Authorization and Obligation Policies in Dynamic Systems

Michael Gelfond[1] and Jorge Lobo[2]

[1] Texas Tech University
mgelfond@cs.ttu.edu
[2] IBM T. J. Watson Research Center
jlobo@us.ibm.com

Abstract. The paper defines a language for specifying authorization and obligation policies of an intelligent agent acting in a changing environment and presents several ASP based algorithms for checking compliance of an event with a policy specified in this language. The language allows representation of defeasible policies and is based on theory of action and change.

1 Introduction

The goal of this paper is to provide a simple language for specifying authorization and obligation policies of an intelligent agent acting in a changing environment. We refer to a pair consisting of an agent and its environment as a *dynamic system*. We limit our attention to dynamic systems which can be reasonably well represented by transition diagrams whose nodes correspond to possible physical states of the environment and arcs are labeled by actions. A transition $\langle \sigma, a, \sigma' \rangle$ belongs to such a transition diagram $\mathcal{T}$ iff σ' may be a state resulting from the execution of action a in state σ. If action a is deterministic and executable in σ then there exists exactly one such σ'. The system's diagram $\mathcal{T}$ contains all physically possible trajectories of the system. By an agent's *policy* we mean a description, $\mathcal{P}$, of a subset of trajectories of $\mathcal{T}$ deemed to be preferable by the system's designer. We often refer to such trajectories as *compliant* with $\mathcal{P}$.

We start with describing *authorization policy* of an agent $\mathcal{A}$ of a dynamic system $\langle \mathcal{A}, \mathcal{T} \rangle$ – a set of conditions under which an agent's action is or is not permitted. Note that the agent's use of the authorization policy can differ from application to application. Some authorization policies can be strict – no unauthorized action can be performed by an agent. In other cases an autonomous agent can opt for performing an unauthorized action. In this case the agent may be forced to pay a penalty, be commended for the initiative or loose his job for insubordination. In all these cases though it is important to be able to determine when an action is authorized and when it is not. Of course the agent can do this only on the basis of his general knowledge of the world, the system's current state, and its own abilities, and goals. The algorithms for checking compliance of agent's actions to his policies can be used by the agent for deciding what actions to perform as well

as by an outside observers evaluating the agent's behaviour. Similar observations are true for *obligation policy* – a set of conditions defining the actions the agent is obligated to perform or to abstain from performing in a given state.

In section 2 we describe the syntax and semantics of a language $\mathcal{APL}$ for specifying authorization policies of dynamic systems. We also discuss several methods for checking compliance of authorization policies using the methods of *Answer Set Programming* [1]. Section 3 expands $\mathcal{APL}$ with obligation policies.

2 Authorization Policy

In this section we define the syntax and semantics of the language $\mathcal{APL}$ for describing authorization policies in a dynamic system $\langle \mathcal{A}, \mathcal{T} \rangle$, and describe how checking compliance of actions and trajectories of the system can be reduced to computing answer sets of logic programs under the answer set semantics [2]. As expected, particular reductions will depend on the knowledge available to the reasoner checking the compliance. One of our main goals is simplicity of the language and a high level of elaboration tolerance of its policies. This of course should be balanced by the expressiveness of the language and the ability to check compliance in a reasonable amount of time.

2.1 Syntax

Let us consider a dynamic system described by an agent, $\mathcal{A}$, and a transition diagram $\mathcal{T}$ over some fixed signature Σ with sorts:

- *fluent* – functions whose values can change as a result of actions;
- *action* – by actions we mean *elementary actions*;
- *domain dependent sorts* representing elements of a particular domain.

We use (possibly indexed) letters f and e to denote elements of first two sorts. Possibly indexed letter a denotes *compound actions* – sets of (simultaneously executed) elementary actions. The corresponding capital letters denote variables ranging over the corresponding sorts. The set of elementary actions will be divided into the set of *agent's actions* and the set of *exogenous actions*. The former are actions executed by the agent $\mathcal{A}$ of the dynamic system. The latter are actions performed by other agents viewed as part of $\mathcal{A}$'s environment or by nature. Expressions of the form $f = y$ where f is a fluent and y is a possible value of f will be called *fluent atom*. Expressions of the form $e = true$ and $e = false$ will be referred to as *action atoms*. A Σ-*atom* is a fluent or action atom of Σ. Σ-atoms $l = true$ and $l = false$ will be often written as l and $\neg l$ respectively.

Recall that a *state* of $\mathcal{T}$ consists of an assignment of values to all the fluents of Σ. If $\langle \sigma, a, \sigma' \rangle$ is a transition of $\mathcal{T}$ then the pair $\langle \sigma, a \rangle$ will often be referred to as an *event*.

Now we are ready to define the language $\mathcal{APL}(\Sigma)$ for specifying authorization policy of an agent whose environment is described by $\mathcal{T}$.

