Answer Set Programming and Planning with Knowledge and World-Altering Actions in Multiple Agent Domains

Enrico Pontelli[1], Tran Cao Son[1], Chitta Baral[2], and Gregory Gelfond[2]

[1] Department of Computer Science
New Mexico State University
epontell,tson@cs.nmsu.edu
[2] Department of Computer Science & Engineering
Arizona State University
chitta,ggelfond@asu.edu

Abstract. This paper discusses the planning problem in multi-agent domains, in which agents may execute not only world-altering actions, but also epistemic actions. The paper reviews the concepts of Kripke structures and update models, as proposed in the literature to model epistemic and ontic actions; it then discusses the use of *Answer Set Programming (ASP)* in representing and reasoning about the effects of actions on the world, the knowledge of agents, and planning. The paper introduces the $m\mathcal{A}_0$ language, an action language for multi-agent domains with epistemic and ontic actions, to demonstrate the proposed ASP model.

1 Introduction

The literature on multi-agent planning has grown at a fast pace in recent years. A large part of the literature deals with coordination between agents [5, 11, 7, 6, 8, 16] and their actions. While these issues are important, what is challenging, but less frequently addressed, is the issue of agents' knowledge about each other's knowledge and beliefs, and the manipulation of such knowledge/beliefs to achieve goals. Indeed, while logistics and coordination is important in multi-agent scenarios, such as those dealing with warfare, history provides myriad examples of battles where smaller armies have outsmarted better equipped ones, partly through the use of misinformation. Intelligence and counter-intelligence actions play important roles in such operations.

Thus, within a multi-agent setting, an important and difficult aspect is planning that involves the manipulation of the knowledge of agents, and not just about the physical world, but about each other's knowledge. In this paper, we take steps towards addressing these issues. In particular, we are interested in planning scenarios where agents reason about each others knowledge, perform actions to manipulate such knowledge, and develop plans that can guarantee awareness/ignorance of certain properties of the world by different agents to reach their own objectives. Consider the following example.

Example 1 (The Strongbox Domain). Agents A, B, and C are in a room. In the room there is a box containing a coin. It is common knowledge amongst them that:

- No one knows whether the coin is showing heads or tails.
- The box is locked and one needs a key to open it.

- Only agent A has the key of the box.
- To determine the face of the coin one can peek into the box, if the box is open.
- If one is looking at the box and someone peeks into it, he will be able to conclude that the agent who peeked knows which face of the coin is showing—but without knowing the state of the coin himself.
- Distracting an agent causes this agent not to look at the box.
- Signaling an agent to look causes that agent to look at the box.
- Announcing that the status of the coin will cause everyone to know this fact.

Suppose that agent A wishes to know which face of the coin is up, and that he would like agent B to become aware of the fact that he knows, while keeping agent C in the dark. Intuitively, agent A could achieve his goal by: (*i*) distracting C, keeping him from looking at the box; (*ii*) signaling B to look at the box; (*iii*) opening the box; and finally (*iv*) peeking into the box.

This simple scenario poses a number of challenges for research in multi-agent planning. The domain contains several classes of actions: *(1)* Actions that allow the agents to change the state of the world (e.g., open the box, signal/distract the agents); *(2)* Actions that change the knowledge of the agents (e.g., peek into the box, announce head/tail); *(3)* Actions that manipulate the beliefs of other agents (e.g., peek while other agents observe). In order for agent A to realize that steps *(i)–(iv)* will achieve his goal, he must be able to reason about the effects of actions:

- On the state of the world (e.g., opening the box causes the box to be open; distracting causes an agent to not look at the box); and
- On the knowledge of agents about her own knowledge (e.g., someone following her actions would know what she knows).

In this paper, we address the aforementioned problems by developing a high-level action language for representing and reasoning about actions in multi-agent domains. The semantics of an action theory is defined by the *answer sets* of a corresponding logic program. Our approach shows that *Answer Set Programming (ASP)* approaches which have been investigated for single-agent domains can be naturally generalized to multi-agent ones. The advantage of this approach stems from the fact that ASP can be used both as a specification and implementation language; an ASP encoding of an action theory can be used for various reasoning tasks, e.g., hypothetical reasoning, planning, and diagnosis. As such, it becomes an indispensable tool to explore epistemic action languages for multi-agent systems and validate the design of action theories.

2 Preliminaries

2.1 Answer Set Programming

Let us assume a collection of propositional variables $\mathcal{P}$. An answer set program (ASP) over $\mathcal{P}$ [10, 2] is a set of rules of the form: $a_0 \leftarrow a_1, \ldots, a_m, not\ \ a_{m+1}, \ldots, not\ \ a_n$, where $0 \leq m \leq n$, each a_i is an atom of $\mathcal{P}$[3], and not represents *negation-as-failure*. A naf-literal has the form $not\ \ a$, where a is an atom. Given a rule of this form, the

[3] A rule with variables is viewed as a shorthand for the set of its ground instances.

left and right hand sides are called the *head* and *body*, respectively. A rule may have either an empty head or an empty body, but not both. Rules with an empty head are called *constraints*—the empty head is implicitly assumed to represent false—while those with an empty body are known as *facts*.

A set of ground atoms X satisfies the body of a rule if $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$ and $\{a_1, \ldots, a_m\} \subseteq X$. A rule with a non-empty head is satisfied if either its body is not satisfied by X, or $a_0 \in X$. A constraint is *satisfied* by X if its body is not satisfied by X. Given a program Π and a set of ground atoms X, the *reduct* of Π w.r.t. X (denoted by Π^X) is the program obtained from the set of all ground instances of Π by:

1. Deleting all the rules that have a naf-literal $not\ \ a$ in the body where $a \in X$, and
2. Removing all naf-literals in the bodies of the remaining rules.

A set of ground atoms X is an *answer set* of a program Π if X is the subset-minimal set of atoms that satisfies all the rules in the program Π^X.

A program Π is said to be *consistent* if it has an answer set, and *inconsistent* otherwise. To make answer set programming easier, Niemelä et al. [13] introduced a new type of rule, called a *cardinality constraint rule*, where each atom in the rule can be a *choice atom*. A choice atom has the form $l\{b_1, \ldots, b_k\}u$, where each b_j is an atom, and l and u are integers such that $l \leq u$. Choice atoms can be also written as $l\{p(\bar{X}) : q(\bar{X})\}u$, where $\bar{X}$ is a set of variables—this is shorthand for the choice atom $l\{p(\bar{s}_1, \ldots, p(\bar{s}_k\}u$, where $\{\bar{s}_1, \ldots, \bar{s}_k\}$ are all the ground instances of $\bar{X}$ such that $q(\bar{X})$ is true. A set of atoms X satisfies a choice atom $l\{b_1, \ldots, b_k\}u$ if $l \leq |X \cap \{b_1, \ldots, b_k\}| \leq u$. The semantics of logic programs which contain such rules is given in [13].

The fact that a program can have multiple (or no) answer sets, encourages an alternative method of solving problems via logic programming [12, 13]. In this approach, we develop logic programs whose answer sets have a one-to-one correspondence with the solutions of the particular problem being modeled. Typically an ASP program consists of *(1)* Rules to enumerate the possible solutions of a problem as *candidate* answer sets; and *(2)* Constraints to eliminate answer sets not representing solutions of the problem.

