies how the agents themselves can manipulate these properties through actions;
- I is a set of states of the domain—called *Initial state*—that describes the diverse (possible) starting configurations of the world. The initial state of **BW** is the position of the blocks and of the mechanical arm before the plan execution.
- G is a set of states of the domain—called *Goal state*—that describes the desired configurations of the domain. In the case of **BW**, this identifies the disposition of the blocks that we want to reach.

Let us note that I and G are described as sets to keep the definition general. In fact, in various planning domains, we can have multiple initial states, due to partial information for example, as well as multiple desired states. As it is not in the scope of this paper to provide a complete explanation of the broad field of classical planning, we address the interested readers to [9, 28, 45] for a complete introduction on the topic.

2.1.1 Goal-networks As mentioned, classical planning has been studied since the birth of AI. This extensive research has brought, among other things, very interesting and powerful ways to exploit any knowledge that we might have on the domain we want to plan on. In fact, while planning on specific scenarios, it is often the case that our domain knowledge is richer than what we simply write in the formal description of the problems. Even if solvers, in general, should be able to plan without any 'additional' knowledge, not exploiting this extra information would be a waste of resources. From this idea, the whole area of *informed planning* [45] was born. This specific field of study researches ways to exploit any additional knowledge—being it dependent or independent from the domain itself—we might have to enhance the solving process.

Using the previously cited Blocksworld as an example, we are aware that some conditions must be met before others when we want to achieve certain goals. For example, if we want to reach the configuration where block `A` is on the table, we are aware that the condition *clear* block `A` must be achieved before picking `A` and placing it on the table. Therefore, if the condition *clear* `A` is not yet true, we first have to free the blocks placed on top of block `A`. This kind of reasoning is elegantly captured by *goal-networks*, which, in turn, can be introduced in the planning domain. Informally, a goal-network is a partially ordered set (*poset*, for brevity) of goals, where each goal is a pair: *goal id* and *creation time*. The goal id is associated with a specific consistent conjunction of Boolean literals. The creation time is used to disambiguate between goals with the same id.

The precedence relation in the goal-network guides the search for a plan, i.e. at each step we would like to execute an action that leads us to the satisfaction of a goal, chosen among the *minimal* ones, i.e. among the goals that have no predecessors in the goal-network. Once a chosen goal is satisfied in the current state of the world, it can be deleted from the goal-network, and another minimal goal is selected. Sometimes, instead of executing an action, we may attack the current chosen minimal goal by the application of a *method*, which consists of the introduction of new goals that have a greater level of precedence with respect to the chosen goal. More formally, we refer to the definition of goal-network following [46].

DEFINITION 2 (Goal-network).
Let $\mathcal{F}$ be a finite set of propositional atoms also known as *fluents*, $\mathcal{L} = \{f : f \in \mathcal{F}\} \cup \{\neg f : f \in \mathcal{F}\}$ be the set of literals obtained over $\mathcal{F}$ and G_{id} be a set of strings. A *goal-network GN* is a tuple $\langle G, \prec, \alpha \rangle$ where:

- $G \subseteq G_{id} \times \mathbb{N}$ is a finite set of goals.
- $\prec \subseteq G \times G$ is a partial order over G.
- $\alpha : G_{id} \rightarrow 2^{\mathcal{L}}$ is a function such that, for every goal id $g_{id} \in G_{id}$, $\alpha(g_{id})$ is a conjunction of literals in $\mathcal{L}$.

Given a goal $g = \langle g_{id}, t \rangle$, the intended meaning of the goal formula $\alpha(g_{id}) = \varphi$ is that, in order to consider g satisfied, φ must hold in the current state. t represents the time at which the goal g was added to the poset.

Given a goal-network $GN = \langle G, \prec, \alpha \rangle$, a planning domain $D = \langle A, M \rangle$ consists of an action domain A enriched by a set of methods M. A method m is a triplet $\langle head, subgoals, prec \rangle$ where $subgoals(m) = \langle g^1_{id}, \ldots, g^k_{id} \rangle$ is a sequence of goal ids, while $prec(m)$ is a conjunction of literals, expressing the precondition for the method to be executed. We say that the post-conditions of m are $post(m) = \alpha(g^k_{id})$ if $subgoals(m)$ is not empty, otherwise $post(m) = prec(m)$. A method (or an action a) is *relevant* to a goal formula $\varphi_G = \alpha(g_{id})$ if $post(m)$ (or $effects(a)$, respectively) has a non-empty intersection with φ_G and if $post(m)$ (or $effects(a)$, respectively) does not contain any negated literal from φ_G.

The idea is that a method (or an action) can be applied (or executed), only when it is relevant to a current minimal goal. The application of a method m at step t consists of the injection into the current goal-network of the goals $\{(g_{id}, t) \mid g_{id} \in subgoals(m)\}$, with a higher priority in terms of precedence relation, compared with the already existing goals in the goal-network.

As in [46], let us define the set of solutions inductively.

DEFINITION 3 (Solution of the goal-network Planning problem).
Given a goal-network problem P with a goal-network $GN = \langle G, \prec, \alpha \rangle$, a planning domain $D = \langle A, M \rangle$, and a current state s_t at time step t:

- If $G = \emptyset$, then the empty plan is a solution.
- Otherwise, let $g = \langle g_{id}, \_ \rangle \in G$ be a goal in GN without predecessors, and $F = \alpha(g_{id})$ be goal formula to satisfy.
  - If F is already satisfied in s_t, let P' be the problem that results from removing g from GN; if π is a solution to P', then π is a solution to P.
  - Let a be an action that is *relevant* for $\alpha(g_{id})$ and executable in s_t. Let P' be the problem obtained from P and the updated initial state $s'_t = \Phi(s_t, a)$ with the time step $t' = t + 1$. If π is a solution of P' then a, π is a solution to P.
  - Let m a method relevant to g. Then, the set of solutions for P includes the set of solutions for P', in which GN has been replaced by GN', resulting from the application of m to g at time step t.

Let us observe that only actions or methods relevant to the current chosen goal can be executed. This means that the goal-network setting constraints the search for the plan more compared with ordered *landmarks* [32], for instance. While several other studies exist that also take into the idea of *goals preference*, e.g. [27], we leave the investigation of such topic for future works.

2.1.2 The $\mathcal{A}^V$ language In this section, we briefly introduce the $\mathcal{A}^V$ action description language, which has been designed to be integrated into ECHO to define classical planning problems. It essentially corresponds to language $\mathcal{A}$, but some typed variables have been introduced (see [25] as a reference for the nomenclature of planning languages). Typed variables are merely used to write schemes describing finite sets of causal laws that are formed according to the same pattern. Schemes are particularly useful in some application domains, such as when an action that models an agent's movement can be parameterized by a variable that can be instantiated with a series of different positions.

In ECHO the $\mathcal{A}^V$ planning instances are translated into Answer Set Programming (ASP) instances. ASP is one the most prominent logic programming paradigms and it is particularly useful in knowledge-intensive applications (see [40] for an introduction to its semantics). Its use as a planning framework has been deeply explained in [16].

The choice of encoding the planning problems in ASP, already presented in [12], is justified by the fact that ASP as a planning engine outperforms standard task planner for PDDL, where (i) domains are rich in fluents; and (ii) plans are usually short [35]. Such characteristics precisely characterize the situations in which we intend to leverage the classical planner in the robotics domain use cases. Moreover, ASP, due to its declarative nature, lets us easily debug any functionality and test new features while also allowing for formal proof of their correctness.

2.2 Multi-Agent Epistemic Planning

Even if we often need to make decisions based on our beliefs, about the environment, and about others' beliefs, automated planners usually do not consider such intricacy. In fact, as said before, most of the efforts in the planning community address domains where the concept of beliefs and/or knowledge is not taken into account. Nonetheless, the growing interest in AI is pushing researchers to steadily improve and model more realistic scenarios. This momentum brought, among other things, to the formalization of a far more compelling (w.r.t. more classical approaches) form of planning, i.e. the so-called MEP. This area of planning reasons within environments where the streams of 'knowledge' or 'beliefs' need to be considered.

Formalizing and reasoning on the idea of knowledge and beliefs have always been of great interest among various research fields (e.g. philosophy, logic and computer science). In particular, in 1962, Hintikka proposed the first complete axiomatization of these concepts [31]. From this initial effort stemmed the field of *epistemic/doxastic logic* which aims to formalize and reason on information itself. While this area of research presents various challenges, it is only focused on capturing the knowledge relations in static domains. To represent even more interesting and realistic scenarios *Dynamic Epistemic Logic* (DEL) was introduced, i.e. the logic of reasoning on information flows in dynamic domains where agents can act and alter these relations themselves.