The signature of $\mathcal{APL}(\Sigma)$ is obtained from Σ by adding a new predicate symbol $permitted(e)$, a collection of terms, $d_1, \ldots, d_n$ used to denote default authorization rules of $\mathcal{APL}(\Sigma)$, and the relation $prefer(d_1, d_2)$.

Definition 1. *[Authorization Policy]*
Authorization policy statements are expressions of the form

$$permitted(e) \text{ if } cond \tag{1}$$

$$\neg permitted(e) \text{ if } cond \tag{2}$$

$$d : \text{normally } permitted(e) \text{ if } cond \tag{3}$$

$$d : \text{normally } \neg permitted(a) \text{ if } cond \tag{4}$$

$$prefer(d_1, d_2) \tag{5}$$

where by *cond* we mean a collection of atoms of $\mathcal{APL}(\Sigma)$ not containing atoms formed by $prefer$.[3] (The last restriction is not necessary but it slightly simplifies the presentation). If *cond* is empty we simply omit "if *cond*" from the sentence. The first two statements will be referred to as *strict* policies. The next two will be called *defeasible*. Names of defeasible authorization statements are optional and can be omitted. By *authorization policy* we mean a collection of authorization policy statements.

Example 1. [Mission Command]
Consider the following (imaginary) policy requirements for mission authorization and command [3] :

1. A military officer is not allowed to command a mission he authorized.
2. A colonel is allowed to command a mission he authorized.
3. A military observer can never authorize a mission.

To express this policy we must have some information about transition system $\mathcal{T}_m$ which serves as a mathematical model of our domain. We assume that the signature, Σ_m of $\mathcal{T}_m$, contains two domain dependent sorts, *mission* and *commander*. Possibly indexed variables M and C range over missions and commanders respectively; Σ_m also contains

actions: $authorize(C, M)$ and $assume\_command(C, M)$;

fluents: $authorized(C, M)$, $commands(C, M)$, $colonel(C)$, and $observer(C)$.

Since the first two authorization policy statements of our example are contradictory we naturally assume them to be defeasible. The first policy statement will be expressed as[4]

[3] The full version of the language will include dynamic preferences, i.e. statements of the form $prefer(d_1, d_2)$ if *cond*. Conceptually the extension is not difficult but require a little more space.

[4] Note that the names of defaults contain the default variables.

$d_1(C, M)$: normally $\neg permitted(assume\_command(C, M))$ if $authorized(C, M)$

The second statement will be expressed as

$d_2(C, M)$: normally $permitted(assume\_command(C, M))$ if $colonel(C)$

Since the second defeasible policy is more specific we assume that it is preferred over the first one and, accordingly, add the sentence

$prefer(d_2(C, M), d_1(C, M))$

The last policy statement seems to be strict and will be represented by

$\neg permitted(authorize(C, M))$ if $observer(C)$

We will denote the resulting policy by $\mathcal{P}_m$.

2.2 Semantics

The semantics of an authorization policy $\mathcal{P}$ will determine a mapping $\mathcal{P}(\sigma)$ from states of $\mathcal{T}$ into *permissions* – sets of statements of the form $permitted(e)$, and *denials* – sets of statements of the form $\neg permitted(e)$.

To give the intuitively correct definition of $\mathcal{P}(\sigma)$ we should be able to refer to valid consequences of defaults expressed by defeasible rules of our authorization policy. This can be done by interpreting policies, states and events in terms of logic programs under the answer set semantics – a non-monotonic logical formalism well suited for reasoning with defaults. To this end we first translate authorisation statements of $\mathcal{APL}$ into their *logic programming counterparts*. The translation, lp, is defined as follows:

- $lp(f = y) =_{def} val(f, y)$;
- $lp(permitted(e)) =_{def} permitted(e)$.
- $lp(\neg permitted(e)) =_{def} \neg permitted(e)$.
- If S is a set of atoms then $lp(S) =_{def} \{lp(at) : at \in S\}$.
- lp of a strict policy statement, SPS, of $\mathcal{P}$ is obtained from SPS by simply replacing "if" by the "$\leftarrow$".
- A defeasible policy statement, DPS, is translated by lp into a standard Answer Set Prolog default rule:

  $permitted(e) \leftarrow lp(cond),$
  not $ab(d)$,
  not $\neg permitted(e)$.

  or

  $\neg permitted(e) \leftarrow lp(cond),$
  not $ab(d)$,
  not $permitted(e)$.
- The preference between two defeasible policies, d_1 and d_2 is translated by lp into

  $ab(d_2) \leftarrow lp(cond_1)$

  where $cond_1$ is the condition of d_1.