2.2 Belief Formulae and Kripke Structures

Let us consider an environment with n agents $\mathcal{AG} = \{1, \ldots, n\}$. The state of the world may be described by a set $\mathcal{F}$ of propositional variables, called *fluents*. Following [9], we associate with each agent i a modal operator $\mathbf{B}_i$, and represent the beliefs of an agent as belief formulae in a logic extended by these operators:

- *Fluent formulae:* a fluent formula is a propositional formula built using the atomic formulae in $\mathcal{F}$ and the traditional propositional connectives $\vee$, $\rightarrow$, $\neg$, etc. A *fluent literal* is either a fluent atom $f \in \mathcal{F}$ or its negation $\neg f$.
- *Belief formulae:* a belief formula is a formula which has one of the following forms: *(1)* a fluent formula; *(2)* a formula of the form $\mathbf{B}_i\varphi$ where φ is a belief formula; *(3)* A formula of the form $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ or $\neg\varphi_1$ where φ_1 and φ_2 are belief formulae.

In addition, given a belief formula, φ, and a non-empty set $\alpha \subseteq \mathcal{AG}$, we call $\mathbf{E}_\alpha\varphi$ and $\mathbf{C}_\alpha\varphi$ *group formulae.* Furthermore, we use the shorthand form $\mathbf{C}\varphi$ to denote $\mathbf{C}_{\mathcal{AG}}\varphi$. In the following sections, we will simply use the term formula instead of belief formula. We denote with $\mathcal{L}_{\mathcal{AG}}$ the set of all formulae over $\mathcal{F}$ and $\mathcal{AG}$.

Definition 1 (Kripke Structure). A *Kripke structure* with respect to[4] an $\mathcal{F}$ and $\mathcal{AG}$ is a tuple $\langle S, \pi, \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, where S is a set of state symbols, π is a function that associates an interpretation of $\mathcal{F}$ to each element of S, and $\mathcal{B}_i \subseteq S \times S$ for $1 \leq i \leq n$. A *pointed Kripke structure* is defined as a pair (M, s) where $M = \langle S, \pi, \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$ is a Kripke structure and $s \in S$ (referred to as the *real state* of the world).

Given a Kripke structure, $M = \langle S, \pi, \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$, and a state symbol, $s \in S$, the satisfaction relation between belief formulae and a pointed Kripke structure (M, s) is defined as follows:

- $(M, s) \models \varphi$ if φ is a fluent formula and $\pi(s) \models \varphi$;
- $(M, s) \models \mathbf{B}_i \varphi$ if $\forall t \in S$ s.t. $(s, t) \in \mathcal{B}_i$, $(M, t) \models \varphi$;
- $(M, s) \models \neg\varphi$ if $(M, s) \not\models \varphi$;
- $(M, s) \models \varphi_1 \vee \varphi_2$ if $(M, s) \models \varphi_1$ or $(M, s) \models \varphi_2$;
- $(M, s) \models \mathbf{E}_\alpha \varphi$ if $(M, s) \models \mathbf{B}_i \varphi$ for every $i \in \alpha$.
- $(M, s) \models \mathbf{C}_\alpha \varphi$ if $(M, s) \models E^k_\alpha \varphi$ for every $k \geq 0$ where: $E^1_\alpha \varphi = E_\alpha \varphi$ and $E^{k+1}_\alpha = E_\alpha(E^k_\alpha \varphi)$.

We often view a Kripke structure M as a directed labeled graph, with S as its nodes and with an arc of the form (s, i, t) if and only if $(s, t) \in \mathcal{B}_i$. We use $M[S]$, $M[\pi]$, and $M[i]$, to denote the components S, π, and $\mathcal{B}_i$ of M, respectively.

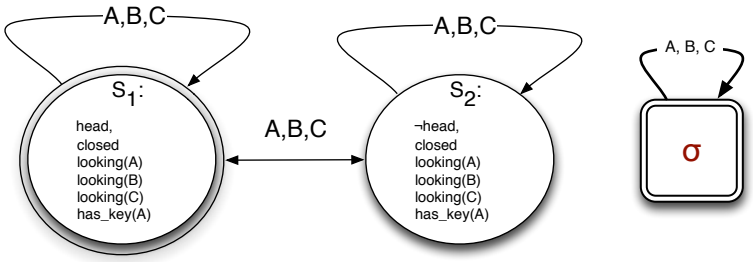

Fig. 1. An Example of a Pointed Kripke Structure and an Update Model

Consider a simplified version of the initial state in Example 1. None of the agents A, B, and C is aware of the state of the coin, and this is common knowledge. The box is closed. All the agents are aware that the box is closed, everyone is looking, and everyone knows that agent A has the key. Assume that the coin is showing heads. The knowledge of the agents together with the real world is captured by the pointed Kripke structure shown in Fig. 1. In the figure, a circle represents a state, and is labeled by its name and interpretation. Arcs denote the belief relations captured by the structure. Lastly, the double circle represents the real physical state of the world.

Intuitively, a Kripke structure denotes the possible worlds envisioned by the agents, with multiple worlds denoting uncertainty and the presence of different beliefs. The relation $(s_1, s_2) \in \mathcal{B}_i$ captures the notion that agent i when at the world s_1 can not distinguish between the state described by the world s_1 and the one described by the

[4] When it is clear from the context we may not explicitly mention the associated $\mathcal{F}$ and $\mathcal{AG}$ of a Kripke structure.

world s_2. Thus, $M[\pi](s_1) \models \varphi$ and $M[\pi](s_2) \models \neg\varphi$, indicates that agent i is uncertain about the truth of φ.

We are often interested in Kripke structures with certain properties; for example, a Kripke structure $\langle S, \pi, \mathcal{B}_1, \ldots, \mathcal{B}_n \rangle$ is **S5** if, for each agent i and formulae φ and ψ the following axioms hold: (**K**) $(\mathbf{B}_i\varphi \wedge \mathbf{B}_i(\varphi \Rightarrow \psi)) \Rightarrow \mathbf{B}_i\psi$; (**T**) $\mathbf{B}_i\psi \Rightarrow \psi$; (**4**) $\mathbf{B}_i\psi \Rightarrow \mathbf{B}_i\mathbf{B}_i\psi$; and (**5**) $\neg\mathbf{B}_i\psi \Rightarrow \mathbf{B}_i\neg\mathbf{B}_i\psi$.

2.3 Update Models

Program models are used to represent action occurrences, using structures similar to pointed Kripke structures; they describe the effects of an action on states using an update operator. The original proposal [1] deals with sensing and announcement actions, later extended to world-altering (a.k.a. ontic) actions [15] (and called update models).

An $\mathcal{L}_{\mathcal{AG}}$-substitution is a set $\{p_1 \rightarrow \varphi_1, \ldots, p_n \rightarrow \varphi_n\}$, where each p_i is a distinct fluent and each φ_i is a formula in $\mathcal{L}_{\mathcal{AG}}$. We will assume that for each $p \in \mathcal{F} \setminus \{p_1, \ldots, p_n\}$, the substitution contains $p \rightarrow p$. The set of all $\mathcal{L}_{\mathcal{AG}}$-substitutions is denoted with $SUB_{\mathcal{L}_{\mathcal{AG}}}$.

Definition 2. *An* update model $\mathbf{\Sigma}$ *is a tuple* $(\Sigma, \{R_i \mid i \in \mathcal{AG}\}, pre, sub)$ *where*

- Σ *is a set, whose elements are called* events*;*
- $R_i \subseteq \Sigma \times \Sigma$ *for each* $i \in \mathcal{AG}$*;*
- $pre : \Sigma \rightarrow \mathcal{L}_{\mathcal{AG}}$ *is a function mapping each event* $a \in \Sigma$ *to a formula in* $\mathcal{L}_{\mathcal{AG}}$*;*
- $sub : \Sigma \rightarrow SUB_{\mathcal{L}_{\mathcal{AG}}}$.