DEL represents the foundation of MEP, the setting concerned with finding the best series of actions that modifies the information flows, to reach goals that (might) refer to agents' knowledge/beliefs. In what follows, for brevity, we will use the term 'knowledge' to encapsulate *both* the notions of an agent's knowledge and beliefs. In fact, these concepts are captured by the same modal operator in DEL and their difference resides in structural properties that the epistemic states respect (see [21] for more details). As it is not the objective of this paper to completely present MEP, in what follows we will provide only some fundamental concepts that are necessary to explain the contribution of this work. Far more complete introductions to this topic may be found in [4, 21, 51].

Let us start by presenting the language of well-formed DEL formulae used to express agents' knowledge. This is expressed as follows:

$$\varphi ::= \mathsf{f} \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{B}_{\mathsf{i}}\varphi \mid \mathbf{C}_{\alpha}\varphi,$$

where $\mathsf{f} \in \mathcal{F}$ is a propositional atom called *fluent*, i is an agent that belongs to the set of agents $\mathcal{AG}$ s.t. $|\mathcal{AG}| \geq 1$ φ and ψ are belief formulae and $\emptyset \neq \alpha \subseteq \mathcal{AG}$. A *fluent formula* is a DEL formula with no occurrences of modal operators. A *belief formula* is recursively defined as follows:

- A fluent formula is a belief formula;
- If φ is a belief formula and $\mathsf{i} \in \mathcal{AG}$, then $\mathbf{B}_{\mathsf{i}}\varphi$ (i knows/believes that φ) is a belief formula where the modal operator $\mathbf{B}$ captures the concept of *knowledge*;
- If φ_1, φ_2 and φ_3 are belief formulae, then $\neg\varphi_3$ and $\varphi_1 \circ\mathsf{p} \varphi_2$ are belief formulae, where $\circ\mathsf{p} \in \{\wedge, \vee, \Rightarrow\}$;
- If φ is a belief formula and $\emptyset \neq \alpha \subseteq \mathcal{AG}$ then $\mathbf{C}_{\alpha}\varphi$ is a belief formula, where $\mathbf{C}_{\alpha}$ captures the *Common knowledge* of the set of agents α.

The formula $\mathbf{C}_{\alpha}\varphi$ translates intuitively into the conjunction of the following belief formulae:

- every agent in α knows φ;
- every agent in α knows that every agent in α knows φ; and
- so on *ad infinitum*.

The semantics of DEL formulae is traditionally expressed using *pointed Kripke structures* [38], but other representations are also possible [17, 26]. We refer the interested reader to [8, 21, 26] for a comprehensive introduction to how Kripke structures, or similar formalisms, are used to capture the idea of an epistemic state and how the concept of entailment is defined.

Let us note that the epistemic action language that we will consider in our work implements three *types of action* and three *observability relations*. These are standard concepts in the epistemic planning community and, therefore, we will provide an intuitive description of those addressing the interested reader to [4] for a complete description of this topic. In particular, we assume that each agent can execute one of the following types of action:

- *World-altering* action (also called *ontic*): used to modify certain properties (i.e. fluents) of the world.
- *Sensing* action: used by an agent to refine her beliefs about the world.
- *Announcement* action: used by an agent to affect the beliefs of other agents.

Moreover, each agent is associated with one of the following observability relations during an action execution:

- *Fully-observant*: the agent is aware of the action execution and also knows the effect of the action.
- *Partially-observant*: the agent is aware of the action execution without knowing the effects of the action. Let us note that no agent can be partially observant of an ontic action as it is impossible to decouple the witnessing of a world-altering action and the witnessing of its effects.
- *Oblivious*: the agent is not even aware of the action execution.

Each type of action defines a transition function and alters an epistemic state in different ways. Given the complexity of the topic we address the reader to [4, 17, 18] for a formal definition of these, and others, update functions on diverse epistemic state representations.

Finally, let us introduce the concept of *MEP domain* that, intuitively, contains the information needed to describe a planning problem in a multi-agent epistemic setting.

DEFINITION 4 (MEP Domain).
A *MEP domain* is a tuple $\mathcal{D} = \langle \mathcal{F}, \mathcal{AG}, \mathcal{A}, \varphi_{ini}, \varphi_{goal} \rangle$, where $\mathcal{F}$, $\mathcal{AG}$, $\mathcal{A}$ are the sets of *fluents*, *agents*, *actions* of $\mathcal{D}$, respectively; φ_{ini} and φ_{goal} are DEL formulae that must be entailed by the *initial* and *goal e-state*, respectively. The former e-state describes the domain's initial configuration, while the latter encodes the desired one.

We refer to the elements of a domain $\mathcal{D}$ with the parenthesis operator; e.g. the fluent set of $\mathcal{D}$ is denoted by $\mathcal{D}(\mathcal{F})$. An *action instance* $\mathsf{a}\langle \mathsf{i} \rangle \in \mathcal{D}(\mathcal{AI}) = \mathcal{D}(\mathcal{A}) \times \mathcal{D}(\mathcal{AG})$ identifies the execution of action a by an agent i. Let $\mathcal{D}(\mathcal{S})$ be the set of all possible e-states of the domain. The *transition function* $\Phi : \mathcal{D}(\mathcal{AI}) \times \mathcal{D}(\mathcal{S}) \to \mathcal{D}(\mathcal{S}) \cup \{\emptyset\}$ formalizes the semantics of action instances (the result is the empty set if the action instance is not executable).

2.3 Answer Set Programming

We briefly introduce the syntax and semantics of Answer Set Programming, a dialect of logic programming widely used in knowledge representation and reasoning (see, e.g. [24]).

Let $\mathcal{P}$ be a set of predicate symbols, $\mathcal{F}$ be a set of constant and function symbols and $\mathcal{V}$ be a enumerable set of variable symbols. $\mathsf{ar}(\cdot)$ is a function applied to predicate and function symbols that return the number of arguments. Terms are defined recursively as usual: a variable is a term; if $f \in \mathcal{F}$, such that $\mathsf{ar}(f) = n \geq 0$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Let us observe that if $c \in \mathcal{F}$, $\mathsf{ar}(c) = 0$, then c is a term (a constant). If $p \in \mathcal{P}$, where $\mathsf{ar}(p) = n \geq 0$, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an atomic formula (or atom). Let us observe that if $p \in \mathcal{P}$, $\mathsf{ar}(p) = 0$, then p (without arguments) is an atom, referred as propositional atom. A literal is either an atom A or its negation (as failure) $\texttt{not}A$. An ASP rule is of the form

$$A_0 \ \texttt{:-}\ A_1, \cdots, A_m, \texttt{not}\ A_{m+1}, \cdots, \texttt{not}\ A_n,$$

where $A_0, \ldots, A_n$ are atoms. A_0 is referred to as the *head* of the rule r, while the set of literals $\{A_1, \cdots, A_m, \texttt{not}\ A_{m+1}, \cdots, \texttt{not}\ A_n\}$ is referred to as the *body* of the rule r. A rule is a *fact* if $n = 0$. An ASP program is a collection of ASP rules.

Variables possibly occurring in r are intended as universally quantified. For instance the program

```
person(a).    person(b).    person(c).    person(d).
equal(X,X).
sibling(X,Y) :- parent(Z,X), parent(Z,Y), not equal(X,Y).
```

define the predicate `sibling`: for all `X,Y,Z` if `Z` is parent of `x` and of `Y` and `X` and `Y` are not the same person (they are not `equal`) then `X` and `Y` are siblings.

These universal properties are intended to hold for all the terms the program deals with. Thus, since the constants `a,b,c,d` are used in the facts defining the predicate `person`, then the above

rules can be instantiated:

```
equal(a,a).    equal(b,b).    equal(c,c).    equal(d,d).
sibling(a,a) :- parent(a,a), parent(a,a), not equal(a,a).
...
sibling(a,b) :- parent(c,a), parent(c,b), not equal(a,b).
sibling(b,a) :- parent(c,b), parent(c,a), not equal(b,a).
...
```

This process is called *grounding* and it transforms a first-order program into a propositional program (atoms without variables, i.e. *ground*, can be seen as propositional atoms). In what follows, we will focus on programs that have a finite grounding.