Finally

Definition 2. *[Logic programming counterparts of policies and events]*

$$lp(\mathcal{P}) =_{def} \{lp(st) : \text{ st } \in \mathcal{P}\}$$

$$lp(\mathcal{P}, \sigma) =_{def} lp(\mathcal{P}) \cup lp(\sigma)$$

These programs will be used to define important properties of authorization policies as well as their semantics:

Definition 3. *[Consistency of Authorization Policy]*
An authorization policy $\mathcal{P}$ for $\langle \mathcal{A}, \mathcal{T} \rangle$ is called *consistent* if for every state σ of $\mathcal{T}$ logic program $lp(\mathcal{P}, \sigma)$ is consistent, i.e. has an answer set.

Definition 4. *[$\mathcal{P}(\sigma)$ for authorization]*
Let $\mathcal{P}$ be a consistent authorization policy for $\langle \mathcal{A}, \mathcal{T} \rangle$. Then

- $permitted(e) \in \mathcal{P}(\sigma)$ iff a logic program $lp(\mathcal{P}, \sigma)$ entails $permitted(e)$[5].
- $\neg permitted(e) \in \mathcal{P}(\sigma)$ iff a logic program $lp(\mathcal{P}, \sigma)$ entails $\neg permitted(e)$.

Definition 5. *[Policy Compliance]*

- An event $\langle \sigma, a \rangle$ is *strongly compliant* with authorization policy $\mathcal{P}$ if for every $e \in a$ we have that $permitted(e) \in \mathcal{P}(\sigma)$.
- An event $\langle \sigma, a \rangle$ is *weakly compliant* with $\mathcal{P}$ if for every $e \in a$ we have that $\neg permitted(e) \notin \mathcal{P}(\sigma)$.
- An event $\langle \sigma, a \rangle$ is not compliant with $\mathcal{P}$ if for some $e \in a$ we have that $\neg permitted(e) \in \mathcal{P}(\sigma)$.
- A path $\langle \sigma_0, a_0, \sigma_1, ..., \sigma_{n-1}, a_{n-1}, \sigma_n \rangle$ of $\mathcal{T}$ is said to be *strongly (weakly) compliant* with $\mathcal{P}$ if for every $0 \leq i < n$ the event $\langle \sigma_i, a_i \rangle$ is strongly (weakly) compliant with $\mathcal{P}$.

Notice that $lp(\mathcal{P}, \sigma)$ may have answer sets S_1 and S_2 such that $permitted(e) \in S_1$ and $permitted(e) \notin S_2$. According to our definition $permitted(e)$ is not a consequence of $\mathcal{P}$, i.e. ambiguity is treated as a complete absence of knowledge. In some cases (probably most of the time) the system designer may want to avoid ambiguity and to limit himself to policies satisfying the following condition:

Definition 6. *[Categoricity]*
An authorization policy $\mathcal{P}$ for $\langle \mathcal{A}, \mathcal{T} \rangle$ is called *categorical* if for every state σ of $\mathcal{T}$ logic program $lp(\mathcal{P}, \sigma)$ as categorical, i.e. has exactly one answer set.

To illustrate these definitions let us go back to Example 1.

[5] A logic program Π entails literal l ($\Pi \models l$) if l belongs to every answer set of Π

Example 2. [Mission Command Revisited]
Let us populate the domain of Example 1 with a mission, m_1 and a commander c_1. One can use standard answer set programming techniques to easily prove that for any state σ and action e executable in σ the program $lp(\mathcal{P}_m, \sigma)$ is consistent (and categorical). Hence policy $\mathcal{P}_m$ is consistent and unambiguous.

Now let us consider an event $\langle \sigma_0, e_0 \rangle$ where

$\sigma_0 = \{colonel(c_1), authorized(c_1, m_1), \neg commands(c_1, m_1), \neg observer(c_1)\}$

and

$e_0 = assume\_command(c_1, m_1)$.

The answer set of $lp(\mathcal{P}_m, \sigma_0)$ contains $permitted(e_0)$ and hence the event is strongly compliant with $\mathcal{P}_m$. Similarly one can check that an event $\langle \sigma_1, e_0 \rangle$ where

$\sigma_1 = \{\neg colonel(c_1), authorized(c_1, m_1), \neg commands(c_1, m_1), \neg observer(c_1)\}$

is not compliant with $\mathcal{P}_m$. Finally consider a policy $\mathcal{P}'_m$ obtained from $\mathcal{P}_m$ by removing its first authorization rule. Again one can easily check that the event $\langle \sigma_1, e_0 \rangle$ is weakly (but not strongly) compliant with $\mathcal{P}'_m$.

2.3 Checking Compliance

In this section we discuss the ways to automatically check compliance of an agent's behaviour with consistent authorization policy $\mathcal{P}$ of $\langle \mathcal{A}, \mathcal{T} \rangle$. The algorithms will obviously depend on the type of input information and the goals of the checker. Let us start with the simplest possible scenario:

Scenario 1: an agent, acting in an environment $\mathcal{T}$, has *complete knowledge about his current state, σ, and contemplates the execution of action e.*

The following proposition, which follows immediately from the definition of compliance and properties of ASP logic programs, reduces the task to checking consistency of logic programs.