A update instance ω *is a pair* $(\mathbf{\Sigma}, e)$ *where* $\mathbf{\Sigma}$ *is an update model* $(\Sigma, \{R_i \mid i \in \mathcal{AG}\}, pre, sub)$ *and* e*, referred to as a* designated event*, is a member in* Σ.

Intuitively, an update model represents different views of an action occurrence which are associated with the observability of agents. Each view is represented by an event in Σ. The designated event is the one that agents who are aware of the action occurrence will observe. The relation R_i describes agent i's uncertainty on action execution—i.e., if $(\sigma, \tau) \in R_i$ and event σ is performed, then agent i may believe that event τ is executed instead. pre defines the action precondition and sub specifies the changes of fluent values after the execution of an action. Update models and instances are graphically represented similarly to (pointed) Kripke structures. The update instance on the right of Fig. 1 is $(\mathbf{\Sigma_1}, \sigma)$ where $\mathbf{\Sigma_1} = (\{\sigma\}, \{R_A, R_B, R_C\}, pre_1, sub_1)$ and $R_A = R_B = R_C = \{(\sigma, \sigma)\}$, $pre_1(\sigma) = true$, and $sub_1(\sigma) = \{closed \rightarrow false\}$.

Definition 3. *Given a Kripke structure* M *and an update model* $\mathbf{\Sigma} = (\Sigma, \{R_i \mid i \in \mathcal{AG}\}, pre, sub)$*, the* update operator *defines a Kripke structures* $M' = M \otimes \mathbf{\Sigma}$*, where*

- $M'[S] = \{(s, \tau) \mid s \in M[S], \tau \in \Sigma, (M, s) \models pre(\tau)\}$,
- $((s, \tau), (s', \tau')) \in M'[i]$ *iff* $(s, s') \in M[i]$ *and* $(\tau, \tau') \in R_i$*, and*
- *For each* $f \in \mathcal{F}$*,* $M'[\pi]((s, \tau)) \models f$ *iff* $f \rightarrow \varphi \in sub$ *and* $(M, s) \models \varphi$.

Intuitively, the Kripke structure M' is obtained from the component-wise cross-product of the old structure M and the update model $\mathbf{\Sigma}$, by keeping only those new states (s, τ) s.t. (M, s) satisfies the action precondition.

Example 2. Continuing with the previous examples, let us compute $M' = M_1 \otimes \mathbf{\Sigma_1}$:

- $M'[S]$=$\{(s_1, \sigma), (s_2, \sigma)\}$. Let u=(s_1, σ) and v=(s_2, σ).
- $M'[A] = M'[C] = \{(u, u), (v, v)\}$ and $M'[B] = \{(u, u), (v, v), (u, v), (v, u)\}$.
- $M'[\pi](u) = \{head, \neg closed\}$ and $M'[\pi](v) = \{\neg head, \neg closed\}$. □

An update template is a pair $(\mathbf{\Sigma}, \Gamma)$ where $\mathbf{\Sigma}$ is an update model with the set of events Σ and $\Gamma \subseteq \Sigma$. The update of a state (M, s) given a update template $(\mathbf{\Sigma}, \Gamma)$ is a set of states, denoted by $(M, s) \otimes (\mathbf{\Sigma}, \Gamma)$, where for each (M', s')∈$(M, s) \otimes (\mathbf{\Sigma}, \Gamma)$, it holds that M'=$M \otimes \mathbf{\Sigma}$ and $s' = (s, \tau)$ where $\tau \in \Gamma$ and $s' \in M'[S]$.

Observe that the discussion in this section focuses on the changes caused by an update model $\mathbf{\Sigma}$ (resp. update template $(\mathbf{\Sigma}, \Gamma)$) on a state (M, s). It does not place any requirement (e.g. **S5**) on the Kripke structure M of the state. In a dynamic environment, agents might be unaware of action occurrences and thus could have false beliefs. For instance, the agent C (Example 1) would still believe that agent A does not know the status of the coin after A executes the action sequence (i)-(iv). As such, it will be interesting to investigate the properties of the Kripke structures after an update by an update model. We leave this as an important future research topic as it is outside the scope of this paper.

3 Basic ASP Encodings

In this section, we will present the ASP rules encoding the update operator. The encoding is general and does not assume any properties of the underlying Kripke structures.

Encoding Kripke Structures in ASP: Each fluent and belief formula of interest (i.e., used in the domain specification) φ is represented by a corresponding term $\tau(\varphi)$, defined in the natural recursive manner:

- if φ is true, then $\tau(\varphi) = top$
- if φ is a fluent, then $\tau(f) = f$
- if φ is the literal $\neg f$, then $\tau(\neg f) = neg(f)$
- if φ is the formula $\varphi_1 \vee \varphi_2$, then $\tau(\varphi) = or(\tau(\varphi_1), \tau(\varphi_2))$
- if φ has the form $\mathbf{B}_i\varphi_1$, then $\tau(\varphi) = b(i, \tau(\varphi_1))$

In order to represent a Kripke structure, we need to describe its three components: the state symbols, their associated interpretations, and the accessibility relations. We will assume that each pointed Kripke structure has a name—represented by a term; we will use *pointedKS*(t) to assert that t is the name of a pointed Kripke structure. The components of each pointed Kripke structure (M, s) A pointed Kripke structure (M, s), named $t_{(M,s)}$, is described by atoms of the form:

- *state*$(u, t_{(M,s)})$ denoting the fact that $u \in M[S]$;
- *real*$(s, t_{(M,s)})$ denoting that s is the real state of the world in the Kripke structure;
- $r(i, u, v, t_{(M,s)})$ denoting the fact that $(u, v) \in M[i]$;
- $holds(\tau(\ell), u, t_{(M,s)})$ denoting the fact that $M[\pi](u) \models \ell$ for a fluent literal ℓ.

The following constraints are useful to guarantee core properties of a Kripke structure: for each $f \in \mathcal{F}$, $u \in M[S]$

$$\begin{array}{l} \leftarrow\ holds(\tau(f), u, t_{(M,s)}), holds(\tau(\neg f), u, t_{(M,s)}) \\ \leftarrow\ not\ holds(\tau(f), u, t_{(M,s)}), not\ holds(\tau(\neg f), u, t_{(M,s)}) \end{array}$$

The predicate $holds(\tau(\varphi), s, t_{(M,s)})$ expresses the truth value of a formula φ with respect to a pointed Kripke structure (M, s) (i.e., $(M, s) \models \varphi$), and is defined recursively on the structure of φ. For example:

- if φ is a literal, then its definition comes from the pointed Kripke structure;
- if φ is of the form $\varphi_1 \vee \varphi_2$ then:

$$\begin{array}{l} holds(or(\tau(\varphi_1), \tau(\varphi_2)), S, T)\ \leftarrow\ holds(\tau(\varphi_1), S, T) \\ holds(or(\tau(\varphi_1), \tau(\varphi_2)), S, T)\ \leftarrow\ holds(\tau(\varphi_2), S, T) \end{array}$$

- if φ is of the form $\mathbf{B}_i\varphi_1$ then:

$$\begin{array}{l} n\_holds(b(i, \tau(\varphi)), S, T)\ \leftarrow\ r(i, S, S_1, T), not\ holds(\tau(\varphi_1), S_1, T) \\ holds(b(i, \tau(\varphi)), S, T)\ \leftarrow\ not\ n\_holds(b(i, \tau(\varphi)), S, T) \end{array}$$