If a program does not use negated literals in rule bodies (definite clause programs), then its semantics is the minimum model semantics (that corresponds to the sets of atoms that hold in all logical models of the program). This set can be computed in polynomial time. In case of ASP programs using negation, the notion of minimum model does not make sense. For instance, the program

```
p(a) :- not q(a).
q(a) :- not p(a).
```

admits the two independent minimal models {`p(a)`} and {`q(a)`}. The semantics of ASP programs is given in terms of *answer sets*; intuitively, an answer set S is a minimal model of the program that *supports* each true atom—i.e. assuming that the atoms in S holds and the others do not hold, for each atom in S there is a rule of the program having such atom as head and whose body is satisfied by S.

The precise definition is given in a *guess and verify* style: a set of ground atoms S is an answer set of a program P if S is the minimum model of the *reduct program* P^S, which is obtained from P and S as follows:

- Remove from P all rules r such that there is a negated literal `not` q in the body of r and $q \in S$;
- Remove all negated literals from the remaining rules.

By construction, P^S is a set of rules that do not contain any occurrence of negation and, as recalled above, it admits a unique minimum model.

Given an ASP program, an ASP-solver is used to compute its answer sets. ASP solvers typically work in two stages: first, the program is grounded, then an answer set of the ground version is looked for. In this paper we will use the ASP solver Clingo [22].

3 The ECHO System

In this section, we will present the original contribution of our work which is, essentially, threefold. We will start by presenting the overall architecture, in Section 3.1, to then illustrate its components with their relative novelties. In particular, Section 3.2 introduces the classical planners, one that uses a standard search process and the other that exploits the goal-network formalism, based on `ASP`. Finally, in Section 3.3 we describe how we implemented the idea of *macros* in MEP, within the planner *EFP* [17].

3.1 The Architecture

Let us now describe the general functionality of the ECHO system, which is the main contribution of this work. The complete code, alongside some working examples and several guides on how to use it, can be found at the following link: http://clp.dimi.uniud.it/sw/. The main object of our framework is to provide a tool that supports MEP and that is able to reason on this problem within acceptable times. We decided to accomplish this through a combination of MEP and classical planning techniques. In particular, ECHO combines these two strategies in a hierarchical way. To be more precise, our architecture first employs MEP for generating *tasks*—actions that have a certain degree of abstraction w.r.t. their functionality in the real world—and then it exploits classical planners to break down these tasks into their refined components.

Once again, we can use the Blocksworld domain to exemplify the difference between the level of abstractions that the epistemic and classical components of ECHO can consider. In particular, if we imagine modelling a 'real-world' version of such domain, where an actual robotic arm has to move blocks the task `move block A on top of B` would be comprised of much more detailed actions: (i) `move the arm on the position of A`;(ii) `pick-up A`;(iii) `move the arm on the position of B`; and(iv) `release A on top of B`. This final set of instructions, when converted into the right formalism (`MoveIt!` commands), represents the atomic actions that the arm should follow in order to execute the plan properly. Nonetheless, when we are planning on the epistemic level we are not interested in such a level of detail but rather in the information flows. For this reason, we introduced the idea of hierarchical planning that allows us to plan on different levels of abstraction. This means that when we are concerned with the epistemic part of the planning we can simply assume that the agent has the ability to move blocks correctly to reduce the computational resources needed by such an intricate process. Once the MEP problem is solved then ECHO can refine all the ontic tasks by finding plans that are actually comprised of atomic actions and can, therefore, be used to manoeuver the arm. To summarize, we employ a hierarchical combination of the two solving techniques (i.e. classical and epistemic) to avoid unfeasibility in the planning process for domains rich in fluents and actions. The key idea is to abstract most of the domain intricacy from the epistemic level and handle it at the classical level, when it is possible, given its vastly superior performances.

Let us now explain in greater detail how the ECHO system processes, solves and eventually executes a planning problem. To better visualize the overall framework, we present a graphical class-based representation of it in Figure 1. In the scheme, we also included the 'actuators' because, as we will see later, we implemented a pipeline that directly sends instructions to actual robotics arms and executes the plan in a real-world domain. While what follows will explain the complete scheme, let us note that when we want to use ECHO in a simulated environment we just need to skip the interactions with the actuators.

First of all the *master process* reads the domain and problem descriptions, which contain the initial state description. This description is specified using a Python encoding that allows defining initial states, goal states, fluents and both epistemic and classical actions, everything using a coherent syntax. ECHO then generates the epistemic planning instance, specified in E-PDDL [20], and sends it to an epistemic planner, i.e. *EFP* [17], which is employed as a black box and returns the sequence of epistemic tasks to be executed. After this first resolution, each task is properly processed following Listing 1.1 to be then converted into a classical planning problem instance. The first step is to break down, or *to flat*, macros (introduced in Section 3.3), then we can reason on a sequence of no macro tasks, hereinafter called *simple tasks*. Intuitively, a simple pure epistemic task can be directly executed. Simple ontic tasks that alter the physical world will undergo further processing. To break

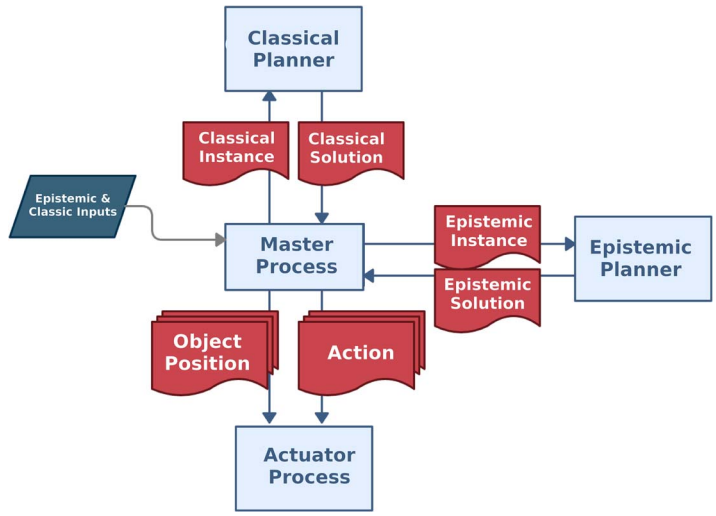


FIGURE 1. A class-based representation of the ECHO system.

down ontic simple tasks into a sequence of classical actions, the *master process* defines an instance of classical planning where:

- the *initial state* is the classical initial state description for the first run of the classical planner, otherwise, it coincides with the final state reached by the previous run of the classical planner;
- the *goal* is extracted from the effects of the ontic simple task if you run as a sub-routing the *plain classical planner*. Otherwise, if the *goal-network classical planner* is used, an initial poset of goals to satisfy is added. This is possible since the library allows to enrich simple ontic tasks with a poset of goals.

Then, iteratively, the *master process* sends the classical action tasks to the *actuator process*, which translates them into `MoveIt!` commands, and makes finally the robots execute the movement.

Pseudo-code for the main steps of ECHO. Let us stress, while our architecture has been devised to tackle hierarchical epistemic problems, the generality of the master process allows ECHO to 'understand' several scenarios. In particular, our framework can tackle: (i) 'pure' classical planning problems that may also be enriched with the knowledge to create goal-networks. In our specific instance of ECHO the resolution of these instances is delegated to the solvers presented in Section 3.2. Nonetheless, these solvers can be easily swapped with alternatives from the literature, e.g. Fast Downward [30], if needed.(ii) Epistemic planning domains where there is no component that must be refined, i.e. all those problems where the classical planning component of ECHO is not employed. The domains can be solved whether it is enriched with the knowledge to 'create' macros (formally presented in Section 3.3), to speed up the solving process, or not.(iii) Finally, as expected, ECHO is able to solve problems defined in the epistemic settings that have 'abstracted' ontic actions that need to be refined by a classical solving process. This type of problem can contain the information to create goal-networks or epistemic macros even if it is not necessary.

LISTING 1.1 Pseudo-code for the main steps of ECHO.

```
def process_e_plan(list<task> task_plan, c_init_state c_init):
    for task in task_plan:
        if is_macro_task(task):
            simple_task_plan = break_down_macro(macro=task)
            for simple_task in simple_task_plan:
                c_init = process_simple_task(t, c_init)
        else:
            c_init = process_simple_task(task)

def process_simple_task(task t, c_init_state c_init):
    # invariant: not is_macro_task(t)
    if is_pure_epistemic_task(t): # sensing or announcement
        execute(t)
    else:
        c_plan, c_init = send_to_c_planner(ontic_task=t, \
                                           initial_state=c_init)
        for c_action in c_plan:
            send_to_actuator(c_action)
    return c_init
```

3.2 Answers Set Classical Planning

The idea of Answer Set Programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solvers, such as `clingo` [23], to effectively compute a solution. *Answer set planning* was first introduced in [39] and has soon received particular interest among researchers in recent years as illustrated in [48]. In what follows we will present a high-level description of the `ASP` encoding used to define the models to compute plans for single-agent actions. We will begin by introducing the *plain* `ASP` planner and then the one that uses the goal-networks formalism.

