$\mathcal{APIA}$: An Architecture for Policy-Aware Intentional Agents

John Meyer
Miami University, Ohio, USA
meyerjm@miamioh.edu

Daniela Inclezan
Miami University, Ohio, USA
inclezd@miamioh.edu

This paper introduces the $\mathcal{APIA}$ architecture for policy-aware intentional agents. These agents, acting in changing environments, are driven by intentions and yet abide by domain-relevant policies. This work leverages the $\mathcal{AIA}$ architecture for intention-driven intelligent agents by Blount, Gelfond, and Balduccini. It expands $\mathcal{AIA}$ with notions of policy compliance for authorization and obligation policies specified in the language $\mathcal{AOPL}$ by Gelfond and Lobo. $\mathcal{APIA}$ introduces various agent behavior modes, corresponding to different levels of adherence to policies. $\mathcal{APIA}$ reasoning tasks are reduced to computing answer sets using the CLINGO solver and its Python API.

1 Introduction

This paper introduces $\mathcal{APIA}$,[1] an architecture for intentional agents that are aware of policies and abide by them. It leverages and bridges together research by Blount, Gelfond, and Balduccini [5, 6] on intention-driven agents ($\mathcal{AIA}$[2]), and work by Gelfond and Lobo [12] on authorization and obligation policy languages ($\mathcal{AOPL}$[3]). Both $\mathcal{AIA}$ and $\mathcal{AOPL}$ are expressed in action languages [11] that have a seamless translation into logic programs [10], and are implemented in Answer Set Programming (ASP) [14].

With the current rise in autonomous systems the question arises of how to best construct agents that are capable of acting in a changing environment in pursuit of their goals, while also abiding by domain-relevant policies. For instance, we would want a self-driving car to not only take us to our destination but also do so while respecting the law and cultural conventions. This work is a first step in this direction. As in Blount *et al.*'s work, we focus on agents:

- whose environment (including actions and their effects) and mental states can be represented by a transition diagram. Physical properties of the environment and the agent's mental states form the nodes of this transition diagram. Arcs from one state to another are labeled by actions that may cause these transitions.
- who are capable of making correct observations, remembering the domain history, and correctly recording the results of their attempts to perform actions;
- who are *normally* capable of observing the occurrences of exogenous actions; and
- whose knowledge bases may contain some pre-computed plans for achieving certain goals, which we call *activities*, while agents can compute other plans on demand.

Additionally, these agents possess specifications of authorization and obligation policies and have access to reasoning mechanisms for evaluating the policy compliance of their actions.

[1] $\mathcal{APIA}$ stands for "Architecture for Policy-aware Intentional Agents."
[2] $\mathcal{AIA}$ stands for "Architecture for Intentional Agents."
[3] $\mathcal{AOPL}$ stands for "Authorization and Obligation Policy Language."

Thus, in our work, we extend the $\mathcal{AIA}$ architecture introduced by Blount *et al.* [5, 6] with the notion of policy compliance. $\mathcal{AIA}$ agents are Belief-Desire-Intention (BDI) agents [16] who are driven by goals, have intentions that persist, and can operate with pre-computed activities. However, they are oblivious to the different nuances of authorization and obligation policies (e.g., actions whose execution is permitted vs. not permitted), and policy compliance. These intentional agents have only two behavioral modes: either ignore policies altogether (when policies are not represented by any means in the agent's knowledge base) or blindly obey policies that disallow certain actions from being performed, even if this may be detrimental (when policies are represented as actions that are impossible to be executed). Instead, we introduce a wider range of possible behaviors that may be set by an agent's controller (e.g., prefer plans that are *certainly* policy-compliant to others that are only *possibly* policy-compliant, but if no plans of the former type exist, accept plans of the latter kind).

In formalizing the different policy compliance behavior modes, we rely on the language $\mathcal{AOPL}$ [12] for specifying authorization and obligation policies, and priorities between such policies. While Gelfond and Lobo specify reasoning algorithms for determining the degree of policy compliance for a sequence of actions, this is done from the perspective of a third-person observer analyzing the actions of *all* agents *after* they already occurred. Instead, our work focuses on an agent making decisions about its own future actions. As a result, various courses of action may be compared to determine the most policy-compliant one according to the behavior mode set by the agent's controller. Moreover, our research addresses interactions between authorization and obligation policies that were not discussed in the work on $\mathcal{AOPL}$.

There have been several other attempts to enable agents to reason over various kinds of policies. However, these involve reasoning over access control policies only [7, 17, 1] and a few utilize ASP as a reasoning tool for this purpose [3, 4]. Access control policies are more restrictive than the kinds of policies an agent using $\mathcal{AOPL}$ can reason over. The work that is closest to our goal is the PDC-agent by Liao, Huang, and Gao [13]. The PDC architecture extends the BDI architecture with a policy and contract-aware methodology the authors call BGI-PDC logic. A PDC-agent is an event-driven multi-component framework which allows for controlled and coordinated behavior among independent cooperative agents. Liao *et al.* use policies to control agent behavior, and contracts as a mechanism to coordinate actions between agents. This architecture was later extended to support reasoning over social norms (the NPDC-agent) [15]. The PDC-agent architecture is defined as a 7-tuple of the following components: (Event Treating Engine, Belief Update, Contract Engine, Policy Engine, Goal Maintenance, Plan Engine, Plan Library) [13]. A major distinction of the PDC-agent agent architecture is that it supports coordination among multiple agents. This is beyond the scope of our work. Both $\mathcal{AIA}$ and $\mathcal{APIA}$ focus on an agent with individual goals. Expanding these architectures into multi-agent frameworks by introducing communication acts is still part of future work. However, knowledge about the changing environment is expressed in the PDC-agent architecture in terms of a Domain Conceptualization Language (DCL) [8] and a Concept Instance Pattern (CIP). While DCL and CIP can represent plans (which are analogous to activities in the $\mathcal{AIA}$ architecture), there is no support for expressing the direct or indirect effects of an action. This is a disadvantage in comparison to action-language-based architectures since plans have to be pre-computed and the goals that they accomplish must be annotated according to the agent's designer's intuition. Since action languages only require a description of the effects of individual actions (and plans consisting of all combinations of actions can be automatically computed), there is significantly less work for a human designer when working with $\mathcal{APIA}$ than the PDC-agent architecture.