- if φ is of the form $E_\alpha\varphi_1$ then:

$$\begin{array}{ll} n\_holds(\tau(\varphi), S, T)\ \leftarrow\ not\ holds(b(i, \tau(\varphi_1)), S, T) & \text{for each } i \in \alpha \\ holds(\tau(\varphi), S, T)\ \leftarrow\ not\ n\_holds(\tau(\varphi), S, T) & \end{array}$$

- if φ is of the form $C_\alpha\varphi_1$ then:

$$\begin{array}{ll} connect(S, S_1, \alpha, T)\ \leftarrow\ r(i, S, S_1, T) & \text{for each } i \in \alpha \\ connect(S_1, S_2, \alpha, T)\ \leftarrow\ connect(S_1, S_3, \alpha, T), connect(S_3, S_2, \alpha, T) & \\ n\_holds(\tau(\varphi), S, T)\ \leftarrow\ connect(S, S_1, \alpha, T), not\ holds(\tau(\varphi_1), S_1, T) & \\ holds(\tau(\varphi), S, T)\ \leftarrow\ not\ n\_holds(\tau(\varphi), S, T) & \end{array}$$

  Observe that the above encoding utilizes the following property: $(M, s) \models C_\alpha\varphi$ iff $(M, s') \models \varphi$ for every state $s' \in M[S]$ such that there exist a sequence of agent $i_1, \ldots, i_k \in \alpha$ and a sequence of states $s_1, \ldots, s_k$ such that $s_1 = s$, $s_k = s'$, and $s_{i+1} \in M[i]$ for $1 \leq i < k$.

For a formula φ, let Π_φ^T denote the set of rules defining $holds(\tau(\varphi), S, T)$ (including the rules for the sub-formulae of φ). For a pointed Kripke structure (M, s), let $\kappa\rho^t(M, s)$ be the following set of facts:

- $real(s, t)$;
- *state*(u, t) for each $u \in M[S]$;
- $r(a, u, v, t)$ if $(u, v) \in M[a]$;
- $holds(\ell, u, t)$ if $(M, u) \models \ell$, for each fluent literal ℓ.

Proposition 1. *$\Pi(M, s, t) = \kappa\rho^t(M, s) \cup \Pi_\varphi^t$ has a unique answer set S and $(M, u) \models \varphi$ iff $holds(\tau(\varphi), u, t) \in S$.*

Encoding Update Models and Update Templates: The encoding of an update model and update template follows an analogous structure as a Kripke structure; given an update template $(\mathbf{\Sigma}, \Gamma)$ where $\mathbf{\Sigma} = (\Sigma, \{R_i \mid i \in \mathcal{AG}\}, pre, sub)$, we introduce a fact of the form *updateT*(t) to indicate that t is the term naming the update template. The description of $(\mathbf{\Sigma}, \Gamma)$ contains the rules $\upsilon\mu^t(\mathbf{\Sigma}, \Gamma)$:

- $actual(e, t)$ for each $e \in \Gamma$;
- $event(e, t)$ for each $e \in \Sigma$;
- $acc(a, e, e', t)$ if $(e, e') \in R_a$;
- $pre(e, \tau(\varphi), t)$ if $pre(e) = \varphi$;
- $sub(e, \tau(f), \tau(\varphi), t)$ and $sub(e, \tau(\neg f), \tau(\neg\varphi), t)$ if $(f \to \varphi) \in sub(e)$ and $\varphi \not\equiv f$.

In the case of an update instance, $\upsilon\mu^t(\mathbf{\Sigma}, \Gamma)$ will contain a single fact of $actual$.

Encoding Update Operators in ASP: The outcome of the update operation between a pointed Kripke structure and an update instance is a new pointed Kripke structure; the rules encoding the update operation will thus define the relations describing the components of the new pointed Kripke structure.[5] Let us introduce the fact $occ(t_K, t_\Sigma)$ to identify the application of an update instance to a given pointed Kripke structure. The following rules are used to determine the pointed Kripke structure resulting from the application of an update mode.

The identification of the new pointed Kripke structure comes from the rule

$$pointedKS(app(KS, UT)) \leftarrow pointedKS(KS), updateT(UT), occ(KS, UT)$$

The states of the new Kripke structure are defined as follows:

$$\begin{aligned} state(st(S, E), app(KS, UT)) \leftarrow\ & occ(KS, UT), state(S, KS), event(E, UT), \\ & pre(E, F, UT), holds(F, S, KS) \end{aligned}$$

The accessibility relation of the new pointed Kripke structure is a direct consequence of the accessibility relations of the Kripke structure and the update model:

$$\begin{aligned} r(Ag, st(S1, E1), st(S2, E2), app(KS, UT)) \leftarrow\ & occ(KS, UT), \\ & state(st(S1, E1), app(KS, UT)), \\ & state(st(S2, E2), app(KS, UT)), \\ & r(Ag, S1, S2, KS), acc(Ag, E1, E2, UT) \end{aligned}$$

The real state of the world is defined by

$$\begin{aligned} real(st(S, E), app(KS, UT)) \leftarrow\ & occ(KS, UT), state(st(S, E), app(KS, UT)), \\ & real(S, KS), actual(E, UT) \end{aligned}$$

Finally, we need to determine the interpretations associated to the various states:

$$\begin{aligned} & complement(F, neg(F)) && \leftarrow fluent(F) \\ & complement(neg(F), F) && \leftarrow fluent(F) \\ & holds(L, st(S, E), app(KS, UT)) && \leftarrow occ(KS, UT), state(st(S, E), app(KS, UT)), literal(L), \\ & && \quad sub(E, L, Form, UT), holds(Form, S, KS) \\ & holds(L, st(S, E), app(KS, UT)) && \leftarrow occ(KS, UT), state(st(S, E), app(KS, UT)), literal(L), \\ & && \quad complement(L, L1), holds(L, S, KS), \\ & && \quad not\ holds(L1, st(S, E), app(KS, UT)) \end{aligned}$$

Let us denote this set of rules with $\alpha\pi$. To prepare for the following proposition, let us introduce some notations. Given an update model $\mathbf{\Sigma} = (\Sigma, \{R_i \mid i \in \mathcal{AG}\}, pre, sub)$, let us define $\Phi(\mathbf{\Sigma}) = \{\varphi \mid \exists e \in \Sigma.\ (f \to \varphi) \in sub(e)\} \cup \{pre(e) \mid e \in \Sigma\}$.

[5] The code has been simplified for readability—e.g., by removing some domain predicates.

Proposition 2. *Let (M, s) be a pointed Kripke structure and let $(\mathbf{\Sigma}, \Gamma)$ be an update template. Let t_1 be the term denoting the name given to (M, s) and t_2 the term denoting the name given to $(\mathbf{\Sigma}, \Gamma)$. Let*

$$\Pi((M,s),t_1,(\mathbf{\Sigma},\Gamma),t_2) = \alpha\pi \cup \kappa\rho^{t_1}(M,s) \cup \upsilon\mu^{t_2}(\mathbf{\Sigma},\Gamma) \cup \bigcup_{\varphi \in \Phi(\mathbf{\Sigma})} \Pi_{\varphi}^{t_1} \cup \{occ(t_1,t_2)\}.$$

Let S be an answer set of $\Pi((M, s), t_1, (\mathbf{\Sigma}, \Gamma), t_2)$. Then

- $(u, e) \in (M \otimes \mathbf{\Sigma})[S]$ *iff* $state(st(u, e), app(t_1, t_2)) \in S$
- $((u, e), (u', e')) \in (M \otimes \mathbf{\Sigma})[i]$ *iff* $r(i, st(u, e), st(u', e'), app(t_1, t_2)) \in S$
- $(M \otimes \mathbf{\Sigma}, (u, e)) \models \psi$ *iff* $holds(\tau(\psi), st(u, e), app(t_1, t_2)) \in S$
- $(M \otimes \mathbf{\Sigma}, (u, e)) \in (M, s) \otimes (\mathbf{\Sigma}, \Gamma)$ *iff* $real(st(u, e), app(t_1, t_2)) \in S$

4 An Application in Multi-agent Planning

In this section we instantiate the generic principles illustrated above to the case of a multi-agent action language—the language $m\mathcal{A}_0$ [14]. We review the syntax of $m\mathcal{A}_0$ and illustrate the specific encoding of some of its actions using ASP.