Proposition 1. *[Checking compliance of a completely known event]*

- Event $\langle \sigma, e \rangle$ is strongly compliant with consistent policy $\mathcal{P}$ of $\mathcal{T}$ iff a logic program
$$lp(\mathcal{P}, \sigma) \cup \{\leftarrow permitted(e)\}$$
is inconsistent.
- Event $\langle \sigma, e \rangle$ is weakly compliant with $\mathcal{P}$ iff a logic program
$$lp(\mathcal{P}, \sigma) \cup \{\leftarrow \neg permitted(e)\}$$
is consistent.
- Event $\langle \sigma, e \rangle$ is not compliant with policy $\mathcal{P}$ iff a logic program
$$lp(\mathcal{P}, \sigma) \cup \{\leftarrow \neg permitted(e)\}$$
is inconsistent.

Now let us look at the slight generalization of this scenario.

Scenario 2: Suppose that the *agent's knowledge about the current state is limited to the values of some (but not necessarily all) fluents.*

Let us denote the collection of such fluent atoms by s, and assume that $\delta(s)$ consists of all states of the system containing s. If an event $\langle \sigma, e \rangle$ is strongly (weakly) compliant with the agent's policy for every $\sigma \in \sigma(s)$ then the execution of e is obviously authorized (not prohibited); if for every $\sigma \in \sigma(s)$ the event $\langle \sigma, e \rangle$ is not permitted the agent will be wise not to perform e. Otherwise the agent does not have enough information to determine compliance of the event. But how our reasoner can check which of the above conditions (if any) are satisfied? It is obvious that to be able to do that he needs sufficient knowledge about $\delta(s)$. The precise logical form of this knowledge depends on the way we choose to describe our transition system $\mathcal{T}$. For the purposes of this paper we assume that such description is given by an action theory, A, in an action language $\mathcal{AL}$ [4] Such action theories provide a concise and convenient way of describing a large class of discrete dynamic systems. In particular we will need *static causal laws* (often referred to as *state constraints*) of $\mathcal{AL}$ – statements of the form

$$f = y \text{ if } P \tag{6}$$

where P is a collection of atoms of signature Σ of $\mathcal{T}$. We say that a (partial) assignment of values to fluents of Σ *satisfies* (6) if it contains $f = y$ or does not contain P. A *state* of a transition system defined by action theory A is a complete assignment of values to fluents which satisfies all the static causal laws of A. Partial assignment satisfying these laws is called a *simple knowledge state* of an agent with action theory A.

To compute $\sigma(s)$ we expand the translation lp of our policy statements into logic programs to the laws of the form (6): function lp will map (6) into

$$val(f, y) \leftarrow lp(P).$$

(The translation follows [5].) Let D be the collection of all statements of the form

$$val(f, y_1) \text{ or } \ldots \text{ or } val(f, y_k)$$

where f is a fluent, $\{y_1, \ldots, y_k\}$ is the set of all its possible values, and let SL be the set of all static causal laws from the action theory A describing $\mathcal{T}$. The following proposition reduces computing of $\delta(s)$ to finding answer sets of logic programs.

Proposition 2. *[States compatible with partial knowledge state s]*
Let s be a simple knowledge state of an agent with action theory A. Then $\sigma \in \delta(s)$ iff $lp(\sigma)$ is an answer set of $lp(s) \cup D \cup lp(SL)$.

The following proposition provides the means for checking authorization status of action e given a simple knowledge state s of an agent whose transition diagram is given by an action theory A.

Proposition 3. *[Checking compliance given simple knowledge state]*
For any consistent authorization policy $\mathcal{P}$

- Event $\langle \sigma, e \rangle$ is strongly compliant with $\mathcal{P}$ for every $\sigma \in \delta(s)$ iff program $lp(\mathcal{P}, s) \cup D \cup lp(SL) \cup \{\leftarrow permitted(e)\}$ is inconsistent.
- If $\mathcal{P}$ is categorical then an event $\langle \sigma, e \rangle$ is weakly compliant with $\mathcal{P}$ for every $\sigma \in \delta(s)$ iff program $lp(\mathcal{P}, s) \cup D \cup lp(SL) \cup \{\leftarrow \text{ not } \neg permitted(e)\}$ is inconsistent.
- Event $\langle \sigma, e \rangle$ is not compliant with $\mathcal{P}$ for every $\sigma \in \delta(s)$ iff program $lp(\mathcal{P}, s) \cup D \cup lp(SL) \cup \{\leftarrow \neg permitted(e)\}$ is inconsistent.
- If $\mathcal{P}$ is categorical then an event $\langle \sigma, e \rangle$ is not compliant with $\mathcal{P}$ for some $\sigma \in \delta(s)$ iff program $lp(\mathcal{P}, s) \cup D \cup lp(SL) \cup \{\leftarrow \text{ not } \neg permitted(e)\}$ is consistent.