Thus, our proposed $\mathcal{APIA}$ architecture is, to the best of our knowledge, the only intentional agent architecture that is capable to model compliance with complex authorization and obligation policies, *while allowing agents to come up with policy-compliant activities on the fly.*

The major contributions presented in this paper work are as follows:

1. Create a bridge between research on intentional agents and policy compliance, thus producing a *policy-aware intentional agent* architecture $\mathcal{APIA}$.
2. Introduce various agent behavior modes with respect to compliance with authorization and obligation policies.
3. Introduce mechanisms to check the consistency of a policy containing both authorization and obligation policies and reason over the interactions between these two.
4. Implement $\mathcal{APIA}$ in CLINGO (version 5.4.1)[4] while leveraging CLINGO's Python API.[5] (As a by-product, $\mathcal{AIA}$ was also updated to this CLINGO version).

2 Background

In this section, we briefly present the $\mathcal{AIA}$ architecture and $\mathcal{AOPL}$ language, which form the two pillars of our work. We direct the unfamiliar reader to outside resources on Answer Set Programming [10, 14] and action language $\mathcal{AL}$ [9], which are also relevant to our research.

2.1 $\mathcal{AIA}$: Architecture for Intentional Agents

The *Architecture for Intentional Agents*, $\mathcal{AIA}$ [6], builds upon the Observe-Think-Act control loop of the AAA architecture[6] [2] and extends it in a couple of directions. First, $\mathcal{AIA}$ adds the possibility for action failure: the agent *attempts* to perform an action during its control loop but may find that it is unable to do so. In this case, the action is deemed *non-executable*. Second, $\mathcal{AIA}$ addresses a limitation of AAA in which plans are not persisted across iterations of the control loop. In $\mathcal{AIA}$, agents pursue goals by persisting in their intentions to execute action plans known to satisfy these goals (i.e., activities). Activities are represented via a set of statics [5, 6]:

$$\begin{aligned}\{&activity(M),\ length(M,L),\ goal(M,G),\\ &component(M,1,C_1),\ component(M,2,C_2),\ \ldots,\ component(M,L,C_L)\ \}\end{aligned}$$

where M is a unique identifier for the activity; $C_1,C_2,\ldots,C_L$ are the $1^{st},2^{nd},\ldots,L^{th}$ *components* of the activity; and G is the goal that $[C_1,C_2,\ldots,C_L]$ achieves. Some activities are pre-computed and stored in the agent's knowledge base, while the rest can be generated on demand.

In addition to fluents and actions describing the agent's environment, $\mathcal{AIA}$ introduces *mental fluents* to keep track of the agent's progress in the currently intended activity and towards its desired goal. Mental fluents are updated through *mental actions*. The Theory of Intentions is a collection of axioms that maintain an agent's mental state. Elements of the agent's mental state include the currently selected goal G, stored in the $active\_goal\,(G)$ inertial fluent, and the current planned activity, stored in the $status(M,k)$ inertial fluent. When either a goal is selected or an activity is planned, they are said to be *intended*. Mental action $start(M)$ initiates the agent's intention to execute activity M and $stop(M)$ terminates it. Though most actions are executed by the agent itself, some must be executed by the agent's controller.

[4]https://potassco.org/clingo/

[5]https://potassco.org/clingo/python-api/5.4/

[6]AAA stands for "Autonomous Agent Architecture."

The exogenous mental action $select(G)$ causes the agent to intend to achieve goal G while $abandon(G)$ causes the agent to cease its intent to achieve goal G.

An agent in the $\mathcal{AIA}$ architecture performs the following loop:

1. Interpret observations.
2. Find intended action A.
3. Attempt to perform A; record this attempt in history. (1)
4. Observe the world; record observations in history.
5. Repeat (i.e. go to step 1).

2.2 $\mathcal{AOPL}$: Authorization and Obligation Policies in Dynamic Systems

In real-world applications, an autonomous agent may be required to follow certain rules or ethical constraints, and may be penalized when acting in violation of them. Thus, it is necessary to discuss policies for agent behavior and a formalism with which agents can deduce the compliance of their actions. Gelfond and Lobo [12] introduce the Authorization and Obligation Policy Language $\mathcal{AOPL}$ for policy specification. An *authorization* policy is a set of conditions that denote whether an agent's action is permitted or not. An *obligation* policy describes what an agent must do or must not do. $\mathcal{AOPL}$ works in conjunction with a dynamic system description written in an action language such as $\mathcal{AL}$. An agent's policy is the subset of the trajectories in the domain's transition diagram that are desired by the agent's controller.

Policies of $\mathcal{AOPL}$ are specified using predicates *permitted* for authorization policies, *obl* for obligation policies, and static laws similar to those from action language $\mathcal{AL}$:

$$\begin{array}{rll} permitted(a) & \textbf{if} & cond \\ \neg permitted(a) & \textbf{if} & cond \\ obl(h) & \textbf{if} & cond \\ \neg obl(h) & \textbf{if} & cond \end{array}$$

where a is an action; h is a happening (i.e., an action or its negation[7]); and *cond* is a, possibly empty, conjunction of fluents, actions, or their negations. In addition to these strict policy statements, $\mathcal{AOPL}$ supports defeasible statements and priorities between them as in:

$$\begin{array}{lcll} d : \textbf{normally} & permitted(a) & \textbf{if} & cond \\ d : \textbf{normally} & \neg permitted(a) & \textbf{if} & cond \\ d : \textbf{normally} & obl(h) & \textbf{if} & cond \\ d : \textbf{normally} & \neg obl(h) & \textbf{if} & cond \\ & prefer(d_i, d_j) & & \end{array} \tag{2}$$

Gelfond and Lobo define *policy compliance* separately for authorizations vs. obligations:

Definition 1 (Policy Compliance – adapted from Gelfond and Lobo [12]) Authorization: *A set A of actions occurring at a transition system state σ is strongly compliant with a policy P if all $a \in A$ are known to be permitted at σ using rules in P. A is non-compliant with P if any of the actions are known to be not permitted at σ using rules in P. Otherwise, if P is unclear and does not specify whether an action $a \in A$ is permitted or not at σ, then the set of actions is weakly compliant.*

[7] If $obl(\neg a)$ is true, then the agent must not execute a in the current state.