3.2.1 Plain `ASP` Planner We begin the description of the encoding by highlighting the most important predicates and describing their meaning:

- `holds(F,T)`: where `F` is the fluent `F`, or its negation, true at step `T`;
- `A(T)`: the action `A` occurs at step `T`;
- `A(T)`: the action `A` is executable at time step `T`.

In order to make the encoding more readable, we divide it into two sets of rules: the first one is domain-dependent, while the second one is present in each encoding. From a notation point of view, $\mathcal{P} = \langle D, \Gamma, G \rangle$ is the instance written in $\mathcal{A}^V$, and $\pi(\mathcal{P})$ is its encoding in `ASP`. To avoid unnecessary clutter we will report simplified versions of the rules and only those that have not-straightforward meaning. For example, we will not report the types of variables when clear from the context.

Domain-Dependent Rules

- Each type definition is reported directly, e.g. *stack*(1..6).;
- For each action `a` with parameters $\mathtt{X}_1, \ldots, \mathtt{X}_n$ the following rules are added:

- The action a definition, with its parameters $X_1, \ldots, X_1$

$$\texttt{action}(\texttt{a}(X_1, \ldots, X_n)) : -\texttt{t}_1(X_1), \ldots, \texttt{t}_n(X_n).$$

  where $\texttt{t}_i$ is the type of the variable and limits the range of values it can be instantiated with.
- The action a executability conditions are encoded as

$$\texttt{exec}(\texttt{a}(X_1, \ldots X_n), \texttt{f}(X_i, \ldots, X_j)) \texttt{:-} \texttt{t}_1(X_1), \ldots, \texttt{t}_n(X_n).$$

- The action a effects $\texttt{f}(X_i, \ldots, X_j)$ are represented with

$$\texttt{causes}(\texttt{a}(X_1, \ldots, X_n), \texttt{f}(X_i, \ldots, X_j)) \texttt{:-} \texttt{t}_1(X_1), \ldots, \texttt{t}_n(X_n).$$

– For each *initially f* in Γ s.t. f is a positive fluent literal we have the rule holds(f, 0).
– For the goal achievement, for each *finally(f)*, we add the fact: goal(f)

Domain-Independent Rules

– We define the opposite of a fluent as

```
opposite(F,neg(F)):- fluent(F).
opposite(neg(F),F):- fluent(F).
```

– The actions executability is expressed with the constraint:

```
not_executable(A,t):- fluent(F),exec(A,F),not holds(F,t).
not_executable(A,t):- opposite(G,F),exec(A,F),holds(G,t).
    executable(A,t):- not not_executable(A,t),action(A).
```

– The inertia is guaranteed with the rules:

```
holds(F,t+1):- opposite(F,G),holds(F,t),not holds(G,t+1).
```

– Only one action occurs at each step:

```
1{occurs(A,t) : action(A)}1.
```

– We satisfy executability with the following constraint:

```
:- action(A),occurs(A,t),not_executable(A,t).
```

Note that all the rules that contain the term t belong to the subprogram #program step(t). We also add the following subprogram #program check(t) to check the satisfaction of the goal:

```
:- goal(F),not holds(F,t+1),query(t).
:- goal(neg(F)),holds(F,t+1),query(t).
```

The length for the trajectory solution is t. This compilation exploits clingo multi-shot features [23].

3.2.2 Goal-Network ASP Planner In order to integrate the goal-network concept into the ASP planner we enrich the previous compilation with the following predicates:

- `selected_goal(G,T1,T2)`: which is the selected goal among those minimal.
- `relevant(M,T)`: the method `M` is relevant for the selected goal at time step `T`.
- `relevant(A,T)`: the action `A` is relevant for the selected goal at time step `T`.
- `goal_to_sat(G,T1,T2)`: where `T1` is the time of the creation of the goal, `T2` is the current time and `G` is the id of the goal. We need the creation time because the same goal may appear more times in the same goal-network.
- `prec_to_sat(G1,T1,G2,T2)`: where `T1` and `T2` are creation times, this predicate encodes the precedence relation between introduced goal `(G1,T2)` and `(G2,T2)`.

We also enriched the sets of domain-dependent and -independent rules with the following rules.

Domain-Dependent Rules

- For each goal `G` labelled with a set of literals $\{l_1, \ldots, l_n\}$, we introduce the fact: $\texttt{goal\_labeled(Gl}_\texttt{i}\texttt{)}$ for each i.
- For each method `M` that, upon execution, enriches the current poset with new goals and precedence relationships to be fulfilled, we introduce

```
method_req(M,G1,G2).
method_end(M,G).
```

- For each method $m(X_1, \ldots, X_n)$ we introduce the fact: $\texttt{method(m(X}_1\texttt{,}\ldots\texttt{,X}_\texttt{n}\texttt{))}$, and specify its executable conditions.

Domain-Independent Rules

- We exploit the `ASP`'s expressive features to identify the goals that are presently *minimal* concerning the precedence relation within the current poset of goals to be satisfied:

```
not_minimal(G2,T2,t):- prec_to_sat(G1,T1,G2,T2),
                       goal_to_sat(G1,T1,t),goal_to_sat(G2,T2,t).
not_minimal(G2,T2,t):- T2 < T1,
                       goal_to_sat(G1,T1,t),goal_to_sat(G2,T2,t).
    minimal(G,T1,T2):- not not_minimal(G,T1,T2),goal_to_sat(G,T1,T2).
```

- We introduce a predicate to select a goal among the minimal ones:

```
1{selected_goal(G,T,t) : minimal(G,T,t)}1.
```

- In order to execute a method or an action, we must guarantee that it is `relevant` for the `selected_goal`.

- For methods we introduce the following rules:

```
not_relevant(M,t):- method(M),goal_labeled(G,F),method_end(M,G),
                    selected_goal(SG,ST,t),goal_labeled(SG,NEGF),
                    opposite(F,NEGF).
    relevant(M,t):- method(M),selected_goal(SG,ST,t),
                    goal_labeled(SG,F),method_end(M,MG),
                    goal_labeled(MG,F),not not_relevant(M,t).
```

- For actions we introduce the following rules:

```
not_relevant(A,t):- action(A),selected_goal(SG,ST,t),causes(A,F),
                    goal_labeled(G,NEGF),opposite(F,NEGG).
    relevant(A,t):- action(A),selected_goal(SG,ST,t),causes(A,F),
                    goal_labeled(SG,F),not not_relevant(M,t).
```

– For the sake of readability we omit the rules for the *goal_to_sat* inertia. We just mention that when a goal is satisfied in a state, it is no more propagated.
– We have to model also what happens when a method occurs:

```
goal_to_sat(G,t+1,t+1):- occurs(M,t),method_req(M,G,G1).
goal_to_sat(G,t+1,t+1):- occurs(M,t),method_end(M,G).
prec_to_sat(G1,t+1,G2 t+1):- occurs(M,t),method_req(M,G,G1).
```

– We guarantee that at most one operation among methods and actions is executed each time. It may happen that no operation occurs at all when we select a goal such that it is already satisfied in its current state. In this case, we do not propagate this goal to the next step.

As in the previous encoding, all the rules that contain the term *t* belong to the subprogram `#program step(t)`. To check the satisfaction of the goal we also add the following subprogram `#program check(t)`:

```
:- goal_to_sat(G,T,t+1),query(t).
```

This states that after the execution of the last action or method, we do not want any goal in the current goal-network, i.e. there is no goal to satisfy.

3.3 Macros in MEP

As already mentioned, a significant contribution of this work is the formalization, and consequent employment, of *macros* in the MEP setting. A *macro*, which can be informally described as 'an encapsulated sequences of elementary planning operators', is formally defined as follows:

DEFINITION 5 (Macro).
Let $\mathcal{D}(\mathsf{i}), \mathcal{D}(\mathsf{j}) \in \mathcal{D}(\mathcal{AI})$ be two action instances, and $\mathsf{s} \in \mathcal{D}(\mathcal{AG})$ be an e-state of a given a domain . A macro $\mathsf{m}_{\mathsf{i,j}} \in \mathcal{D}$ representing the subsequent execution of j after i can be defined as $\Phi(\mathsf{j}, \Phi(\mathsf{i}, \mathsf{s}))$. Let us note that *i)* we assume that if action a is not executable in a state s, then the result of the update $\Phi(\mathsf{a}, \mathsf{s})$ is $\perp$; and *ii)* the execution of any action a over $\perp$ results in $\perp$ as well.