Scenario 3: In many cases however the agent's knowledge base contains neither current physical nor current simple knowledge state of the system. Instead, as in the agent architecture from [6], it may maintain history, H_n, of the system's activity – *the complete or partial description of the initial state σ_0 together with a collection of actions $e_0, \ldots, e_{n-1}$ performed in the domain up to the current time-step n*[6]. We discuss how, given this information, the agent can check if he is permitted to execute a particular action e. First we will reify steps of history and define a new function, $lp(\mathcal{P}, I)$, obtained by adding step I as an additional (last) parameter to predicates from the definition of function lp. For instance,

$lp(f = y, I) =_{def} val(f, y, I)$,

$lp(e, I) =_{def} occurs(e, I)$,

etc. Similarly, for any history $H_n = \langle s, [e_0, \ldots, e_{n-1}] \rangle$

$lp(\mathcal{P}, H_n) =_{def} lp(\mathcal{P}, I) \cup lp(s, 0) \cup lp(e_0, 0) \cup \ldots \cup lp(e_{i-1}, I-1)$.

By D_0 we denote the collection of all statements of the form

$$val(f, y_1, 0) \text{ or } \ldots \text{ or } val(f, y_k, 0)$$

where f is a fluent and $\{y_1, \ldots, y_k\}$ is the set of all its possible values.

Finally, let $Check_1(H_n)$ and $Check_2(H_n)$ be pairs of rules

$\neg strongly\_compliant \leftarrow occurs(E, I), \text{ not } permitted(E, I)$

$\leftarrow \text{ not } \neg strongly\_compliant$

and

$\neg weakly\_compliant \leftarrow occurs(E, I), \neg permitted(E, I)$

$\leftarrow \text{ not } \neg weakly\_compliant$

[6] In addition history can contain observations of values of particular fluents at any step $0 \leq i \leq n$.

respectively. Let us recall that a trajectory $\sigma_0, e_0, \ldots, e_{n-1}, \sigma_n$ of $\mathcal{T}$ is called a *model* of history $H_n = \langle s, [e_0, \ldots, e_{n-1}] \rangle$ if $s \subseteq \sigma_0$ [1]. Intuitively such models are possible past compatible with the agent's knowledge. Policy compliance of an agent with history H_n can be checked using the following proposition.

Proposition 4. *[Checking compliance given system's history]]*
For any categorical authorization policy $\mathcal{P}$ and history H_n of the system

- Every model of H_n is strongly compliant with $\mathcal{P}$ iff a program $lp(\mathcal{P}, H_n) \cup D \cup lp(A) \cup Check_1$ is inconsistent.
- Every model of H_n is weakly compliant with $\mathcal{P}$ iff a program $lp(\mathcal{P}, H_n) \cup D \cup lp(A) \cup Check_2$ is inconsistent.

3 Obligation Policy

Now we are ready to consider obligation policies. As before we assume a fixed dynamic system described by an agent, $\mathcal{A}$, and a transition diagram $\mathcal{T}$ over signature Σ, and define syntax and semantics of the policy language, $\mathcal{AOPL}(\Sigma)$, allowing specification of authorization and obligation policies.

3.1 Syntax

The signature of the new language, $\mathcal{AOPL}(\Sigma)$, is obtained from the signature of $\mathcal{APL}(\Sigma)$ by adding a new predicate symbol $obl(E)$ where E is an elementary action of $\mathcal{A}$ or negation of such an action. Intuitively if $obl(e)$ is true in a state σ of a dynamic system $\langle \mathcal{A}, \mathcal{T} \rangle$ then agent $\mathcal{A}$ has an obligation of executing e in this state; if instead $obl(\neg e)$ holds in σ then $\mathcal{A}$ is obligated to refrain from executing this action.

Definition 7. *[Obligation Policy]*
Obligation policy statements of $\mathcal{AOPL}(\Sigma)$ are expressions of the form

$$obl(happening) \text{ if } cond \tag{7}$$

$$\neg obl(happening) \text{ if } cond \tag{8}$$

$$d : \text{normally } obl(happening) \text{ if } cond \tag{9}$$

$$d : \text{normally } \neg obl(happening) \text{ if } cond \tag{10}$$

$$prefer(d_1, d_2) \tag{11}$$

where *happening* stands for an elementary action of $\mathcal{A}$ or its negation and *cond* is a collection of atoms of $\mathcal{AOPL}(\Sigma)$ not containing atoms formed by $prefer$. The form of obligation rules is very similar to that of authorization rules. Syntactically permissions are replaced by obligations and actions by *happenings* – actions or their negations.