Obligation: *A set of actions A occurring at state σ is* compliant *with an obligation policy P if $a \in A$ whenever $obl(a)$ can be derived from the rules of P and $a \notin A$ whenever $obl(\neg a)$ can be derived from P at σ. Otherwise, it is* non-compliant.

Note that a set of actions A can be strongly, weakly, or non-compliant with an authorization policy, but A can only be compliant or non-compliant with an obligation policy. Given an $\mathcal{AOPL}$ policy (with authorization and obligation policy statements), A is *strongly compliant* if it is strongly compliant with its authorization policy and compliant with its obligation policy. Likewise for *weak compliance* and *non-compliance*. Computing policy compliance is reduced to the problem of finding answer sets of a logic program obtained by translating the policy rules into ASP. In this translation, predicates *obl* and *permitted* are extended to include an extra argument I standing for the time step, while $lp(x)$ denotes the ASP transformation of x, where x can be a rule, an action literal, a fluent literal, or a condition.

$\mathcal{AOPL}$ does not discuss interactions between authorization and obligation policies on the same action, does not define compliance in terms of obligation policies for a trajectory in the dynamic system, and does not compare the degree of compliance of two trajectories. All of these aspects are relevant when modeling a policy-aware intentional agent and are addressed in our work.

3 $\mathcal{APIA}$ Architecture

We can now introduce our $\mathcal{APIA}$ architecture for policy-aware intentional agents. We focus on two main aspects of $\mathcal{APIA}$: reframing $\mathcal{AOPL}$ to fit an agent-centered architecture, and the encoding of different policy compliance behavior modes of an $\mathcal{APIA}$ agent.

3.1 Re-envisioning $\mathcal{AOPL}$ Policies in an Agent-Centered Architecture

Gelfond and Lobo [12] conceived $\mathcal{AOPL}$ as a means to evaluate policy compliance in a dynamic system. This differs from $\mathcal{AIA}$ in the following ways:

- $\mathcal{AOPL}$ evaluates trajectories in a domain's transition diagram from a global perspective whereas $\mathcal{AIA}$ distinguishes between agent actions and exogenous actions, and chooses which agent actions to attempt next.
- $\mathcal{AOPL}$ evaluates histories at "the end of the day" whereas the $\mathcal{AIA}$ architecture, while still reasoning over past actions in its diagnosis mode, places an emphasis on planning future actions to achieve a future desired state.

These differences prevent $\mathcal{AOPL}$ policies from interoperating with the $\mathcal{AIA}$ agent architecture out of the box. To address the first issue, we constrain $\mathcal{AOPL}$ policies to describe only agent actions in our $\mathcal{APIA}$ architecture. For the second issue, we adjust our policy compliance rules such that only future actions affect policy compliance. Since our focus is on planning, in $\mathcal{APIA}$ past actions are always considered "compliant" although they might not have been at the time. For an agent that previously had no choice but a non-compliant action, this allows the agent to conceive of "turning a new leaf" and seeking policy-compliant actions in the future.

Also, $\mathcal{AOPL}$ does not include specification on how authorization policy statements interact with obligation policy statements. For example, consider the following $\mathcal{AOPL}$ policy:

$$\begin{gathered} permitted(a) \\ obl(\neg a) \end{gathered}$$

which is contradictory, since the agent is permitted to perform action a but at the same time is obligated to refrain from it. Appealing to common sense, if an agent is obligated to refrain from an action, one would conclude that the action is not permitted. Likewise, it makes sense to say that, if an agent is obligated to do an action, then it must be permitted. Thus, we take these intuitions and create the following non-contradiction ASP axioms, in which we use literals obl and $permitted$ expanded to include a new argument I representing the time step:

$$\begin{array}{ll} \leftarrow & agent\_action(A),\ obl(A,I),\ \neg permitted(A,I). \\ \leftarrow & agent\_action(A),\ obl(neg(A),I),\ permitted(A,I). \end{array} \tag{3}$$

These enforce that, at the very least, the authorization and obligation policies do not contradict each other, while allowing for defeasible policies to work appropriately.

We also extend the translation of defeasible policy statements. Suppose we have the following:

$$\begin{array}{c} \textbf{normally}\ permitted(a) \\ obl(\neg a)\ \ \textbf{if}\ \ cond \end{array} \tag{4}$$

Using Gelfond and Lobo's approach [12], the corresponding ASP translation would be:

$$\begin{array}{l} permitted(a,I) \leftarrow \textbf{not}\ \neg permitted(a,I). \\ obl(neg(a),I) \leftarrow lp(cond). \end{array}$$

where $lp(cond)$ and $obl(neg(a),I)$ represent the logic programming encoding of $cond$ and $obl(\neg a)$, respectively. Based on policy (4), both $permitted(a,I)$ and $obl(neg(a),I)$ would be true at a time step I when $cond$ is met. This violates the non-contradiction axioms in (3). So, we replace the translation of the defeasible statement in (4) with the following encoding:

$$permitted(a,I) \quad \leftarrow \quad \textbf{not}\ \neg permitted(a,I),\ \ \textbf{not}\ obl(neg(a),I).$$

This allows the presence of $obl(\neg a)$ to be an exceptional case to the defeasible rule.