The introduction of macros is justified by the fact that, often, patterns of actions performed in sequence are repeated in the same domain. For example, it often happens that an ontic action is followed by an announcement action, because an agent may want to communicate the results of the former to another (oblivious) agent. For now, as pointed out in the Dagstuhl seminar [3], there have been many proposals in the classification of epistemic actions. As pointed out above, the community mostly agrees that the basic classification considers *ontic*, *announcement* and *sensing* as possible categories. Among these three types of actions, only the ontic one has some effect on the physical world. That is why we consider the remaining two, i.e. sensing and announcement, as *purely epistemic* actions. Consequently, we say that a task is purely epistemic if it involves only announcement or sensing actions, or a macro aggregating these two types. In Listing 1.1 the function call `is_pure_epistemic_task(task)` checks exactly this property.

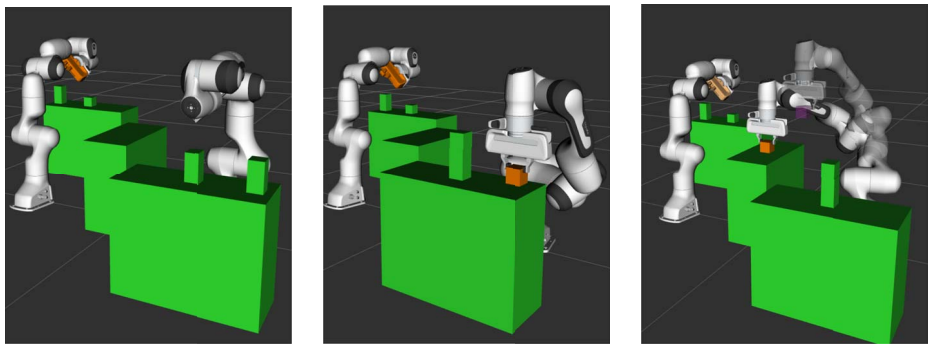
4 Case study

We can now present an application of ECHO in a two-agent scenario. The framework here presented has been validated in a simple multi-agent scenario in which two Franka Emika robots have been involved. Franka Emika is a robotic arm with 7 Degrees Of Freedom (DOF) with torque sensors at each joint. It is also equipped with a gripper that allows the automated 'arm' to handle objects. The client side of the Franka Control Interface is called `libfranka`. At a higher level we find `franka_ros`, which is an ROS [42] package that contains the description of the robot and the end-effector in terms of kinematics, joint limits, visual surfaces and collision space, a hardware abstraction of the robot for the ROS control framework based on the `libfranka` API, and a set of services to expose the full `libfranka` API in the ROS ecosystem. Let us note that, although the target robots are two Franka Emika, the architecture can be adapted to another collection of manipulators with only minor changes. The diverse problem instances contain several fluents and actions, as usual in the robotics domain.

The two agents involved are the two robotic arms, called `robot1` and `robot2`. In front of each of them, three stacks of coloured blocks are placed on a *private table*. Figure 2a presents an example of a state. Initially, the two robots may ignore what blocks are placed in front of them, but they can execute a sensing action to check if they have a particular block. They can also move a block from the top of a stack to the top of another stack or to a *shared table* in between the two robots. We assume that only once a robot has placed a block on the *shared table*, it is able to communicate the colour of such a block to the other arm. Examples in Figure 2 are obtained using the RViz simulator [42] integrated into `MoveIt!`.

4.1 Modelling the Problem Instance

When we model a scenario as a planning problem in order to eventually effectively execute it, we need to address the 'anchoring' problem. i.e. we need to connect the model description to the physical objects in the real (or simulated) world. Let us take the process of grasping, executed by an automated arm, as an example. Grasping is usually composed of different phases: (i) *pre-grasping*, in which the *end-effector* approaches the object to be grasped with a given direction and orientation. (ii) Actual *grasping*, in which the final joints of the gripper close. Finally, (iii) *post-grasping*, in which the end-effector moves away from the position in which it grasped the object in a given direction and orientation. During each of these sub-processes, we must ensure that no collisions occur between the robotic arm and the objects in its workspace. In the approach we adopted, we hard-coded the execution of the three grasping sub-processes in the `ROS` program, modelling at the



(a) Initial state. (b) A robot picking up a block from its table. (c) A robot placing a block on the shared table.

FIGURE 2. Examples of the robotic test environment states. These images have been generated using RViz. A video of the 'real' execution with the Panda Robots is available: http://clp.dimi.uniud.it/sw/.

action description level only the 'act of grasping'. Since this is not the focus of this article, we just mention that we used the `MoveIt!` library to calculate collision-free trajectories. `MoveIt!` is provided with information about the location of objects to calculate trajectories with fast algorithms, such as *Rapidly-exploring Random Tree*. The initial state of the domain, the effect of the action and some static information of the environment, like the position of the *shared table*, lead us to update the current state of the `MoveIt!` setting before and after the execution of an action under a closed world assumption.

The two modelling action domains setting in ECHO allows the modeller to decide what to model at an MEP level and what to a classical level. The main concept is that the classical action description domain can be viewed as a refinement of the ontic actions of the MEP action description domain. For instance, we are not interested in modelling the picking and placing actions at an epistemic level as ontic actions, but we just want an ontic action that models the fact of moving a block from the *private table* to the *shared table* and *viceversa*. Furthermore, the set of MEP fluents is a subset of the classical fluents. Fluents that appear at the MEP level are the only ones over which we want to characterize the beliefs of agents. However, to make the agent effectively act in the real world, we may enrich the classical description with more fluents. With respect to this study case, we just want to express the fact that an agent owns a block, and we push down to the classical level the fact that a block may be on top of another block.

For the sake of readability, we will not report all the actions' descriptions, rather we will show only some meaningful examples. Since ECHO provides an intuitive and user-friendly `python` API interface, we show how to describe a few actions both of an MEP and classical domain, and how they are then compiled, respectively, in E-PDDL and ASP. The initial state description and the fluents encoding are also presented. Let us start, by showing how to define types, fluents and then variables in Listing 1.2. Examples of types, fluents and variables.

Then, in Listing 1.3, we present how to define predicates by providing the following examples:

- literals
- disjunction and conjunction of predicates
- belief and common knowledge formulas

LISTING 1.2 Examples of types, fluents and variables.

```
#types:
stack = IntType('stack', 1, 6)
agent = AgentType(['robot1', 'robot2'])
color = EnumType('colo', ['red', 'orange'])
color_pair = StructType('colorXcolor', [color, color])
agent_color = StructType('agentXcolor', [agent, color])

#fluents:
on_block = Fluent('on_block', color_pair)
owner = Fluent('owner', agent_color)
free_gripper = Fluent('free_gripper')

#variables:
C1 = Variable('C1', color)
R = Variable('R', robot)
```

LISTING 1.3 Examples of predicates.

```
free_gripper()
-free_gripper()

disj(free_gripper(), -free_gripper())
conj(free_gripper(), owner(R, 'red'))

B([R], owner(R, 'red'))
C(['robot1', 'robot2'] owner('robot1', 'red'))
```

Examples of predicates. We can express both classical actions and epistemic tasks with the provided ECHO API. Since actions are a fundamental part of the domain description, let us show some examples of their encoding in Listing 1.4. In particular, we will present the following actions:

- `pick`, a classical action that describes the grasping of a block situated on the top of a stack.
- `from_private_to_shared`, which is an ontic task that describes the process of moving an object from a *private table* to the *shared table*.
- `check`, a sensing task in which the robot checks if there is a block of a specific colour in front of it.

As previously mentioned, during the execution of ECHO these actions are compiled into ASP and E-PDDL. We show in Listing 1.5 a fragment of the compiled representation of the actions defined in Listing 1.4. For a detailed explanation of the syntax of E-PDDL, we address the reader to [20].