4.1 Syntax

The action language $m\mathcal{A}_0$ is based on the same logic language introduced in Section 2.2. We extend the language signature with a set $\mathcal{A}$ of *actions*.

For the Strongbox domain, the set $\mathcal{AG} = \{A, B, C\}$, the set of fluents contains the fluents $head$ (the coin is head's up), $closed$ (the box is closed), $looking(i)$ (agent i is looking at the box), and $key(i)$ (agent i has a key), and the set of actions include actions like $open(i)$ (agent i opens the box), $peek(i)$ (agent i peeks into the box), and $announce(i, \varphi)$ (agent i announces that the formula φ is true).

Each action is associated to exactly one executability law of the form

$$\textbf{executable } a \textbf{ if } \psi \tag{1}$$

indicating that the action $a \in \mathcal{AG}$ can be executed only if the formula ψ is satisfied. We distinguish three types of actions in $m\mathcal{A}_0$, i.e., $\mathcal{A} = \mathcal{A}_o \uplus \mathcal{A}_s \uplus \mathcal{A}_a$:

- *Ontic* actions are used to modify properties of the world; they are described by statements of the form

  $$a \textbf{ causes } \ell \textbf{ if } \psi \tag{2}$$

  indicating that the action $a \in \mathcal{A}_o$ will make the literal ℓ true if the action is executed and the formula ψ is satisfied. For example, the action $open(i)$ is described by:

  $$open(i) \textbf{ causes } \neg closed \textbf{ if } true$$

- *Sensing* actions enable agents to observe unknown properties of the world, refining their knowledge. Each sensing action $a \in \mathcal{A}_s$ is described by a statement

  $$a \textbf{ determines } f \tag{3}$$

  where $f \in \mathcal{F}$ is the property being observed. For example, the $peek(i)$ action is described by: $peek(i)$ **determines** $head$.

- *Announcement* actions are used by an agent to share knowledge with other agents; each announcement action $a \in \mathcal{A}_a$ is described by a statement of the form

$$a \textbf{ announces } \varphi \tag{4}$$

where φ is a formula describing the knowledge being shared. For example, the action $announce(i, \neg head)$ is described by:

$$announce(i, \neg head) \textbf{ announces } \neg head$$

Another distinct feature of $m\mathcal{A}_0$ is action observability; the effects of each action on a pointed Kripke structure is dependent on which agents can observe the execution of the action and its effects. This is a dynamic property which is handled explicitly in the domain description through statements of the form:

$$ag \textbf{ observes } a \textbf{ if } \varphi \tag{5}$$

$$ag \textbf{ partially\_observes } a \textbf{ if } \varphi \tag{6}$$

where $a \in \mathcal{A}$ and φ is a formula. For example (for $X, Y \in \mathcal{AG}$):

$$X \textbf{ observes } peek(X) \textbf{ if } true$$

$$X \textbf{ partially\_observes } peek(Y) \textbf{ if } X \neq Y \wedge looking(X)$$

A domain description $\mathcal{D}$ is a collection of statements of the type (1)-(6). For simplicity, we assume that each action has one statement of type (1); we also assume that each sensing and announcement action is described by one statement of type (3) or (4).

The semantics of $m\mathcal{A}_0$ has been introduced in [3, 4] and it relies on the definition of a transition function $\Phi_D(a, B)$ which determines the result of executing an action a in a set of pointed Kripke structures B (a.k.a. a *belief state*)—as a new belief state.

4.2 Modeling $m\mathcal{A}_0$ Actions in ASP

Executability. We envision actions being executed in a state of the world/knowledge described by a pointed Kripke structure. Actions can be executed only if their executability condition is met; if a is an action and **executable** a **if** ψ is its executability law, then we add to $\epsilon\varsigma^t$ the constraint

$$\leftarrow occ(t, a), real(s, t), not\ holds(\tau(\psi), s, t)$$

Observability. Let us introduce three ASP predicates that will capture the observability properties of an action: obs indicating that an agent is fully knowledgeable of the effects of the action, $pobs$ indicating that the agent is only aware of the action execution but not its effects, and obv denoting that the agent is oblivious about the action execution. The rules defining these predicates compose the logic program $o\beta^t$. For each action a, if i **observes** a **if** φ is in $\mathcal{D}$, then $o\beta^t$ contains

$$obs(i, a, t) \leftarrow real(s, t), occ(t, a), holds(\tau(\varphi), s, t)$$

If $\mathcal{D}$ contains i **partially_observes** a **if** φ, then $o\beta^t$ contains

$$pobs(i, a, t) \leftarrow real(s, t), occ(t, a), holds(\tau(\varphi), s, t)$$

Finally, we need to add to $o\beta^t$ the rules

$$obv(i, a, t) \leftarrow occ(t, a), not\ obs(i, a, t), not\ pobs(i, a, t)$$

For the sake of the discussion in the following subsections, given an action a and a pointed Kripke structure (M, s), we define

$$\begin{aligned}
\alpha(a, M, s) &= \{i \mid (i \textbf{ observes } a \textbf{ if } \varphi) \in \mathcal{D}, (M, s) \models \varphi\} \\
\beta(a, M, s) &= \{i \mid (i \textbf{ observes } a \textbf{ if } \varphi) \in \mathcal{D}, (M, s) \models \varphi\} \\
\gamma(a, M, s) &= \mathcal{AG} \setminus (\alpha(a, M, s) \cup \beta(a, M, s))
\end{aligned}$$

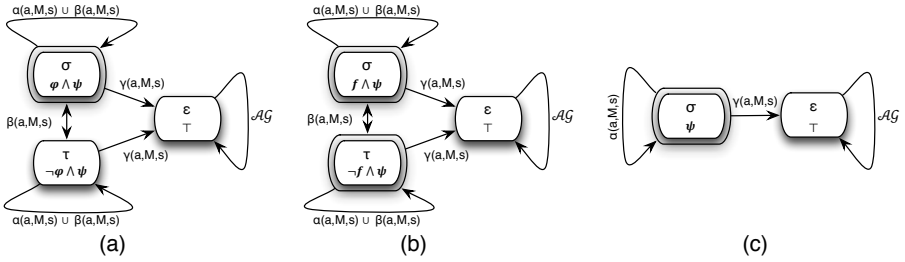

Fig. 2. Graphical Representations of the Update Templates

Announcements Actions. Let us consider an announcement action a described by the law a **announces** φ. This action can be captured by an update template which is schematically summarized in Figure 2(a) (the double circle denotes the real event, the formulae inside the circles represent the precondition pre). For this update model, $sub(e) = \emptyset$ for each $e \in \Sigma$. The intuition behind this update model is as follows: σ represents the actual announcement, which will make all agents in $\alpha(a, M, s)$ aware that φ is true; the agents in $\beta(a, M, s)$ will partially observe the action, thus learning that all agents in $\alpha(a, M, s)$ are aware of the value of φ, but unable to know whether φ or $\neg\varphi$ is true—and thus they cannot distinguish between the even σ and the event τ (that announces $\neg\varphi$. The agents in $\gamma(a, M, s)$ are unaware of the action execution and thus do not see any change of beliefs (this is encoded by the event ϵ).