In general, we propose translating the different types of defeasible statements in (2) as follows, respectively, where predicate ab facilitates dealing with possible additional (weak) exceptions:

$$\begin{array}{rcl} permitted(a,I) & \leftarrow & lp(cond),\ \textbf{not}\ ab(d,I),\ \textbf{not}\ \neg permitted(a,I),\ \textbf{not}\ obl(neg(a),I). \\ \neg permitted(a,I) & \leftarrow & lp(cond),\ \textbf{not}\ ab(d,I),\ \textbf{not}\ permitted(a,I),\ \textbf{not}\ obl(a,I). \\ obl(a,I) & \leftarrow & lp(cond),\ \textbf{not}\ ab(d,I),\ \textbf{not}\ \neg obl(a,I),\ \textbf{not}\ \neg permitted(a,I). \\ obl(neg(a),I) & \leftarrow & lp(cond),\ \textbf{not}\ ab(d,I),\ \textbf{not}\ obl(neg(a),I),\ \textbf{not}\ permitted(a,I). \end{array}$$

3.2 Policy-Aware Agent Behavior

The $\mathcal{AIA}$ architecture, which is the underlying basis of $\mathcal{APIA}$, introduces *mental* fluents and actions in addition to physical ones. In $\mathcal{APIA}$, we additionally introduce *policy* fluents and actions needed to reason over policy compliance (see Table 1). The new policy action descriptions encode the effects of future agent actions on policy compliance and provide means for the control loop to deem non-compliant activities futile and execute compliant ones in their place.

For example, the dynamic causal laws in (5) define inertial policy fluents $auth\_compliance(weak)$ and $auth\_compliance(strong)$ according to the definitions for strong and weak authorization policy compliance of $\mathcal{AOPL}$ seen in Definition 1:

$$\begin{array}{lllll} a & \textbf{causes} & \neg auth\_compliance(strong) & \textbf{if} & \textbf{not}\ permitted(a) \\ a & \textbf{causes} & \neg auth\_compliance(weak) & \textbf{if} & \neg permitted(a) \end{array} \tag{5}$$

Table 1: List of Policy Fluents and Actions in the $\mathcal{APIA}$ Architecture

Fluents		**Actions**
Inertial:	$auth\_compliance(strong)$	$ignore\_not\_permitted(a)$
	$auth\_compliance(weak)$	$ignore\_neg\_permitted(a)$
	$obl\_compliant(do\_action)$	$ignore\_obl(a)$
	$obl\_compliant(refrain\_from\_action)$	$ignore\_obl(neg(a))$
Defined:	$policy\_compliant(f)$, for every physical fluent f	for every physical action a

These rules are defined for every physical action a of the transition system. Should an action a occur where $permitted(a)$ is not known to be true, then the scenario ceases to be strongly compliant (i.e., it becomes weakly compliant). Since $auth\_compliance(strong)$ cannot be made true again by any action, the rest of the scenario remains weakly compliant by inertia. Likewise, should an action a occur where $permitted(a)$ is false, then the scenario ceases to be weakly compliant (i.e., it becomes non-compliant) and remains in this state by inertia.

For every physical fluent f, we introduce a new defined policy fluent $policy\_compliant(f)$. This allows us to reuse the $\mathcal{AIA}$ control loop shown in (1) as is. When the agent controller wants to specify that the agent should achieve goal f in a policy-compliant manner, the controller simply has to initiate action

$$select\_goal(policy\_compliant(f))$$

instead of the original $select\_goal(f)$. The policy fluent $policy\_compliant(f)$ is true iff f is true and $auth\_compliance(l)$ is true, for some minimum compliance threshold l set by the controller. Thus, when $policy\_compliant(f)$ is an agent's goal, activities below l-compliance are deemed as futile and the agent works to achieve fluent f subject l-compliance.

Authorization Policies and Agent Behavior. To allow for cases when the threshold l set by the controller is not the maximum possible level of compliance or, in the future, cases when an agent deliberately chooses to act without l-compliance, we add policy actions $ignore\_not\_permitted(a)$ and $ignore\_neg\_permitted(a)$, where a is a physical action. By executing these actions concurrently with a, our agent ignores a's effect at that time step on weak compliance or non-compliance, respectively. This enables our agent to look for activities with a lower level of compliance if no activities that achieve f are strongly-compliant. One can imagine that this capability can be used to model multiple agent behaviors, based on what the minimum and maximum requirements of adherence to their authorization policy are. We have parameterized the agent's behavior as seen in Table 2, and introduced names for these possible agent behaviors.

	Require weak	**Prefer weak over non-compl.**	**Ok with non-compl.**
Require strong	Paranoid	(Invalid)	(Invalid)
Prefer strong over weak	Cautious	Best effort	(Invalid)
Ok with weak	Subordinate	Subordinate when possible	Utilitarian

Table 2: $\mathcal{APIA}$ Authorization Policy Modes

One behavior mode is for the agent to strictly adhere to its authorization policy such that it never chooses to perform $ignore\_neg\_permitted(a)$. This causes all non-compliant actions to indirectly cause $policy\_compliant(f)$ to be false, if they are executed. Hence, only activities with weakly or strongly compliant actions are considered. Since this mode never dares to become non-compliant, it is called *subordinate*.

A similar behavior mode causes the agent to never perform $ignore\_not\_permitted(a)$. This causes all weak and non-compliant actions to indirectly cause $policy\_compliant(f)$ to be false when executed. Hence, only activities with strongly compliant actions are considered. Since weakly compliant actions are actions for which the policy compliance is unknown, this mode is called *paranoid* as it treats weakly compliant actions as if they were non-compliant.

Another behavior mode allows unrestricted access to the two actions, $ignore\_neg\_permitted(a)$ and $ignore\_not\_permitted(a)$. This mode is called *utilitarian* because it reduces the behavior of $\mathcal{APIA}$ to that of $\mathcal{AIA}$, where policies are not considered at all.

An interesting feature of the $ignore\_neg\_permitted(a)$ and $ignore\_not\_permitted(a)$ actions is the ability to optimize compliance. Using preference statements in ASP, we can require the control loop to minimize the use of these two policy actions. Hence, if it is possible to execute an activity that is strongly compliant, the agent will prefer it over a weakly or non-compliant one (since the use of these actions is required to allow $policy\_compliant(f)$ to be true). Under this condition, $ignore\_not\_permitted(a)$ and $ignore\_neg\_permitted(a)$ are only used when it is impossible to achieve the fluent f in a strongly or weakly compliant manner, respectively.