Example 3. [Student's Responsibilities]
Let us consider the following sentence from the list of student's responsibilities "Students are expected not to miss classes, to do their homework independently and to submit it on time". To represent this information we start with introducing sorts, *student*, *class*, *meeting* and *assignment* of the corresponding signature Σ. Students could be represented by names or social security numbers and classes by the corresponding course numbers (e.g. $cs4101$). Meetings and assignments will be represented by records $m(class, pos\_int)$ and $a(class, pos\_int)$. For instance, $m(cs4101, 3)$ refers to the third meeting of the class $cs4101$ while $a(cs4101, 5)$ refers to the fifth assignment given to students of this class. Signature Σ will also contain fluents

$enrolled(student, class)$,
$due\_date(meeting, assignment)$.

For instance Mary may be enrolled in $cs4101$ ($enrolled(Mary, cs4101)$) and the due date for the third assignment in this class may be the seventh class meeting, $due\_date(m(cs4101, 7), a(cs4101, 3)))$.

We will also need the following actions

$attend(student, meeting)$,
$submit(student, assignment, meeting)$,
$accept\_unauthorized\_help(student)$.

Now we need to think about our understanding of the obligation policy rules from our example. Are they strict or defeasible? The informal specification does not say and hence the decision is left to us. One can easily imagine the situation when the first rule can be canceled for some particular student and/or meeting by the introduction of some exceptional circumstances. A person can be released from the first obligation because of illness, family emergencies, etc. Instead of putting these exceptions in the condition of the first rule we adopt a more elaboration tolerant approach and view the rule as defeasible. This leads us to the following formal rule:

$d_1(S, C, N)$: normally $obl(attend(S, m(C, N)))$ if $enrolled(S, C)$.

In the properly extended signature an exception to this default can be represented as follows

$\neg obl(attend(S, M))$ if $family\_emergency(S, M)$.

where $family\_emergency(S, M)$ holds if student S has family emergency at the time of meeting M. Similarly for other exceptions. The second rule basically tells the students not to cheat. Since cheating is never justified we make this a strict obligation policy rule.

$obl(\neg accept\_unauthorized\_help(S))$.

The knowledge base we are building may be extended by possible exceptions to the third rule. Hence we make it defeasible: Normally a student S should submit

its assignment N_1 for class C at the N_2'th meeting of class C if S is enrolled in C and N_2 is the deadline for the assignment.

$$d_2(S, C, N_1, N_2) : \text{normally } obl(submit(S, a(C, N_1), m(C, N_2)) \text{ if } enrolled(S, C), due\_date(m(C, N_2), a(C, N_1)).$$

As expected preference between defeasible obligation policies will be used when policies lead to contradictory obligations. For instance a religious obligation of abstaining from work during important religious holidays can contradict the obligation of attending classes. Some schools allow such holidays to be a sufficient excuse for not attending classes, while others do not. In a simplified form the new obligation can be expressed as

$$d_3(S, M) : \text{ normally } obl(\neg attend(S, M)) \text{ if } religious\_holiday(M).$$

where $religious\_holiday(M)$ holds if some important religious holiday occurs at the same day as the meeting M. If the designer of our knowledge base believes that religious obligations overrule secular once he can expand the base by

$$prefer(d_3(S, m(C, N)), d_1(S, C, N)).$$

The opposite preference can be given by a more secularly minded designer. Of course if no preference is given the user of the base may have two different contradictory obligations. Such a policy however will be ambiguous and will allow the user freedom to decide if one or both obligations should be ignored and accept the corresponding rewards and punishments. As with authorization policies the designer should attempt to avoid ambiguty and limit himself to categorical policies.

3.2 Semantics

To define the semantics of $\mathcal{AOPL}(\Sigma)$ we expand the function $\mathcal{P}(\sigma)$ from the definition of the semantics of authorization policy of $\mathcal{APL}(\Sigma)$. In addition to permissions and denials the function will now return *obligations* the agent of a dynamic system has in a state σ. As expected this can be done by expanding logical counterpart lp defined above by mapping statements of the form (7) and (8) to logic programming rules

$$obl(h) \leftarrow lp(cond)$$

$$\neg obl(h) \leftarrow lp(cond).$$

Statements (9) and (10) will be mapped into rules

$$obl(h) \leftarrow lp(cond). \text{ not } ab(d), \text{ not } \neg obl(h).$$

$$\neg obl(h) \leftarrow lp(cond). \text{ not } ab(d), \text{ not } obl(h).$$

The notions of consistency and categoricity of a policy of a new language remains unchanged. The function $\mathcal{P}(\sigma)$ is expanded as follows

Definition 8. *[$\mathcal{P}(\sigma)$ for obligations]*
Let $\mathcal{P}$ be a consistent policy for $\langle \mathcal{A}, \mathcal{T} \rangle$. Then $obl(h) \in \mathcal{P}(\sigma)$ iff a logic program $lp(\mathcal{P}, \sigma)$ entails $obl(h)$.

