Knowledge Representation in ASP and Reasoning about Policies

Michael Gelfond

Texas Tech University

Joint Work with Jorge Lobo

Some Questions

- **Logic: What are correct consequences of a collection of defaults?**
- **Computer Science: Why is programming so difficult and how to make it easier?**
- **Software Engineering: How to build software components of intelligent agents capable of reasoning and acting in a changing environment?**

Common Thread: Declarative Programming

Declarative Programming is an attempt to apply axiomatic method of mathematics to computer science and to practice of programming.

Need to know how to

- **represent knowledge about given domain,**
- **compute consequences of this knowledge,**
- **reduce computational problems to computing these consequences.**

Outline of the Talk

1. I review some things we learned about knowledge representation:

- **defaults and their exceptions,**
- **CR-Prolog,**
- **dynamic domains.**

2. Explain how this knowledge can be used to solve an important problem of representing and reasoning with authorization policies.

Representing Defaults

Questions:

- **What is a default?**
- **How to represent defaults and their exceptions?**
- **What is the set of valid conclusions entailed by a default theory?**
- **How to arrive at these conclusions?**

Representing Defaults

A default "Normally elements of class C have property P" is often represented by a rule:

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \end{aligned}$$

where $d(X)$ is the name of the default.

Direct Exceptions to Defaults

Default $d(X)$ may have two types of *direct* exceptions:

***strong* exception refutes the default's conclusion,**

***weak* exception renders the default inapplicable.**

Suppose that exceptions to default $d(X)$ are given by relation $e(X)$.

Completeness of this set can be expressed by CWA:

$$\neg e(X) \leftarrow \ not\ e(X)$$

Direct Exceptions to Defaults

A weak exception $e(X)$ to $d(X)$ is encoded as

$$ab(d(X)) \leftarrow \ not\ \neg e(X).$$

If e is a strong exception we need one more rule,

$$\neg p(X) \leftarrow e(X)$$

which will allow us to defeat d's conclusion.

Indirect Exceptions to Defaults

Consider rules

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \\ q(X) \leftarrow\ & p(X). \end{aligned}$$

and two observations: $c(x)$ and $\neg q(x)$.

A commonsense conclusion leads us to believe that x is an (indirect) exception to the default. However our program is inconsistent.

Question: how to represent indirect exceptions?

Indirect Exceptions to Defaults

Possible informal explanation may follow from an axiom:

"for every X from class c, $ab(d(X))$ may be true, but such a possibility is very rare".

One may informally argue that since the application of the default to x leads to a contradiction x must satisfy this rare property.

The axiom can be represented by a *consistency restoring rule* **(cr-rule)**

$$ab(d(X)) \stackrel{+}{\leftarrow} c(X)$$

of CR-Prolog.

Indirect Exceptions in CR-Prolog

In CR-Prolog default $d(X)$ can be represented as

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \\ ab(d(X)) \stackrel{+}{\leftarrow}\ & c(X). \end{aligned}$$

According to the semantics of CR-Prolog this rule, used together with

$$\begin{aligned} q(X) \leftarrow\ & p(X). \\ & c(x). \end{aligned}$$

entails $p(x)$ and $q(x)$. But if we add

$$\neg q(x)$$

these conclusions will be withdrawn.

CR-Prolog: syntax and semantics

CR-Prolog extends ASP by cr-rules:

$$l \stackrel{+}{\leftarrow} body$$

$stand(\Pi)$ **- the collection of standard rules of CR-Prolog program** Π**.**

$\alpha(r)$ **- a standard rule obtained from a cr-rule** r **by replacing** $\stackrel{+}{\leftarrow}$ **by** $\leftarrow$**.**

For a set $\mathcal{R}$ **of cr-rules,** $\alpha(\mathcal{R}) = \{\alpha(r) : r \in \mathcal{R}\}$**.**

A minimal (with respect to set theoretic inclusion) collection $\mathcal{R}$ **of cr-rules of** Π **such that** $stand(\Pi) \cup \alpha(\mathcal{R})$ **has an answer set is called an** *abductive support* **of** Π**.**

A set S **is called an** *answer set* **of** Π **if it is an answer set of ASP program** $stand(\Pi) \cup \alpha(\mathcal{R})$ **for some abductive support** $\mathcal{R}$ **of** Π**.**

What is next?

To learn more about (efficient) reasoning with defaults one may attempt to

- **Improve the efficiency of ASP reasoning (my favorite approach is to intelligently combine various methods of reasoning including ASP, CLP, SAT, SMT, etc.)**
- **Have a serious mathematical study of CR-Prolog and improve the efficiency of its reasoning engine.**
- **Study defaults in a richer environment: formulas (with quantifiers) instead of literals, probabilistic information, etc.**

Reasoning about Dynamic Systems

Informally *dynamic system* **consists of an agent and its environment.**

We assume that environment can be modeled by a transition diagram, $\mathcal{T}$ **whose nodes correspond to its possible physical states and arcs are labeled by actions.**

A transition $\langle \sigma, a, \sigma' \rangle \in \mathcal{T}$ **iff** σ' **may be a state resulting from the execution of action** a **in state** σ**.**

The system's transition diagram contains all physically possible trajectories of the system.

Reasoning about Dynamic Systems

Due to the size of the diagram, the problem of finding its concise specification is not trivial and has been a subject of research for a comparatively long time.

(a) To describe the change one needs to describe causal effects of actions in the presence of complex interrelations between fluents;

(b) To describe what stayed the same one needs to represent the law of inertia.

Describing Change in ASP

A direct effect of action a informally described as "execution of a in a state satisfying p causes f" can be represented in ASP as

$$holds(f, S+1) \leftarrow occurs(a, S), \\ holds(p, S).$$

Relations between fluents are described by rules of the form:

$$holds(f, S) \leftarrow holds(p, S)$$

(if fluents from p are true in state S than so is f).