The combination of compliance optimization with the first three behavior modes allows for more possible configurations. For example, adding optimization to the *subordinate* option makes a *cautious* mode. In this mode, the agent will try to mimic the behavior of the *paranoid* mode (all strongly compliant actions), but ultimately it will reduce to *subordinate* (all weakly compliant actions) in the worst case. Likewise, adding optimization to the *utilitarian* mode adds two options: *best effort* and *subordinate when possible*. *Best effort* prefers strong compliance over weak compliance and weak compliance over non-compliance, but ultimately permits non-compliance when no better alternatives exist. *Subordinate when possible* prefers weak compliance over non-compliance but does not optimize from weak compliance to strong compliance.

A new feature of this approach to optimization is the ability to optimize within the weakly and non-compliant categories. Consider two weakly compliant activities, 1 and 2, where activity 1 has more weakly compliant actions than activity 2. Since weakly compliant actions do require a concurrent $ignore\_not\_permitted(a)$ action, activity 1 will have more $ignore\_not\_permitted(a)$ actions than activity 2. Hence, activity 2 will be preferred to activity 1, even though they both fall in the weakly compliant category. Gelfond and Lobo [12] do not consider such a feature.

Obligation Policies and Agent Behavior. So far we discussed authorization policies induced by an $\mathcal{AOPL}$ policy. To address obligation policies, we add policy fluents $obl\_compliant(do\_action)$ and $obl\_compliant(refrain\_from\_action)$ with policy actions $ignore\_obl(a)$ and $ignore\_obl(neg(a))$, as seen in Table 1. (For configurability, we consider obligation policies to *do* actions and to *refrain* from actions separately). We extend the definition of $policy\_compliant(f)$ to require both $obl\_compliant(do\_action)$ and $obl\_compliant(refrain\_from\_action)$ to be true. Like with authorization compliance, if $obl(a)$ is true but action a does not occur, then $obl\_compliant(do\_action)$ becomes false and remains false by inertia. Likewise for $obl\_compliant(refrain\_from\_action)$. If $ignore\_obl(a)$ or $ignore\_obl(neg(a))$ are performed, then these effects on the $obl\_compliant$ fluents are temporarily waived.

There are five different configurations (or behavior modes) an agent in the $\mathcal{APIA}$ architecture can have regarding its obligation policy (see Table 3). When in *subordinate* mode, the agent will never use either $ignore\_obl(a)$ and $ignore\_obl(neg(a))$ actions. Hence, all activities achieving $policy\_compliant(f)$ will be compliant with both aspects of its obligation policy. When in *best effort* mode, the agent prefers

	Honor $obl(\neg a)$	**Prefer honoring** $obl(\neg a)$	**Ignore** $obl(\neg a)$
Honor $obl(a)$	Subordinate	Permit commissions	(Not reasonable)
Prefer honoring $obl(a)$	Permit omissions	Best effort	(Not reasonable)
Ignore $obl(a)$	(Not reasonable)	(Not reasonable)	Utilitarian

Table 3: $\mathcal{APIA}$ Obligation Policy Modes

using other actions over these policy actions. Hence, activities will be compliant if possible but may include non-compliant elements when no other goal-achieving activities exist. The *permit omissions* and *permit commissions* options are variations of these modes. Mode *permit omissions* is like *best effort* with regards to $obl(a)$ policy statements, but like *subordinate* with regards to $obl(\neg a)$ policy statements. Likewise, *permit commissions* is like *subordinate* with regards to $obl(a)$ policy statements but like *best effort* regarding $obl(\neg a)$ statements. *Utilitarian* mode, like with authorization policies, reduces the behavior of an $\mathcal{APIA}$ agent with respect to its obligation policy to that of an $\mathcal{AIA}$ agent.

Behavior Mode Configurations. An agent's combined authorization and obligation policy configuration can be represented by a 2-tuple (A,O), where A is the authorization mode and O is the obligation mode. When an $\mathcal{APIA}$ agent is running in mode $(utilitarian, utilitarian)$, its behavior reduces to that of an $\mathcal{AIA}$ agent (i.e., policy actions are not used in this mode). This is due to an optimization we provide internally.

For each $\mathcal{APIA}$ configuration, we adjust the definition of $policy\_compliant(f)$ such that excess policy actions are not required. For instance, in the case of an agent with a subordinate authorization mode, we adjust $policy\_compliant(f)$ such that $ignore\_not\_permitted(a)$ is never needed since such an agent always disregards strong compliance.

4 Examples

To demonstrate the operations of an agent in the $\mathcal{APIA}$ architecture, we will introduce a series of examples that illustrate prototypical cases. For conciseness, we will focus on three $\mathcal{APIA}$ configurations: *(paranoid, subordinate)*, *(best effort, best effort)*, and *(utilitarian, utilitarian)*, and we limit ourselves to examples about authorization policies.

4.1 Example A: Fortunate case

To begin with a simple case, suppose that two people are in an office space that has four rooms with doors in between them. Room 1 is connected by door d_{12} to Room 2. Room 2 is connected by door d_{23} to Room 3 and so on. Door d_{34} has a lock and is currently in the unlocked position. Suppose our agent, Alice, wants to greet another agent, Bob. This scenario is represented by a dynamic domain description that considers:

- **Fluents:** $door\_locked(D)$ for each door D, $in\_room(P,R)$, $greeted\_by(P,A)$ where person P is greeted by person A; and
- **Actions:** $move\_through(A,D)$, $lock\_door(A,D)$, $unlock\_door(A,D)$, $greet(A,P)$, where A is the person doing the action, D is a door, and P a person (the direct object of the action).

Assume that agent Alice is given a policy specifying that all actions are permitted along with the following pre-computed activity that is stored in her knowledge base as the set of facts:

$$\begin{array}{l}\{activity(1),\quad length(1,4),\quad goal(1, policy\_compliant(greeted\_by(alice, bob))),\\ \quad component(1,1, move\_through(alice, d_{12})),\ component(1,2, move\_through(alice, d_{23})),\\ \quad component(1,3, move\_through(alice, d_{34})),\ component(1,4, greet(alice, bob))\}\end{array}$$

Before the control loop begins, Alice observes that she is in Room 1, Bob is in Room 4, the door d_{34} is unlocked, and that she has not yet greeted Bob.