Let us now consider what happens when an epistemic ontic action appears in the epistemic plan. Consider a configuration in which the red block is stacked under a black block and the epistemic ontic action suggests moving the red block on the *shared table*. The robot cannot take directly the red block, but it should first move the black one at the top of another stack and then take the red

LISTING 1.4 Examples of actions.

```
pick = Action(
      name = 'pick',
      params = [R, C1, C2],
      precondition = [top(C), on_block(C1, C2),
                      free_gripper(R), owner(R, C1)],
      effects = [-top(C1), -on_block(C1, C2), top(C2),
                 -free_gripper(R), gripped(R, C1)]
)

from_private_to_shared = MEAction(
      name = 'from_private_to_shared',
      params = [A1, C1],
      precondition = [owner(A1, C1), free_table(),
                      B([A1], owner(A1, C1)],
      effects = [-owner(A1, C1), on_table(C1), -free_table()],
      full_obs=[Forall(A1, who=A2)]
)

check = MEAction(
      name = 'check',
      type = MEActionType.sensing,
      params = [A1, C1],
      effects = [When(owner(A1, C1), owner(A1, C1)],
      full_obs=[A1]
)
```

block to put it on the shared table. Therefore, such an epistemic ontic task may be translated into the sequence of actions presented in Listing 1.6.

5 Experimental results

To further assess and demonstrate the capabilities of the developed framework, we tested our architecture on a wide spectrum of scenarios that stem from the configuration described in Section 4. Experiments showed the positive impact of using macros in the MEP setting. Furthermore, we were also able to prove the feasibility of the solving procedure in real-world situations.

All the experiments have been conducted on an Intel i7-8565U CPU at 1.80GHz and Ubuntu 18.04 OS. For each problem instance, a time limit of 120 s (2 min) was applied. Clingo [23] was used to solve the ASP encoding, and EFP [17] to tackle the resolution of MEP problems. The benchmarks used in this manuscript and other encoding examples can be found in http://clp.dimi.uniud.it/sw/. Other than the domain presented in Section 4, we also have formalized the (i) 'pure' classical Blocksworld domain [29]; (ii) the **Coin in the Box** [34, 37] domain, which does not have actions that require refinement; (iii) the **Table** domain, which is used to show an example of a domain enriched with the knowledge to form a goal-network; and (iv) the well-known **Gossip Problem** [15].

Table 1 reports the confrontation of three solving approaches for an MEP domain. In particular, we confront the solving times of *EFP*, the solver that is used as the epistemic planner in ECHO, with

LISTING 1.5 Examples of ontic actions.

```
(
 :action         pick_and_place_on_shared
 :act_type       ontic
 :parameters     (?ag - agent ?c - color)
 :precondition   (and ([?ag](owner ?ag ?c))
                      (free_table)
                      (owner ?ag ?c))
 :effect         (and -(owner ?ag ?c), (on_table ?c),
                                       -(free_table))
 :observers      (forall (?ag2) (?ag2))
)

(
 :action         check
 :act_type       sensing
 :parameters     (?ag - agent ?c - color)
 :effect         (and (when (owner ?ag ?c) (owner ?ag ?c)))
 :observers      (?ag)
)
```

LISTING 1.6 List of single-arm actions.

```
% pick the black block over the red one
pick (black , red)

% place the black block over the green one
place (black ,green)

% pick the red block from stack 2
pick_from_ground (red ,2)

% place the red block on the shared table
put_on_barrier (red)
```

two variations of ECHO itself. These two configurations are the plain version of ECHO and the one enriched with the concept of macros. This will permit us to highlight how the use of macros can help in reducing the solving time.

The test cases are obtained with a cross-product between goals of increasing difficulty with an increasing number of blocks inserted in the domain (rows and columns of Table 1, respectively). In particular, we have that:

- α has the objective to let one agent know the colour of at least one block initially located in the table in front of the other robot;
- β has the same goal as α while also requesting that the shared table must be free at the end of the plan;

TABLE 1. Time, in seconds, to find a goal, given an initial state. Each instance varies the number of available colours. In **boldface** is highlighted the fastest solving process w.r.t. the activation of the *from_private_to_shared_announce* and *from_shared_to_private_announce* macros. Let us note that all the values were obtained by averaging the times of 5 iterations on the same instance.

	1 Colour			**2** Colours			**3** Colours			**4** Colours		
Macro	EFP	ECHO	ECHO-M	EFP	ECHO	ECHO-M	EFP	ECHO	ECHO-M	EFP	ECHO	ECHO-M
α	TO	0.019	**0.004**	TO	1.200	**0.080**	TO	5.192	**1.382**	TO	104.000	**25.889**
β	TO	0.083	**0.013**	TO	4.946	**0.242**	TO	21.780	**4.235**	TO	TO	**85.144**
γ	TO	ND	ND	TO	18.663	**1.427**	TO	TO	**24.805**	TO	TO	TO
δ	TO	ND	ND	TO	TO	**4.827**	TO	TO	**92.998**	TO	TO	TO

- γ encodes the scenario where the goal is for a robot to know two different colours of blocks initially placed in front of the other arm;
- δ shares the goal of γ with an 'extra' condition imposing that the shared table must be free when the planning process is concluded.

For the sake of readability we will use the following notations in Table 1:

- ND that stands for Not Defined. This is used e.g. to indicate that instance γ cannot be performed with just one colour as it required that a robot learns two different colours while only one is available.
- TO that stands for Time Out. As said before, after 110 s the planning procedure is forcefully stopped if it could not find a solution.

From the domain it is evident that when an agent takes a block from the shared table, to place it on top of one of its stacks, it often happens that the agent announces that the table has been freed. This small sequence of actions constitutes the macro `announce:from_private_to_shared`. As mentioned before, in Table 1, we compare the solving process when this macro is not activated (ECHO) and when it is (ECHO-M) as well as when we only employ *EFP* (EFP).

To summarize, Table 1 shows that *EFP* has always outperformed and that macros improve considerably the performance. This shows that ECHO, especially when coupled with the concept of macros, permits tackling epistemic domains in reasonable times. The use of ECHO makes it possible to have better scalability both in terms of the number of fluents and of plan lengths, thereby introducing the possibility to reason on an epistemic level in complex multi-agent environments.

For the sake of completeness, let us now address the initial state generation. This is the first step performed during the calculation of the epistemic plan. We report evaluations for this task as it is well established in the epistemic planning literature that generating the initial state from its description is a very resource-heavy problem that also requires specific formalization constraints [17, 49]. This process is empirically almost independent of the use of macros, but it is affected by the number of fluents and actions used in the planning model. To better exemplify this, let us report the average times needed to compute the initial states, increasing the number of fluents, in our case the number of coloured blocks: **(1)** for one colour 0.003 s;**(2)** for two colours 0.031 s;**(3)** for three colours 0.126

TABLE 2. Cumulative time, in seconds, to break down ontic tasks into classical actions. The number of calls to the classical planner is reported in the column 'calls'.

	1 Colour		**2** Colours		**3** Colours		**4** Colours	
	time	calls	time	calls	time	calls	time	calls
α	0.074	1	0.083	1	0.092	1	0.112	1
β	0.157	2	0.172	2	0.177	2	0.202	2
γ	ND	ND	0.250	3	0.267	3	TO	TO
δ	ND	ND	0.332	4	0.351	4	TO	TO

s; and**(4)** for four colours 2.117 s. As a consequence of this observation, it is a good rule of thumb to limit the number of fluents considered to the strict necessary required for epistemic planning.

As a design choice, we decided to run the classical planner at run-time, each time its use is required to break down an ontic task in a sub-plan of classical actions. Justification for this choice is provided by Table 2, in which, for each instance, the cumulative time required by the A^V solver and the times it was called are reported. Let us note that the times required by the epistemic planner without the support of the classical one, i.e. the times required by the solver if the domain was entirely described at the epistemic level were always higher than the timeout.

Finally, let us justify the introduction of the goal-network formalism that could take the place of the 'plain' classical solver as sub-routing to break down ontic tasks into simpler actions executable by a robot. In what follows we illustrate the main characteristics of scenarios in which goal-network formalism provides advantages with respect to the 'plain' solver. (i) The actions that are (usually) needed to satisfy the goals are just a small subset of the total number of available actions. (ii) Specific domain knowledge is present and can be formalized and injected into the solving process with the goal-network. (iii) Specific order of execution between the action needs to be enforced. In what follows, we will provide some experimental results to support (i) and (ii). For condition (iii) we argue that, while it might be met by modifying the actions preconditions or iterating over successful plans produced by the 'plain' solver (eliminating those plans that do not respect the right action order), the use of a partially ordered set of goals is far more elegant and clean.