Describing Inertia in ASP

The inertia axiom which says that "Things tend to stay unchanged" can be viewed as a default.

Hence it can be represented as

$$\begin{aligned} holds(F, S+1) \leftarrow\ & inertial\_fluent(F, S), \\ & holds(F, S), \\ & not\ ab(inertia(F, S)), \\ & not\ \neg holds(F, S+1). \\ \neg holds(F, S+1) \leftarrow\ & inertial\_fluent(F, S), \\ & \neg holds(F, S), \\ & not\ ab(inertia(\neg F, S)), \\ & not\ holds(F, S+1). \end{aligned}$$

Reasoning about Dynamic Systems in ASP

Three types of axioms defined above provide a surprisingly good and simple model of causal reasoning.

They describe direct and indirect effects of actions as well as inertia.

From practical standpoint they led to the development of ASP planning, scheduling, and diagnostic algorithms and systems.

They also shed some light on the relationship between two basic philosophical notions - that of *belief* and *causality*.

Authorization Policies in Dynamic Systems

Consider a dynamic system with agent A and environment described by a transition diagram $\mathcal{T}$.

By agent's *authorization 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 $\mathcal{P}$.

Different Requirement for Compliance

Authorization policies define conditions under which an agent's action is or is not permitted.

In some situations agent's compliance with the policy must be strict – no unauthorized action can be performed.

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.

Compliance Checking Algorithms

Independently of the compliance requirements the system needs algorithms determining when an action is authorized and when it is not.

To check that the algorithms needs knowledge about the policy and about the world.

The algorithms can be used by the agent for deciding what actions to perform as well as by outside observers evaluating the agent's behaviour.

Language $\mathcal{APL}(\Sigma)$ for Specifying Authorizations

Authorization policy statements are expressions of the form

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

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

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

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

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

Example

Policies to refine:

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.**

We need to know the signature of dynamic domain associated with these policies.

Domain Signature Σ

sorts: $mission$ **and** $commander$**.**

actions:

$authorize(C, M)$ **and** $assume\_command(C, M)$

fluents:

$authorized(C, M)$**,** $command(C, M)$**,**

$colonel(C)$**,** $observer(C)$

Example Policies in $\mathcal{APL}(\Sigma)$

Policy $d_1(C, M)$ - "A military officer is not allowed to command a mission he authorized" will be viewed as default and represented by:

- **normally** $\neg perm(assume\_com(C, M))$ **if** $auth(C, M)$

Similarly for policy $d_2(C, M)$ - "A colonel is allowed to command a mission he authorized":

- **normally** $perm(assume\_com(C, M))$ **if** $colonel(C)$

We prefer more specific policy hence:

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

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

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

Example Policies in $\mathcal{APL}(\Sigma)$

- **In a state:**

$auth(c, m),\ \ colonel(c),\ \ \neg observer(c),\ \ \neg command(c, m)$

policy $\mathcal{P}_m$ defined above entails

$perm(assume\_com(c, m))$**.**

- **In a state:**

$auth(c, m),\ \neg colonel(c),\ \neg observer(c),\ \neg command(c, m)$

it entails

$\neg perm(assume\_com(c, m))$**.**

- **In a state:**

$auth(c, m),\ \ colonel(c),\ \ observer(c), \neg command(c, m)$

it entails

$perm(assume\_com(c, m))$ **and** $\neg perm(authorize(c, m))$**.**

Semantics of $\mathcal{APL}(\Sigma)$

The semantics of a policy $\mathcal{P}$ gives a mapping $\mathcal{P}(\sigma)$ from states of $\mathcal{T}$ into sets of actions permitted and prohibited in these states.

This is done by translating a state σ and $\mathcal{P}$ into a logic program $lp(\mathcal{P}, \sigma)$.

- **$\mathcal{P}$ is *consistent* if for every σ program $lp(\mathcal{P}, \sigma)$ has an answer set.**

- **$\mathcal{P}$ is *unambiguous* if for every σ program $lp(\mathcal{P}, \sigma)$ has exactly one answer set.**

- **$\mathcal{P}$ is *complete* if for every σ program $lp(\mathcal{P}, \sigma)$ has exactly one answer set, A, and for every action e executable in σ, $permitted(e) \in A$ or $\neg permitted(e) \in A$.**

From Policies to Logic Programs

- **Strict policy is written as**

$permitted(e) \leftarrow cond.$

- **Defeasible policy d is written as a default:**

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

- **Preference "$prefer(d_1, d_2)$" written as**

$ab(d_2) \leftarrow cond_1$

where $cond_1$ is the condition of default d_1.

$lp(\mathcal{P}, \sigma)$ is the resulting program $lp(\mathcal{P})$ combined with σ represented as a collection of literals.

e is permitted in σ if $lp(\mathcal{P}, \sigma) \models permitted(e)$.

e is prohibited in σ if $lp(\mathcal{P}, \sigma) \models \neg permitted(e)$.

Defining 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 $\mathcal{P}$.**

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.

Comment

Methodology of representing defaults and dynamic systems in ASP was useful for solving a difficult problem of representing and reasoning with authorization policies.

The solution is simpler and substantially more general than other current solutions.

Moreover it allows development of simple algorithms checking compliance.

Conclusion

ASP allows elegant solutions of classical KR problems, and provides powerful tool for declarative programming.

Improving efficiency of ASP reasoning mechanisms can very substantially increase the range of applicability of declarative programming.

Also need serious study of practical KR languages and of mathematical properties of standard ASP including CR-Prolog and P-log.