At timestep 0, the first iteration of the control loop begins. In this first step, Alice analyzes her observations and interprets unexpected observations by assuming undetected exogenous actions occurred. None of her observations are unexpected, so no exogenous actions are assumed to occur. Alice then intends to wait at timestep 0. Alice attempts wait. Alice observes that her wait action was successful and that, in the meantime, the exogenous action

$$select(policy\_compliant(greeted\_by(alice, bob)))$$

happened. The time step is incremented and Alice does not observe any fluents.

The second iteration of the control loop begins. Alice analyzes her observation of $select(policy\_compliant(greeted\_by(alice, bob)))$ and determines that $active\_goal(policy\_compliant(greeted\_by(alice, bob)))$ is true. Alice then starts planning to achieve $policy\_compliant(greeted\_by(alice, bob))$ and determines that she intends to start activity 1. Since each action in activity 1 is strongly compliant, no policy actions are needed.

The rest of the example is very straight forward and is almost identical to scenarios discussed by [5, 6] in the $\mathcal{AIA}$ architecture.

4.2 Example B: Strong compliance degrades to weak compliance

Let us consider a less fortunate example, in which a strongly compliant activity becomes weakly compliant due to an unexpected environmental observation. In the same scenario, suppose we modify Alice's policy from Example A such that regarding $greet(A,P)$ we have:

$$\begin{array}{lll} permitted(greet(A,P)) & \textbf{if} & \neg busy\_working(P)\\ permitted(greet(A,P)) & \textbf{if} & busy\_working(P),\ in\_room(P,R),\\ & & door\_connects(D,R),\ knocked\_on\_door(D)\end{array} \tag{6}$$

We also have new fluents $busy\_working(P)$ and $knocked\_on\_door(D)$, and actions $begin\_working(A)$ and $knock\_on\_door(A,D)$. Let Alice's knowledge base contain two additional activities, 2 and 3, with the same goal as 1 and defined by the sets of facts:

$$\begin{array}{l}\{activity(2),\quad length(2,5),\quad goal(2, policy\_compliant(greeted\_by(alice, bob))),\\ \quad component(2,1, move\_through(alice, d_{12})),\ component(2,2, move\_through(alice, d_{23})),\\ \quad component(2,3, knock\_on\_door(alice, d_{34})),\ component(2,4, move\_through(alice, d_{34})),\\ \quad component(2,5, greet(alice, bob)),\\ \ activity(3),\quad length(3,2),\quad goal(2, policy\_compliant(greeted\_by(alice, bob))),\\ \quad component(3,1, knock\_on\_door(alice, d_{34})),\ component(3,2, move\_through(alice, d_{34}))\}\end{array}$$

Alice observes that Bob is not busy working, in addition to the initial observations of Example A. At timestep 0, the first iteration of the control loop begins. During the second iteration of the control loop (at timestep 1), Alice plans to achieve the $policy\_compliant(greeted\_by(alice,bob))$ goal. Since she believes Bob is not busy working, activity 1 is still strongly compliant and so is activity 2. Alice chooses activity 1 over activity 2 because it requires a shorter sequence of actions. She then executes activity 1 like in Example A until she enters Room 3, at which points she observes that Bob is busy working. During the next iteration (at timestep 5), the agent interprets this observation by inferring that $begin\_working(bob)$ happened at the previous timestep (4).

As a result, activity 1 becomes weakly compliant. Since Bob is busy working but Alice has not knocked on the door, no policy statement describes whether our next action, $greet(alice,bob)$, is compliant or not. If Alice is operating in *(utilitarian, utilitarian)* mode, she continues the execution of activity 1 and greets Bob anyway. (This happens without the use of policy actions due to our internal optimizations). Otherwise, Alice will stop the activity and then either refuse to plan another weakly compliant activity or use a concurrent policy action to dismiss this event.

If our agent is running in *(paranoid, subordinate)*, Alice will refuse to execute a weakly compliant activity. Through planning, Alice will discover that a new activity that includes knocking at the door is strongly compliant (e.g. activity 3) and begin its execution. If our agent is running in *(best effort, best effort)*, she will behave likewise because activity 3 is strongly compliant. The difference is that, if there did not exist a strongly compliant activity, she would plan a new activity that involved a policy action and greeted Bob anyway. Alice knocks on door d_{34} at timestep 7, greets Bob at timestep 8, and stops activity 3 at timestep 9.

4.3 Example C: Compliance degrades to non-compliant

Suppose we take policy rule (6), make it defeasible, and add this $\mathcal{AOPL}$ rule :

$$\neg permitted(greet(A,P)) \quad \textbf{if} \quad busy\_working(P),\ supervisor\_to(P,A)$$

Now, let us imagine that Bob is Alice's supervisor. Similar to Example B, our agent executes activity 1 until the observation that Bob is busy working. This time, we have a strict authorization statement forbidding greeting Bob since he is Alice's supervisor. Under the *(utilitarian, utilitarian)* option, we proceed on with activity 1 anyway. With the *(paranoid, subordinate)* option, our agent stops activity 1 but cannot construct a new activity that achieves the goal subject to its policy. Hence, the goal is futile and the agent waits until its environment changes such that a strongly compliant activity exists. Under the *(best effort, best effort)* option however, our agent constructs a new activity that contains greets Bob anyway. The activity contains: $greet(alice,bob)$, $ignore\_not\_permitted(greet(alice,bob))$, and $ignore\_neg\_permitted(greet(alice,bob))$.