Let us denote this update model by $a_\varphi(T) = (\mathbf{\Sigma}_a(T), \Gamma_a(T))$, where T is the identification of the pointed Kripke structure in which the update model is performed and a is the e action. This behavior is coded in ASP by the following collection of facts:

- The actual event is described by $actual(\sigma, a_\varphi(T))$
- The collection of events is $\{event(\sigma, a_\varphi(T)), event(\tau, a_\varphi(T)), event(\epsilon, a_\varphi(T))\}$.
- The accessibility relations among events are described as follows:

$$\begin{aligned}
acc(Agent, S_1, S_1, a_\varphi(T)) &\leftarrow S_1 \in \{\tau, \sigma, \epsilon\}, obs(Agent, a, T) \\
acc(Agent, S_1, S_1, a_\varphi(T)) &\leftarrow S_1 \in \{\tau, \sigma, \epsilon\}, pobs(Agent, a, T) \\
acc(Agent, \epsilon, \epsilon, a_\varphi(T)) &\leftarrow obv(Agent, a, T) \\
acc(Agent, \sigma, \tau, a_\varphi(T)) &\leftarrow pobs(Agent, a, T) \\
acc(Agent, \tau, \sigma, a_\varphi(T)) &\leftarrow pobs(Agent, a, T) \\
acc(Agent, S_1, \epsilon, a_\varphi(T)) &\leftarrow S_1 \in \{\sigma, \tau\}, obv(Agent, a, T)
\end{aligned}$$

Sensing Actions. Let us consider a sensing action s described by the law s **determines** f. This action can be captured by an update template which is schematically summarized in Figure 2(b). For this update model, $sub(e) = \emptyset$ for each $e \in \Sigma$. The intuition is similar to the case of announcement actions; σ represents the the sensing action detecting

that the fluent f is true, while τ is the sensing action detecting the fluent f being false; the action ϵ is viewed by the agents that are oblivious of the action execution.

Let us denote this update model by $s_f(T) = (\mathbf{\Sigma}_s(T), \Gamma_s(T))$, where T is the identification of the pointed Kripke structure in which the update model is performed and s is the name of the action. This behavior is coded in ASP by the following facts:

- The actual event is described by *actual*$(\sigma, s_f(T))$
- The collection of events is $\{event(\sigma, s_f(T)), event(\tau, s_f(T)), event(\epsilon, s_f(T))\}$.
- The accessibility relations among events are described as follows:

$$\begin{array}{ll} acc(Agent, S_1, S_1, s_f(T)) & \leftarrow\ S_1 \in \{\tau, \sigma, \epsilon\}, obs(Agent, s, T) \\ acc(Agent, S_1, S_1, s_f(T)) & \leftarrow\ S_1 \in \{\tau, \sigma, \epsilon\}, pobs(Agent, s, T) \\ acc(Agent, \epsilon, \epsilon, s_f(T)) & \leftarrow\ obv(Agent, s, T) \\ acc(Agent, \sigma, \tau, s_f(T)) & \leftarrow\ pobs(Agent, s, T) \\ acc(Agent, \tau, \sigma, s_f(T)) & \leftarrow\ pobs(Agent, s, T) \\ acc(Agent, S_1, \epsilon, s_f(T)) & \leftarrow\ S_1 \in \{\sigma, \tau\}, obv(Agent, s, T) \end{array}$$

Ontic Actions. Let us consider an ontic action o described by the set of laws o **causes** ℓ_1 **if** $\varphi_1, \ldots, o$ **causes** ℓ_n **if** φ_n. For the sake of simplicity, we assume here that the various ℓ_i are simply literals (i.e., a fluent or its negation). This action can be captured by an update template which is schematically summarized in Figure 2(c). For this update model, $sub(\epsilon) = \emptyset$ while

$$sub(\sigma) \ =\ \{f \rightarrow \varphi \vee f \mid (o \textbf{ causes } \ f \textbf{ if } \ \varphi) \in D\} \cup \{f \rightarrow \neg\varphi \wedge f \mid (o \textbf{ causes } \ \neg f \textbf{ if } \ \varphi) \in D\}$$

Intuitively, the event σ denotes the actual world-changing event, seen by all agents witnessing the action execution, while ϵ is the event witnessed by all other agents.

Let us denote this update model by $o(T) = (\mathbf{\Sigma}_o(T), \Gamma_o(T))$, where T is the identification of the pointed Kripke structure in which the update model is performed and o is the name of the action. This behavior is coded in ASP by the following facts:

- The actual event is described by *actual*$(\sigma, o(T))$
- The collection of events is $\{event(\sigma, o(T)), event(\epsilon, o(T))\}$.
- The accessibility relations among events are described as follows:

$$\begin{array}{ll} acc(Agent, S_1, S_1, o(T)) & \leftarrow\ S_1 \in \{\sigma, \epsilon\}, obs(Agent, o, T) \\ acc(Agent, \epsilon, \epsilon, o(T)) & \leftarrow\ obv(Agent, o, T) \\ acc(Agent, \sigma, \epsilon, o(T)) & \leftarrow\ obv(Agent, o, T) \end{array}$$

- The substitution is described by all facts of the form ($1 \leq i \leq n$)

$$\begin{array}{ll} sub(\sigma, f_i, \tau(\varphi_i \vee f_i), o(T)) & \ell_i \textit{ is the fluent } f_i \\ sub(\sigma, f_i, \tau(\neg\varphi_i \wedge f_i), o(T)) & \ell_i \textit{ is the literal } \neg f_i \end{array}$$

To conclude this section, it is possible to show that, for each type of action described, the following property holds.

Proposition 3. *Let a be an action described by the update model $\eta(T)$ and let (M, s) be a pointed Kripke structure. Let us consider the program $\Pi((M, s), t_1, \eta(M, s), t_2)$. Then for each formula ψ the following holds: for each pointed Kripke structure $(M', s') \in \Phi_D(a, \{(M, s)\})$, $(M', s') \models \psi$ iff there exists an answer set A of $\Pi((M, s), t_1, \eta(M, s), t_2)$ such that $real(x, v) \in A$ and $holds(\tau(\psi), x, v) \in A$.*

4.3 ASP for Reasoning in $m\mathcal{A}_0$

Various forms of reasoning can be realized using the set of rules described earlier. We illustrate some of these next.

Projection. Given a sequence of actions $a_1, \ldots, a_k$ and a pointed Kripke structure (M, s), let us denote with $\Phi_D^*([a_1, \ldots, a_k], (M, s))$ the set of pointed Kripke structures derived from the execution of the given sequence of actions starting in the pointed Kripke structure (M, s). Let us denote with $ut(a_i)$ the update model of the action a_i. Let us generalize the definition of Π given earlier as

$$
\begin{array}{lcl}
\Pi^0((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) & = & \alpha\pi \cup \kappa\rho^{t_0}(M,s) \\
R^0((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) & = & t_0 \\
\Pi^j((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) & = & \Pi^{j-1}((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) \cup \\
 & & \upsilon\mu^{t_j}(ut(a_j)) \cup \bigcup_{\varphi \in \Phi(ut(a_j))} \Pi_\varphi^{t_j-1} \cup \\
 & & \{occ(R^{j-1}((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k), t_j)\} \\
R^j((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) & = & app(R^{j-1}((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k), t_j) \\
\Pi((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k) & = & \Pi^k((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k)
\end{array}
$$

A *projection query* has the form φ **after** $a_1, \ldots, a_k$; the query is entailed w.r.t. (M, s) if for each $(M', s') \in \Phi^*([a_1, \ldots, a_k], (M, s))$ we have that $(M', s') \models \varphi$.