To better highlight the comparison between the 'plain' and the goal-network formalism, we decided to eliminate the epistemic components from the problem description. This means that the study case is a restricted version of the main one previously discussed, where we require that a robot moves the block at the bottom of a stack to the shared table.

We compared the 'plain' classical planner with the goal-network planner enriched with different types of knowledge. For the sake of readability let us introduce some notations.

- `plain` represents the 'plain' classical planner that does not exploit any addition domain or instance knowledge.
- `domn` represents the goal-network planner used in combination with knowledge at the domain level. This extra information, shown in Listing ??, specifies the following method: in order to pick a block `C2` that is currently under block `C1`, you have to (i) pick `C1` first, (ii) place `C1` to another location and (iii) finally pick `C2`.
- `inst` represents the goal-network planner used in combination with knowledge at the instance level. This knowledge is encoded in Listing ?? and suggests a sequence of sub-goals that needs to be satisfied in order to place the `red` block on the shared table. Note that such a sequence depends strongly on the configuration of the stacks.

LISTING 1.7 Encoding of the domain knowledge exploitable by the goal-network planner.

```
pick_not_free = Method('pick_not_free',
    [R, C1, C2], [on_block(C1, C2)],
    Poset([picked_g(R, C1), placed_g(R, C1, C2),
        picked_g(R, C2)]
    )
)

picked_g = Goal('picked_g', [R, C1], [gripped(R, C1)])

placed_g = Goal('placed_g', [R, C1, C2],
    [-gripped(R, C1), -on_block(C1, C2)]
)

on_table_g = Goal('on_table_g', [C1],
    [on_table_g(on_table(C1))]
)

#Initial PoSet
Poset([picked_g(['bob', 'red']), on_table_g('red')])
```

LISTING 1.8 Encoding of the instance knowledge exploitable by the goal-network planner.

```
Poset([picked_g(['bob', 'yellow']),
       placed_g(['bob', 'yellow', 'black']),
       picked_g(['bob', 'black']),
       placed_g(['bob', 'black', 'red']),
       picked_g(['bob', 'red']),
       on_table_g('red')]
)
```

Encoding of the domain knowledge exploitable by the goal-network planner. Encoding of the instance knowledge exploitable by the goal-network planner. Ultimately, we also aimed to test our classical planning solver when the 'useful' actions constitute only a subset of all the possible actions. To evaluate this scenario, we incorporated *dummy actions*, which lack any significant impact on the given planning problem, and their sole purpose is to increase the size of the planning description. Table 3 shows the execution times (in seconds) of `plain`, `domn` and `inst` on the case study (stripped of its epistemic component) when also enriched with 0, 500, 1000 and 1500 dummy actions.

Table 3 illustrates that the inclusion of knowledge regarding the planning domain and the single instance is advantageous in terms of computational time when the size of the action description is significant. Conversely, in situations where the action description is small, the `plain` planning method outperforms the goal-network approach due to the additional overhead introduced by the inclusion of methods and goal rules in the `ASP` program. Our interest in potentially huge planning domains stems from the fact that our framework is designed to refine ontic actions, even if it can only

TABLE 3. Execution times, in seconds, to solve the case study (stripped of its epistemic component) domain with the diverse configurations of the classical planner.

|Dummy Actions|	`plain`	`domn`	`inst`
0	**0.03**	0.12	0.05
500	12.27	16.20	**0.91**
1000	46.83	45.38	**23.95**
1500	99.60	88.70	**58.58**

be utilized for classical planning problems. In principle, the action description could be significantly large, and for each refinement task, only a fraction of it may be necessary.

Notably, the `inst` section yields superior results compared with the other methods due to the guided search for the plan, which requires sub-goals to be satisfied in a specific order. Although one may argue that this approach is unrealistic in a planning scenario, we emphasize that the addition of instance-based knowledge is not obligatory but can be beneficial if utilized, and our approach facilitates its integration. Additionally, we observe that methods are more effective when they are not parameterized by numerous values due to the grounding phase, and we suggest a maximum of three variables would be a suitable modelling choice.

6 Conclusions

While the field of cognitive robotics has made significant progress over the last few years, there are still some open questions regarding how to integrate new AI components. Moreover, multi-agent scenarios where the acting entities can perform low-level sensing and control tasks are becoming more and more available both in the industrial and academic environments. And, at the same time, epistemic planning is receiving a lot of interest.

In [13] and in [14] Capitanelli *et al.* proposed a set of PDDL+ formulations that allows modelling the problem of manipulating articulated objects in a three-dimensional workspace with a dual-arm robot. Instead, [6] addresses the same domains while encoding the planning problem directly in `ASP` making strong use of macros. Another similar tool that inspired our research is the `ROSoClingo` [2] `ROS` package. This framework integrates Clingo [23] into the `ROS` service and `actionlib` architecture, providing a high-level `ASP`-based interface to control the behaviour of a robot. While these works provided the foundation for our research and are far more complete tools, they do not consider either the information flows between agents, or their knowledge. This aspect is key in every multi-agent scenario, where reasoning from the perspective of others should be taken into consideration.

That is why we proposed ECHO, a framework that offers the possibility of modelling MEP problems in 'rich' multi-agent environments built upon the architecture firstly presented in [47]. This tool stems from the aforementioned approaches and integrates the latest MEP techniques to tackle the planning problems also considering the information flows. While integrating the epistemic aspect of the multi-agent domain is, in our opinion, of utmost importance, it requires high computational resources. That is why ECHO also introduces two methods to improve the planning times and to have feasible solving processes: (i) a hierarchical usage of epistemic and classical planners to abstract and simplify the problems when considered by epistemic solvers; and (ii) the employment of macros in epistemic planning.

While the presented results and the performances of ECHO are very promising, we are currently working on some aspects to improve even further our architecture. First of all, we are defining how to automatically define macros, action refinements and other components that are now part of the input definition. We are planning on doing that by extrapolating experience from previous executions of ECHO that would highlight which actions can form a macro or which one of them can be easily abstracted into an epistemic task. Another idea that we are currently investigating is the one to decouple ECHO from the idea of epistemic planning and use it to 'hierarchically' solve diverse types of planning, exploiting the extremely optimized techniques derived from classical planning as a base. Finally, we are also working on improving the ASP planners themselves, to make them faster and more flexible e.g. by permitting the definition of custom heuristics.

Funding

The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

Acknowledgements

The research presented in this paper is partially supported by Fondazione Friuli/UniUD project AI4HRC, by Interdepartmental Project on AI (Strategic Plan UniUD–22-25), by INdAM-GNCS projects CUP E55F22000270001 and E53C22001930001 and by the European Union's Horizon 2020 research and innovation programme—Grant Agreement No 101034440.

References

[1] AIPlan4EU https://www.aiplan4eu-project.eu/, 2022. [Online; accessed 10-Nov-2022].

[2] B. Andres, P. Obermeier, O. Sabuncu, T. Schaub and D. Rajaratnam. *ROSoClingo: A ROS Package for ASP-Based Robot Control* arXiv preprint arXiv:1307.7398, 2013, 1–14.

[3] C. Baral, T. Bolander, H. van Ditmarsch and S. McIlrath. Epistemic planning (Dagstuhl seminar 17231). *Dagstuhl Reports*, **7**, 1–47, 2017.

[4] C. Baral, G. Gelfond, E. Pontelli and T. C. Son. *An Action Language for Multi-Agent Domains: Foundations* arXiv preprint arXiv:1511.01960, 2015, 1–49.

[5] C. Baral, G. Gelfond, E. Pontelli and T. C. Son. An action language for multi-agent domains. *Artificial Intelligence*, **302**, 103601, 2022.

[6] R. Bertolucci, A. Capitanelli, C. Dodaro, N. Leone, M. Maratea, F. Mastrogiovanni and M. Vallati. An ASP-based framework for the manipulation of articulated objects using dual-arm robots. In *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings, Volume 11481 of Lecture Notes in Computer Science*, M. Balduccini, Y. Lierler and S. Woltran, eds, pp. 32–44. Springer, 2019.

[7] M. Bhatt and E. Erdem. Cognitive robotics. In *Fredrik Heintz, and Michael Spranger*. Cognitive robotics, vol. 28, pp. 779–780, 2016.

[8] T. Bolander. *A Gentle Introduction to Epistemic Planning: The DEL Approach* arXiv preprint arXiv:1703.02192, 2017, 1–22.