4.4 Example D: Hierarchy of contradictory defeasible statements

Further extending Example C, suppose we turn all policy statements from Example A into defeasible ones (i.e., all actions are *normally* permitted) and add another policy statement:

$$\begin{array}{llll} m1(A,D): & \textbf{normally} & permitted(move\_through(A,D)) & \\ m2(A,D): & \textbf{normally} & \neg permitted(move\_through(A,D)) & \textbf{if} \quad in\_room(A,R_1), \\ & & & \quad door\_connects(D,R_2), \\ & & & \quad private\_office(R_2), \\ & & & \quad R_1 != R_2 \end{array}$$

and the static $private\_office(R)$ with $private\_office(r_4)$ as a fact. Since we have two contradictory defeasible statements, we need to add a preference between the two (without a preference our agent can non-deterministically choose between which of the two rules to apply). If we add:

$$prefer(m2(A,D), m1(A,D))$$

then, when Alice observes that Bob is not busy working at the beginning of the scenario, an agent running in *(paranoid, subordinate)* mode will immediately consider the goal to be futile. Unlike in Example C, our agent knows this immediately because $private\_office$ is a static, not an unexpected observation. If our agent is running in *(best effort, best effort)* mode, it creates an activity like activity 1, except that it contains $ignore\_not\_permitted(greet(alice, bob))$ and $ignore\_neg\_permitted(greet(alice, bob))$. Our utilitarian agent, like always, completely ignores our policy and executes activity 1.

5 Implementation

In this section, we discuss two important implementation aspects: the refactoring of the $\mathcal{AIA}$ implementation including its Theory of Intentions and control loop, and the implementation of the $\mathcal{APIA}$ control loop using CLINGO's Python API.

5.1 $\mathcal{AIA}$ Theory of Intentions and Control Loop

Since $\mathcal{APIA}$ takes $\mathcal{AIA}$ as a basis, we first update Blount *et al.*'s [6] $\mathcal{AIA}$ implementation such that it requires a state-of-the-art solver: CLINGO (version 5.4.1).[8] For this purpose, we re-implement the $\mathcal{AIA}$ logic program in ASP using only the description of the architecture presented by Blount *et al.* [5, 6]. During this process, we make minor modifications to $\mathcal{AIA}$ as a whole. First, we refactor the arrangement of ASP rules into multiple files according to their purpose in the $\mathcal{AIA}$ architecture (e.g. whether they are part of the Theory of Intentions, $\mathcal{AIA}$'s rules for computing models of history, or the $\mathcal{AIA}$ intended action rules). Second, we refactor the names of mental fluents in the Theory of Intentions so that their names are more descriptive and self-documenting. Thirdly, we extensively add inline comments to each ASP rule with reference quotations and page numbers from Blount *et al.*'s work. Lastly, we make minor corrections to ASP rules to match the translation of particular scenarios (i.e., histories) with the mathematical definitions proposed by Blount *et al.*.

In addition to upgrading the $\mathcal{AIA}$ logic program, we also refactor the implementation of the $\mathcal{AIA}$ control loop. In his dissertation, Blount [5] introduced the $\mathcal{AIA}$ Agent Manager. This is an interactive Java program that allows an end-user to assign values to agent observations in a graphical interface for each control loop iteration. Since this requires manual input, it does not easily lend itself to automation and reproducibility of execution, which are required for performance benchmarking. Furthermore, the $\mathcal{AIA}$ Agent Manager is structured around interacting with an underlying solver using subprocesses and process pipes. While the $\mathcal{AIA}$ Agent Manager could conceivably invoke CLINGO as a subprocess, CLINGO 5 provides a unique opportunity for more advanced integrations using its Python API.

Because of these two points, we replace the $\mathcal{AIA}$ Agent Manager with a new implementation of the $\mathcal{AIA}$ control loop written in Python 3.9.0. This new implementation uses a command-line interface and allows for reproducible execution through ASP input files. Since this control loop is also the basis for our $\mathcal{APIA}$ implementation, we will discuss it more in the next subsection.

[8] Our updated $\mathcal{AIA}$ implementation is available at `https://gitlab.com/0x326/miami-university-cse-700-aia.git` and is released under the MIT open-source license.

5.2 Python Component

We provide an implementation of the $\mathcal{AIA}$ control loop for the $\mathcal{APIA}$ architecture.[9] The $\mathcal{APIA}$ control loop is implemented using Python 3.9.0 and CLINGO 5.4.1 using CLINGO's Python API. We provide two modes: an automatic mode and a manual mode. The automatic mode is intended to be used for normal execution while the manual mode is intended to aid in debugging unexpected output in answer sets. The automatic mode uses a command-line interface to specify the ASP files of the input domain, the observations of the agent, and the $\mathcal{APIA}$ policy compliance mode the agent should use. The control loop then provides human-readable output as to what happens at each control loop step (see Figure 1 in the appendix).

In the case of unexpected output, the manual mode allows one to examine the answer set at each step of the control loop. It also provides scripts to highlight differences between answer sets of different timesteps in a visual manner and to step through the control loop like one would do in a traditional debugger. Additionally, manual mode addresses certain violations of $\mathcal{AIA}$ and $\mathcal{AOPL}$ underlying assumptions. For example, it generates an invalid predicate when there exists an action that is neither a physical, mental, or policy action. Likewise when an action is neither an agent action nor an exogenous action. In addition, it generates an invalid predicate when an $\mathcal{AOPL}$ policy statement describes an object that is not declared as an action. These rules have been very useful in debugging the implementation of the $\mathcal{APIA}$ architecture and they will aid future end-users who encode and execute scenarios using this architecture. Since these rules are intended during debugging, they are not executed during the automatic mode.

6 Conclusions and Future Work

In this paper, we created an architecture for a policy-aware intentional agent by bridging together previous work on intentional agents [5, 6] and reasoning algorithms for authorization and obligation policies [12]. A main difficulty was adapting the work on policy compliance so that it would be relevant for an agent deciding on which course of actions to take. While Gelfond and Lobo's work could determine whether a trajectory (i.e., sequence of actions) was strongly compliant, weakly compliant, or non-compliant, we introduced a wider range of agent behavior modes, which additionally explore the interactions between authorization and obligation policies.

This work can be further expanded by refining the decision making process in the planning phase of $\mathcal{APIA}$ by introducing a relative ranking system between activities that would achieve the same goal, based on the number of actions that are strongly, weakly, or non-compliant. Moreover, it would be interesting to allow the agent's controller to switch behavior modes while the agent is active, in the middle of executing an activity.