Let $\mathcal{P}_a$ be a policy obtained from $\mathcal{P}$ by dropping its obligation rules; $\mathcal{P}_o$ is obtained from $\mathcal{P}$ by dropping its authorization rules. Obviously $\mathcal{P} = \mathcal{P}_a \cup \mathcal{P}_o$. We say that $\mathcal{P}_a$ is an authorization policy *induced* by $\mathcal{P}$. Similarly for $\mathcal{P}_o$.

Definition 9. *[Policy compliance]*
An event $\langle \sigma, a \rangle$ is *compliant* with obligation policy $\mathcal{P}$ if

1. For every $obl(e) \in \mathcal{P}(\sigma)$ we have that $e \in a$, and
2. For every $obl(\neg e) \in \mathcal{P}(\sigma)$ we have that $e \notin a$.

An event $\langle \sigma, a \rangle$ is *strongly (weakly) compliant* with arbitrary policy $\mathcal{P}$ from $\mathcal{AOPL}(\Sigma)$ if it is strongly (weakly) compliant with the authorization policy induced by $\mathcal{P}$ and with the obligation policy induced by $\mathcal{P}$.

Compliance of events with respect to an obligation policy can be checked by ASP methods similar to those used in Section 2.3. Space limitations preclude discussing the corresponding details.

4 Related Work

To illustrate the relationship between our work and more traditional methods for representing and reasoning with policies let us consider Role-based Access Control (RBAC) – a method used in computer system security for restricting system access to authorized users. The signature of a typical policy of RBAC contains object constants for users (called subjects), their roles (e.g. job functions or titles), operations (e.g. read or write) and resources (e.g. files or disk or databases) to which these operations can be applied. There are relations $plays\_role(S, R)$ – a user S plays a role R, $has\_permission(R, O, D)$ – every user playing a role R has the permission to apply operation O to resource D, and $R_1 \leq R_2$ - R_1 inherits permissions from R_2. There is also an action $execute(S, O, D)$ – user S executes operation O on resource D. States of the system should satisfy a constraint

$$has\_permission(R_1, P) \text{ if } has\_permission(R_2, P), \; R_1 \leq R_2.$$

A typical permission policy has the form:

$$permitted(execute(S, O, D)) \text{ if } plays\_role(S, R), \; has\_permission(R, O, D).$$

Hence the RBAC approach seems to be a very narrow special case of the methodology for specifying and reasoning about policies suggested in this paper. Note

that S can get access to the system (and therefore change its state) only if he is granted permission to do so. Normally the user will not be able to record his actions in the system's log. This will be done by an administrator with his own set of policies not expressible in the language of RBAC. Our method allows natural specification and reasoning with combined, administrative and access control policies. Authorization policies are typically defined as inputs to access control systems. Most access control system more or less follow the operational model behind the XML access control language XACML [7]: given the current state of a system there is a (some times partial) function encoding the policy that maps the state and an operation on the system into a decision as to whether the operation is permitted. Most formal modeling work and analysis of policies make the simplified assumption that policies depend on a subset of the state that does not change over time or if it changes the changes are only made by an administrator of the access control system and they are ignored. For example a system implementing Access Control Lists (ACL) fixes a list of subjects for each operation or a set of operations and if the subject requesting permission to execute the operation is in the list the operation is permitted.[7] Only administrators are allowed to make changes to the ACL. In RBAC only administrators can define new roles and assign permissions to roles. More sophisticated associations can be made between subjects and permissions if one can express the associations as predicates over the state. This is what is expressed operationally with a function in the definition of XACML policies. Still these predicates are on parts of the state that don't change over time (i.e., they change only by administrative changes of the system). There is a large body of work in policy analysis and policy modeling but mostly in these static situations. Barker [8] uses stratified logic programs to describe RBAC-like systems. In a series of papers Jojadia and his co-authors [9–12] developed the Flexible Authorization Framework (FAF) using stratified logic programs. FAF is a very sophisticated extension of RBAC that incorporates positive and negative authorizations as well as methods to express different conflict resolution *policies*, all in the context of access control to relational databases. They handle some state dynamics but it is limited to their database system model. Stoller et al. [13] use planning techniques to characterize the complexity and solve problems over administrative operations in RBAC systems. Kolovski et al. [14] use description logic to formalize and analyze XACML policies including administrative policies. Halpern and Weissman use a subset of first order logic to specified policies (without the system) and do analysis [15].