A program to answer the projection query can be realized by adding to $\Pi((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k)$ the rules

$$
\begin{array}{l}
\leftarrow\ real(s, R^k((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k)), \\
\quad not\ holds(\tau(\varphi), s, R^k((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k))
\end{array}
$$

Let us denote with $Prj_\varphi((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k)$ this program.

Proposition 4. *The program $Prj_\varphi((M,s), t_0, ut(a_1), t_1, \ldots, ut(a_k), t_k)$ has an answer set A iff the projection query φ* **after** *$a_1, \ldots, a_k$ is entailed w.r.t. (M, s).*

Planning. The generalization of projection queries to planning is simple. Let us consider the case of planning with a finite horizon k and let us consider a formula φ representing the planning goal. The rules to generate occurrences of actions

$$
\begin{array}{l}
current(t_0, 0). \\
1\{occ(M, A) : action(A)\}1\ \leftarrow\ current(M, T). \\
current(app(M, A), T + 1)\ \leftarrow\ occ(M, A), current(M, T)
\end{array}
$$

The validation of the goal can be described by the constraint

$$\leftarrow current(M, k), real(S, M), holds(\tau(\varphi), S, M).$$

Let us consider the program $Plan_\varphi((M,s), t_0, k)$ containing the following rules:

- The rules $\alpha\pi$;
- The rules $\kappa\rho^{t_0}(M,s)$;
- The rules $\upsilon\mu^{t_i}(ut(a_i))$ for each action a_i with update model $ut(a_i)$;

- The rules $\bigcup_{\phi \in \Phi(ut(a_i))} \Pi_{\varphi}^{T}$ for each action a_i; and
- The described rules for action occurrence generation and for goal validation.

Proposition 5. *Let (M, s) be a pointed Kripke structure, an horizon k, and a formula φ. Then, the program $Plan_{\varphi}(M, s, t_0, k)$ has an answer set iff there is a sequence of actions $a_1, \ldots, a_k$ such that (M, s) entails φ* **after** $a_1, \ldots, a_k$.

4.4 From Theory to Practice

Towards a More Practical ASP Encoding. In this section, we will discuss practical issues that we have to address in order to derive a workable implementation from the ideas presented in the previous sections. We will assume that the initial pointed Kripke structure is given and is encoded following the description in Section 3. The standard components of an ASP program for plan generation are not changed with respect to the description in the previous section. Since we are working with one pointed Kripke structure at a time and the execution of an action in a pointed Kripke structure results in a pointed Kripke structure, the naming of Kripke structures can be simplified by using the time step of the planning problem (Subsection 4.3).

Because the answer set solver `clingo` accepts λ-restricted programs we can identify formulae by themselves, providing that they are defined by the predicate $formula/1$. The following code defines the types of formulae that we will be considered in the program. To make the program λ-restricted, we consider only formulae that contain at most m operators and connectives.

```
2 {formula(F), length(F, 1)}:- fluent(F).
2 {formula(neg(F)), length(neg(F), 1)}:- fluent(F).
2 {formula(knows(A,F)), length(knows(A,F), L+1)}:-
       formula(F), length(F, L), L < m, agent(A).
2 {formula(and(F1,F2)), length(and(F1,F2), L1+L2)}:-
       formula(F1), formula(F2), length(F1, L1),
       length(F2, L2), L1+L2 < m.
2 {formula(or(F1,F2)), length(or(F1,F2), L1+L2)}:-
       formula(F1), formula(F2), length(F1, L1),
       length(F2, L2), L1+L2 < m.
2 {formula(neg(F)), length(neg(F), L+1)}:-
      formula(F), length(F, L), L < m.
```

The encoding of the updates caused by the update models on pointed Kripke structures can also be simplified by observing that there are at most three events (σ, τ, and ϵ, see Subsection 4.2) in each update model. Since the update models corresponding to the actions are known, it is possible to specialize the code that implements the update operations w.r.t these known update models. For example, let us consider the action $open(X)$ from the Strongbox domain. Since this is an ontic action, we deal with two events σ and ϵ. State symbols $o(S, E)$ for the next pointed Kripke structure are computed as follows:

```
state(o(S, sigma),T+1) :-
   occ(open(X), T),
```

```
state(o(S, epsilon), T+1) :-
   occ(open(X), T),
```

```
ontic(open(X)),time(T), T < n,      ontic(open(X)), time(T),
state(S, T), connected(S, T),       T < n, state(S, T),
holds(has_key(X), S, T).            connected(S, T).
```

The first rule states that `o(S,sigma)` is a new state symbol of the pointed Kripke structure at time step $T+1$ if S is a state symbol of the structure at time step T and the action $open(X)$ is performed at time T. Note that n denotes the length of the desired plan.
The accessibility relation is defined as follows:

```
r(X,o(S1,sigma),o(S2,sigma),T+1):- time(T), T < n,
    state(o(S1, sigma), T+1), state(o(S2, sigma), T+1),
    obs(X, T),  r(X, S1, S2, T).
r(X, o(S1, epsilon), o(S2, epsilon), T+1):- time(T), T < n,
    state(o(S1, epsilon), T+1), state(o(S2, epsilon), T+1),
    agent(X), r(X, S1, S2, T).
r(X, o(S1, sigma), o(S2, epsilon), T+1):- time(T), T < n,
    state(o(S1, sigma), T+1), state(o(S2, epsilon), T+1),
    obv(X, T), r(X, S1, S2, T).
```

The encoding of the interpretation can also be simplified as follows:

```
holds(F, o(S, sigma), T+1):- time(T), T < n,
    state(o(S, sigma), T+1),
    ontic(open(X)), occ(open(X), T), holds(F, S, T).
holds(opened, o(S, sigma), T+1):- time(T), T < n,
    state(o(S, sigma), T+1), ontic(open(X)), occ(open(X), T).
holds(F, o(S, epsilon), T+1) :- time(T), T < n,
    state(o(S, epsilon), T+1), holds(F, S, T).
```

The first two rules define the interpretation for the new state `o(S, sigma)`. The first one encodes the inertial axiom and the second encodes a positive effect of $open(a)$. The last rule encodes the interpretation for the state `o(S, epsilon)`, which is the same as that of S.

The encoding can be used for two purposes: projection and planning. To verify the executability of an action sequence, we can add the action occurrences as facts and verify whether an answer set exists. To compute a plan, we need to add the goal as a constraint; e.g., for the goal of having a to know the state of the coin and both b and c be oblivious, we can use the following code:

```
goal :- real(S, n), holds(b(a, head), S, n),
  holds(b(b, neg(or(b(a, head), b(a, neg(head))))),S,n),
  holds(b(c, neg(or(b(a, head), b(a, neg(head))))),S,n),
:- not goal.
```

Some Experimental Results. We have experimented the above mentioned encoding with the Strongbox domain used earlier and a more complicated domain, called the Prison domain. In the latter, we have agents a, b, and c, where agent a, a friend of c (a double agent in the organization of b), is in the custody of an hostile agent b. a needs c's

help to escape. a needs to take the key without b knowing about it, and make c aware that he has the key. He cannot do that while b is watching. c can only be aware of a's action if he is watching a. c can make a aware of his presence by making noise (e.g., shouting). a could also look around for c.