References

[1] Sandra Alves & Maribel Fernandez (2017): *A graph-based framework for the analysis of access control policies*. Theoretical Computer Science 685, pp. 3–22, doi:10.1016/j.tcs.2016.10.018.

[9] Our $\mathcal{APIA}$ implementation is available at `https://gitlab.com/0x326/miami-university-cse-700-apia.git` and is released under the MIT open-source license.

[2] Marcello Balduccini & Michael Gelfond (2008): *The AAA Architecture: An Overview*. In: *Architectures for Intelligent Theory-Based Agents, Papers from the 2008 AAAI Spring Symposium, 2008*, AAAI Press, pp. 1–6. Available at `https://www.aaai.org/Library/Symposia/Spring/2008/ss08-02-001.php`.

[3] Steve Barker (2012): *Logical Approaches to Authorization Policies*. In Alexander Artikis, Robert Craven, Nihan Kesim Cicekli, Babak Sadighi & Kostas Stathis, editors: *Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 7360, Springer, pp. 349–373, doi:10.1007/978-3-642-29414-3_19.

[4] Steve Barker, Guido Boella, Dov Gabbay & Valerio Genovese (2014): *Reasoning about delegation and revocation schemes in answer set programming*. *Journal of Logic and Computation* 24(1), pp. 89–116, doi:10.1093/logcom/exs014. Publisher: Oxford Academic.

[5] Justin Blount (2013): *An architecture for intentional agents*. Ph.D. thesis, Texas Tech University.

[6] Justin Lane Blount, Michael Gelfond & Marcello Balduccini (2014): *Towards a Theory of Intentional Agents*. In: *2014 AAAI Spring Symposium Series*, pp. 10–17. Available at `https://www.aaai.org/ocs/index.php/SSS/SSS14/paper/viewFile/7681/7708`.

[7] David Ferraiolo, Janet Cugini & D Richard Kuhn (1995): *Role-based access control (RBAC): Features and motivations*. In: *Proceedings of the 11th Annual Computer Security Applications Conference*, pp. 241–48. Available at `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=916537`.

[8] Ji Gao, Cheng-Xiang Yuan & Jing Wang (2005): *SASA5: A Method System for Supporting Agent Social Activities*. *Chinese Journal of Computers* 28(5), pp. 838–848. Available at `http://cjc.ict.ac.cn/eng/qwjse/view.asp?id=1764`.

[9] Michael Gelfond & Yulia Kahl (2014): *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Cambridge University Press, doi:10.1017/CBO9781139342124.

[10] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Computing* 9(3/4), pp. 365–386, doi:10.1007/BF03037169.

[11] Michael Gelfond & Vladimir Lifschitz (1998): *Action languages*. *Electronic Transactions on AI* 3(16), pp. 193–210. Available at `http://www.ep.liu.se/ej/etai/1998/007/`.

[12] Michael Gelfond & Jorge Lobo (2008): *Authorization and Obligation Policies in Dynamic Systems*. In Maria Garcia de la Banda & Enrico Pontelli, editors: *Logic Programming*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 22–36, doi:10.1007/978-3-540-89982-2_7.

[13] Bei-shui Liao, Hua-xin Huang & Ji Gao (2006): *An Extended BDI Agent with Policies and Contracts*. In Zhong-Zhi Shi & Ramakoti Sadananda, editors: *Agent Computing and Multi-Agent Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 94–104, doi:10.1007/11802372_12.

[14] Victor W. Marek & Miroslaw Truszczynski (1999): *Stable Models and an Alternative Logic Programming Paradigm*. In Krzysztof R. Apt, Victor W. Marek, Mirek Truszczynski & David Scott Warren, editors: *The Logic Programming Paradigm - A 25-Year Perspective*, Artificial Intelligence, Springer, pp. 375–398, doi:10.1007/978-3-642-60085-2_17.

[15] Yan-Bin Peng, Ji Gao, Jie-Qin Ai, Cun-Hao Wang & Hang Guo (2008): *An Extended Agent BDI Model with Norms, Policies and Contracts*. In: *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1–4, doi:10.1109/WiCom.2008.1197. ISSN: 2161-9654.

[16] Anand S. Rao & Michael P. Georgeff (1991): *Modeling Rational Agents within a BDI-Architecture*. In: *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pp. 473–484. Available at `https://dl.acm.org/doi/10.5555/3087158.3087205`.

[17] Khair Eddin Sabri & Nadim Obeid (2016): *A temporal defeasible logic for handling access control policies*. *Applied Intelligence* 44(1), pp. 30–42, doi:10.1007/s10489-015-0692-8.

A $\mathcal{APIA}$ Output

```
$ ../run.sh run_paranoid_subordinate_observations.lp --authorization-mode paranoid --obligation-mode subordinate | head -n +52
Iteration 0
  Step 1: Interpret observations
    Grounding...
    Solving...
    Unobserved actions: 0

  Step 2: Find intended action
    Grounding...
    Solving...
    Intended action: wait

  Step 3: Do intended action
    Doing wait

  Step 4: Observe world
    Getting observations from #program observations_1.

Iteration 1
  Step 1: Interpret observations
    Grounding...
    Solving...
    Unobserved actions: 0

  Step 2: Find intended action
    Grounding...
    Solving...
    Intended action: start(1)
    New activity:
      activity_goal(2,policy_compliant(greeted_by("Bob","Alice")))

  Step 3: Do intended action
    Skipping start(1)

  Step 4: Observe world
    Getting observations from #program observations_2.

Iteration 2
  Step 1: Interpret observations
    Grounding...
    Solving...
    Unobserved actions: 0

  Step 2: Find intended action
    Grounding...
    Solving...
    Intended action: move_through("Alice","d12")

  Step 3: Do intended action
    Doing move_through("Alice","d12")

  Step 4: Observe world
    Getting observations from #program observations_3.
```

Figure 1: Automatic execution of Example A using configuration $(paranoid, subordinate)$