[9] T. Bolander and M. B. Andersen. Epistemic planning for single-and multi-agent systems. *Journal of Applied Non-Classical Logics*, **21**, 9–34, 2011.

[10] T. Bolander, M. H. Jensen and F. Schwarzentruber. Complexity results in epistemic planning. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, Q. Yang and M. J. Wooldridge, eds, pp. 2791–2797. AAAI Press, 2015.

[11] M. Bozzano, R. Bussola, M. Cristoforetti, M. Jonas, K. Kapellos, A. Micheli, D. Soldà, S. Tonetta, C. Tranoris and A. Valentini. RobDT: AI-enhanced digital twins for space exploration robotic assets. In *The 2022 International Conference on Applied Intelligence and Informatics (AII2022)*, September, 2022.

[12] A. Burigana, F. Fabiano, A. Dovier and E. Pontelli. Modelling multi-agent epistemic planning in ASP. *Theory Pract. Log. Program.*, **20**, 593–608, 2020.

[13] A. Capitanelli, M. Maratea, F. Mastrogiovanni and M. Vallati. Automated planning techniques for robot manipulation tasks involving articulated objects. In *AI∗IA 2017 Advances in Artificial Intelligence - XVIth International Conference of the Italian Association for Artificial Intelligence, Bari, Italy, November 14-17, 2017, Proceedings, Volume 10640 of Lecture Notes in Computer Science*, F. Esposito, R. Basili, S. Ferilli and F. A. Lisi, eds, pp. 483–497. Springer, 2017.

[14] A. Capitanelli, M. Maratea, F. Mastrogiovanni and M. Vallati. On the manipulation of articulated objects in human–robot cooperation scenarios. *Robotics and Autonomous Systems*, **109**, 139–155, 2018.

[15] Martin C. Cooper, Andreas Herzig, Faustine Maffre, Frédéric Maris and Pierre Régnier. Simple epistemic planning: Generalised gossiping. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum and F. van Harmelen., editors, *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, the Hague, the Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016), Volume 285 of Frontiers in Artificial Intelligence and Applications*, pages 1563–1564. IOS Press, 2016.

[16] A. Dovier, A. Formisano and E. Pontelli. Perspectives on logic-based approaches for reasoning about actions and change. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of his 65th Birthday, Volume 6565 of Lecture Notes in Computer Science*, M. Balduccini and T. C. Son, eds, pp. 259–279. Springer, 2011.

[17] F. Fabiano, A. Burigana, A. Dovier and E. Pontelli. EFP 2.0: A multi-agent epistemic solver with multiple e-state representations. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, J. C. Beck, O. Buffet, J. Hoffmann, E. Karpas and S. Sohrabi, eds, pp. 101–109. AAAI Press, 2020.

[18] Francesco Fabiano, Alessandro Burigana, Agostino Dovier, Enrico Pontelli and Tran Cao Son. Multi-agent epistemic planning with inconsistent beliefs, trust and lies. In D. N. Pham, T. Theeramunkong, G. Governatori and F. Liu, editors, *PRICAI2021: Trends in Artificial Intelligence - 18th Pacific Rim International Conference on Artificial Intelligence, PRICAI 2021, Hanoi, Vietnam, November 8-12, 2021, Proceedings, Part I, Volume 13031 of Lecture Notes in Computer Science*, pages 586–597. Springer.

[19] F. Fabiano, B. Srivastava, J. Lenchner, L. Horesh, F. Rossi and M. B. Ganapini. *E-Pddl: A Standardized Way of Defining Epistemic Planning Problems* arXiv preprint arXiv:2107.08739, 2021, 1–9.

[20] F. Fabiano, B. Srivastava, J. Lenchner, L. Horesh, F. Rossi and M. B. Ganapini. E-PDDL: A standardized way of defining epistemic planning problems *CoRR*, abs/2107.08739, 2021.

[21] R. Fagin and J. Y. Halpern. Reasoning about knowledge and probability. *Journal of the ACM (JACM)*, **41**, 340–367, 1994.

[22] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. Clingo = ASP + control: Preliminary report *CoRR*, abs/1405.3694, 2014.
[23] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub. Multi-shot ASP solving with clingo *CoRR*, abs/1705.09811, 2017.
[24] M. Gelfond and Y. Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents the Answer-set Programming Approach*. Cambridge University Press, 2014.
[25] M. Gelfond and V. Lifschitz. Action languages. *Electron. Trans. Artificial Intelligence*, **2**, 193–210, 1998.
[26] J. Gerbrandy and W. Groeneveld. Reasoning about information change. *Journal of Logic, Language and Information*, **6**, 147–169, 1997.
[27] A. E. Gerevini, P. Haslum, D. Long, A. Saetti and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, **173**, 619–668, 2009 Advances in Automated Plan Generation.
[28] M. Ghallab, D. Nau and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.
[29] N. Gupta and D. S. Nau. Complexity results for blocks-world planning. In *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991*, T.L. Dean and K. R. McKeown, eds, vol. 2, pp. 629–633. AAAI Press / The MIT Press, 1991.
[30] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, **26**, 191–246, 2006.
[31] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, Ithaca, 1962.
[32] J. Hoffmann, J. Porteous and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, **22**, 215–278, 2004.
[33] T. Hofmann, T. Viehmann, M. Gomaa, D. Habering, T. Niemueller, G. Lakemeyer and C. R. Team. Multi-agent goal reasoning with the clips executive in the robocup logistics league. In *ICAART (1)*, pp. 80–91, 2021.
[34] Xiao Huang, Biqing Fang, Hai Wan and Yongmei Liu. A general multi-agent epistemic planner based on higher-order belief change. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1093–1101. ijcai.org, 2017.
[35] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal and P. Stone. Task planning in robotics: An empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, **20**, 363–373, 2019.
[36] K. A. Jonsson, N. Muscettola, H. P. Morris and K. Rajan. Next generation remote agent planner. In *Artificial Intelligence, Robotics and Automation in Space*, vol. 440, p. 363, 1999.
[37] F. Kominis and H. Geffner. Beliefs in multiagent planning: From one agent to many. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*, pp. 147–155. AAAI Press, 2015.
[38] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, **16**, 83–94, 1963.
[39] V. Lifschitz. Answer set planning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 373–374. Springer, 1999.
[40] V. Lifschitz. *Answer Set Programming*. Springer, Berlin, 2019.
[41] N. Lipovetzky and H. Geffner. Best-first width search: Exploration and exploitation in classical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 3590–3596, San Francisco, California, USA, February 4-9, 2017.

[42] S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science*. *Robotics*, **7** eabm6074, 2022.
[43] T. Niemueller, T. Hofmann and G. Lakemeyer. Goal reasoning in the clips executive for integrated planning and execution. *Proceedings of the International Conference on Automated Planning and Scheduling*, **29**, 754–763 May, 2021.
[44] S. Richter and M. Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, **39**, 127–177, 2010.
[45] S. J. Russell and P. Norvig. *Artificial Intelligence - a Modern Approach*. Third International Edition, Pearson Education, 2010.
[46] V. Shivashankar, U. M. D. E. D. U. Ron Alford, U. Kuter and D. Nau. Hierarchical goal networks and goal-driven autonomy: Going where AI planning meets goal reasoning. *In Goal Reasoning: Papers from the ACS Workshop, page*, **95**, 2013.
[47] Davide Soldà, Francesco Fabiano and Agostino Dovier. Epistemic multiagent reasoning with collaborative robots. In R. Calegari, G. Ciatto and A. Omicini, editors, *Proceedings of the 37th Italian Conference on Computational Logic, Bologna, Italy, June 29 - July 1, 2022, Volume 3204 of CEUR Workshop Proceedings*, pages 32–46. CEUR-WS.org, 2022.
[48] T. C. Son, E. Pontelli, M. Balduccini and T. Schaub. *Answer Set Planning: A Survey* arXiv preprint arXiv:2202.05793, 2022.
[49] Tran Cao Son, Enrico Pontelli, Chitta Baral and Gregory Gelfond. Finitary S5-theories. In E. Fermé and J. Leite, editors, *Logics in Artificial Intelligence*, pages 239–252, Cham, 2014. Springer International Publishing.
[50] Stefano Tonetta. *ROBDT Robotic Digital Twin*. https://es.fbk.eu/index.php/projects/robdt/, September2021. Accessed: 2022-05-10.
[51] H. Van Ditmarsch, W. van Der Hoek and B. Kooi. *Dynamic Epistemic Logic*, vol. 337. Springer Science & Business Media, 2007.

Received 1 May 2023