All experiments have been performed a Mac OS machine with an Intel Core i7 2.8 GHz processor and 16 GB memory. The codes used in the experiments are available for download. A detailed discussion on the Prison domain can be found in [4]. The ASP solver used for the experiments is `clingo` version 3.0.3 (based on `clasp` version 1.3.5).

Our initial experiments with the code reveal that, even with $m = 3$ (the maximal length of formula), `clingo` uses most of the time for grounding the formulae and the related rules. For example, `clingo` was not able to return a plan for the Strongbox domain goal described earlier (i.e., a is aware of the state of the coin while b and c are not) within 1 hour. To this end, we simplify the encoding using a preprocessing step, which extracts all the formulae that are used in the goal and in the action descriptions, and specializes the encoding to focus exclusively on this set of formulae. With this optimization, `clingo` was able to generate plans in just a few minutes.

In the Strongbox domain, with the initial pointed Kripke structure of Figure 1, the solver was able to generate a total of 12 plans of length 4 (in 25 seconds) for the goal mentioned earlier. The first group of plans consists of two actions for a—aimed at distracting b and c ($distract(a, b)$ and $distract(a, c)$)—an action to open the box ($open(a)$), and then an action to peek into the box ($peek(a)$). The first three actions can be in any order, leading to 8 plans. A sample plan is:

```
occ(distract(a,c),0); occ(open(a),1); occ(distract(a,b),2);
                                      occ(peek(a),3)
```

The other group of plans represents interesting aspects of multi-agent planning. A plan in this group contains a distract action between b and c ($distract(b, c)$ or $distract(c, b)$), a distract action between a and the executor of the previous action, and the $open(a)$ and $peek(a)$ actions. This group contains four plans, e.g.,

```
occ(distract(b,c),0); occ(open(a),1); occ(distract(a,b),2);
                                      occ(peek(a),3)
```

In the Prison domain, we used an initial pointed Kripke structure where a, b, and c know that a is in the custody of b, b is watching a, both a and b are present, and a does not have a key. a and b do not know whether c is present. In the real state of the world, c is present. We experiment with the goal

```
goal :- real(S, n), holds(b(a, has_key(a)),S,n),
  holds(b(c, has_key(a)),S,n),
  holds(b(b, neg(has_key(a))),S,n).
:- not goal.
```

The goal is related to a: obtaining the key, making c aware of this while keeping b in the dark. This requires that c announces his presence ($shouting(c)$) or a looks around for c, a distracts b ($distract(a, b)$) and signals c ($signal(a, c)$), and a takes the key ($get\_key(a)$). The solver was able to return the first plan within one minute. It found all possible plans in 76 seconds (the problem admits two possible plans).

To test the scalability of the approach, we introduce a new action called $free(c, a, b)$, whose precondition is a conjunction of $B_a(has\_key(a) \wedge present(c))$, $B_c(has\_key(a))$, and $B_{a,c}(\neg looking(b))$ and whose effect is $\neg custody(a, b)$. The minimal plan that achieves the goal $\neg custody(a, b)$ has length 5. The solver is able to generate the first two plans in 627 seconds:

```
occ(shout(c),0); occ(signal(a,c),1); occ(distract(a,b),2);
                 occ(get_key(a),3); occ(free(c,a,b),4)

occ(lookAround(a,c),0); occ(signal(a,c),1); occ(distract(a,b),2);
                 occ(get_key(a),3); occ(free(c,a,b),4)
```

We aborted the computation of all plans for this goal after 30 minutes without being able to identify any additional solutions.

5 Conclusion and Future Work

In this paper, we demonstrated the use of ASP as a technology to model and encode multi-agent planning domains where agents can perform both ontic as well as epistemic actions. The use of ASP enables us to provide a clear semantics to this type of action languages—providing a very high level and executable encoding of actions and their effects on the pointed Kripke structures used to represent the state of the world. ASP allows us also to validate models—supporting the use of different forms of reasoning, such as projection and planning. We illustrated this possibility in two sample domains, with encouraging results.

The work presented in this paper is aimed at laying the foundations for a large breadth investigation in high level action languages for epistemic and ontic multi-agent domains. In particular, there are several aspects in the design of a suitable action language that need to be considered—such as how to embed in the action languages more refined views of visibility of action execution (e.g., allow agents to view agents viewing actions being executed) and how to handle actions related to dishonesty (e.g., telling lies). Another aspect that needs to be considered is the task of planning and/or projection with respect to some initial specification since the present work assumes that the initial state, in the form of a pointed Kripke structure, is given. The experimental steps presented in this paper have also highlighted strengths and weaknesses of ASP in this type of applications. In particular, it is clear that grounding remains a great concern—indeed, in order to obtain an usable program it became necessary to devise an encoding in ASP that restricts the set of formulae to be analyzed—something that would require a smart translator from action theories to ASP. These aspects will be the focus of our future work.

Acknowledgement

Chitta Baral acknowledges the support by ONR-MURI and IARPA. Enrico Pontelli and Tran Cao Son acknowledge the support by NSF-IIS 0812267 grant.

References

1. A. Baltag and L. Moss. Logics for Epistemic Programs. *Synthese*, 2004.
2. C. Baral. *Knowledge Representation, reasoning, and declarative problem solving with Answer sets*. Cambridge University Press, Cambridge, MA, 2003.
3. C. Baral and G. Gelfond. On Representing Actions in Multi-Agent Domains. In *Logic Programming, Knowledge Representation, and Non-Monotonic Reasoning*, pages 213–230. Springer Verlag, 2011.
4. C. Baral, G. Gelfond, E. Pontelli, and T.C. Son. An Action Language for Multi-agent Domains, Technical Report, New Mexico State University, 2011.
5. J.S. Cox and E.H. Durfee. An Efficient Algorithm for Multi-agent Plan Coordination. In *4th International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 828–835. ACM, 2005.
6. M. de Weerdt, A. ter Mors, and C. Witteveen. Multi-agent Planning: An Introduction to Planning and Coordination. In *Handouts of the European Agent Summer School*, pages 1–32, 2005.
7. M. de Weerdt and C. Witteveen. Multi-agent Planning: Problem Properties that Matter. In *Proceedings of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, number SS-06-04, pages 155–156. AAAI, 2006.
8. E.H. Durfee. Distributed Problem Solving and Planning. In *Multi-agent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press, 1999.
9. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. The MIT press, 1995.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings of the Fifth International Conf. and Symp.*, pages 1070–1080, 1988.
11. J. S. Cox and E. H. Durfee and T. Bartold. A Distributed Framework for Solving the Multi-agent Plan Coordination Problem. In *AAMAS*, pages 821–827. ACM Press, 2005.
12. V. Marek and M. Truszczyński. Stable Models as an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, Springer Verlag, pages 375–398, 1999.
13. I. Niemelä. Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
14. E. Pontelli, T.C. Son, C. Baral, and G. Gelfond. Logic Programming for Finding Models in the Logics of Knowledge and its Applications: A Case Study. *Theory and Practice of Logic Programming*, 10(4-6):675–690, 2010.
15. J. van Benthem, J. van Eijck, and B.P. Kooi. Logics of communication and change. *Inf. Comput.*, 204(11):1620–1662, 2006.
16. W. van der Hoek and M. Wooldridge. Tractable Multi-agent Planning for Epistemic Goals. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 1167–1174. ACM, 2002.