The closest to our work are the models presented in [16] and [17]. [16] is mostly concerned with obligations. The authors, as we do, present a model for obligations and authorizations that incorporates the model of a system where the policies are supposed to be enforced. Although the authors claim that minimal changes will accommodate a more general model, obligations are expressed as a condition to obtain authorization to access to a resource. In our case obligations are separate from authorizations and we can represent mutual depen-

[7] This is call a white list. There is a complementary implementation (black list) in which only subjects not in the list are permitted to execute the operation.

dencies between obligations and authorizations. Their model is agnostic to how system transitions and policies are represented. Systems are considered to be a set of state traces (sequences) and obligations abstract functions constraining that set. We have shown in the paper that by choosing an action theory described in an $\mathcal{A}$-like language, we can check system properties using ASP logic programs. Theirs is a theoretical framework to define obligations, ours is more a specification framework where we would like to facilitate the definition of policies and check system properties. Craven et al.[17] also model authorizations, obligations and the system together. They use the event calculus [18] for system description and ASP logic programs to represent the policies. Their main goal is to prove properties of the policies, not of the systems. They use abductive constraint logic programming as proof framework [19]. Finally, [20] provides a model of non-monotonic authorization based on a paraconsistent variant of extended logic programs ASP semantics. Despite multiple differences this can be viewed as a precursor of our work. One fundamental difference between all the work referenced above (except that in [20]) and ours is the presence of defeasible policies. Policies are assumed to be strict limiting the modeling of complex scenarios.

5 Conclusion

We presented a simple and general language, $\mathcal{AOPL}(\Sigma)$, for specifying authorization and obligation policies of an intelligent agent acting in a changing environment and presented several ASP based algorithms for checking compliance of an event with a policy specified in this language. The language has the following distinctive features:

- Our approach allows to represent and reason about both, static and dynamic domains.
- The ability to represent defeasible policies improves elaboration tolerance of the policies and flexibility of the agent's behavior.
- Policy specifications and algorithms can be naturally encorporated into various software systems including agents with high degree of autonomy, access control and other security system, etc. They can be used by an agent for finding authorized sequences of actions to achieve their goals and to fulfill their obligations, as well as by systems monitoring such a compliance.
- The compliance checking methods are based on theory of action and change and on answer set programming. This allows the use of general reasoning techniques and systems (answer set solvers) and simplifies the correctness proofs of the corresponding algorithms.

in the full version of this paper we will test the expressibility of our language on the wide variety of policies and refine and expand the ASP based methods for checking event compliance and other policy related reasoning tasks.

Acknowledgments: We would like to thank Vladimir Lifschitz for drawing our attention to work by T. Woo and S. Lam.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP-88. (1988) 1070–1080
3. Bandara, A., Calo, S., Lobo, J., Lupu, E., Russo, A., Sloman, M.: Toward a formal characterization of policy specification and analysis. In: Electronic Proceedings of the Annual Conference of ITA (ACITA). (2007)
4. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer Academic (2000) 257–279
5. Turner, H.: Representing actions in logic programs and default theories: A situation calculus approach. J. Log. Program. **31**(1-3) (1997) 245–298
6. Balduccini, M., Gelfond, M.: The aaa architecture: An overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08). (2008)
7. OASIS Standard: extensible access control markup language (XACML) v2.0 (2005)
8. Barker, S.: Security policy specification in logic. In: Proc. of Int. Conf. on Artificial Intelligence. (June 2000) 143–148
9. Jajodia, S., Samarati, P., Subrahmanian, V., Bertino, E.: A unified framework for enforcing multiple access control policies. In: Proc. of the ACM Int. SIGMOD Conf. on Management of Data. (May 1997)
10. Jajodia, S., Samarati, P., Subrahmanian, V.: A logical language for expressing authorizations. In: Proc. of the IEEE Symposium on Security and Privacy. (1997) 31
11. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. ACM Trans. Database Syst. **26**(2) (2001) 214–260
12. Chen, S., Wijesekera, D., Jajodia, S.: Incorporating dynamic constraints in the flexible authorization framework. In: ESORICS. (2004) 1–16
13. Stoller, S.D., Yang, P., Ramakrishnan, C.R., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: ACM Conference on Computer and Communications Security. (2007) 445–455
14. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing web access control policies. In: WWW. (2007) 677–686
15. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. In: Proc. of 16th IEEE Computer Security Foundations Workshop. (2003) 251–265
16. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Obligations and their interaction with programs. In: ESORICS. (2007) 375–389
17. Craven, R., Lobo, J., Lupu, E., Ma, J., Russo, A., Sloman, M., Bandara, A.: A formal framework for policy analysis. Technical Report, Department of Computing, Imperial College London (2008)
18. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
19. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive constraint logic programming. J. Log. Program. **44**(1-3) (2000) 129–177
20. Woo, T.Y.C., Lam, S.S.: Authorizations in distributed systems: A new approach. Journal of Computer Security **2**(2-3) (1993) 107–136