# Conflict Resolution in Policy Management

Jan Chomicki

University at Buffalo

State University of New York

`http://www.cse.buffalo.edu/~chomicki`

*Joint work with Jorge Lobo, Bell Labs, and Shamim Naqvi, Convexant.*

# Policies

**Policies** are collections of general principles specifying the desired behavior of a system. Potential application areas:

- communications, network management and monitoring (IETF)
- electronic commerce (IBM CommonRules)
- security and access management.

Examples:

- *If a fax from the Chicago office arrives at the main office fax machine, redirect the fax to Joe's office fax machine.*
- *As soon as an order is received, the ordered product should be mailed and the customer's credit card charged. Defective products shouldn't be mailed.*

# Policy management

Policy execution:

- evaluation
- **conflict detection and resolution**

Policy maintenance:

- specification
- modification
- analysis
- ...

# PDL

PDL policies defined as sets of **Event-Condition-Action** rules.

The policy:

> *If a fax from the Chicago office arrives at the main office fax machine, redirect the fax to Joe's office fax machine.*

is specified in PDL as:

$$\begin{array}{ll} \textit{arrivedFaxOff} & \textbf{causes}\ \textit{sendFaxJoeOff}\,(\textit{arrivedFaxOff.content}) \\ & \textbf{if}\ \ \textit{arrivedFaxOff.from} = \text{``Chicago''}. \end{array}$$

# Action conflicts

A policy manager may specify that several actions **cannot be simultaneously executed**.

Example:

$$\mathit{requestRes}\ \textbf{causes}\ \mathit{processRes}(\mathit{requestRes.user})$$

Two simultaneous reservation requests cannot both be satisfied:

$$\textbf{never}\ \ \mathit{processRes}(\mathit{User_1}) \wedge \mathit{processRes}(\mathit{User_2})\ \textbf{if}\ \ \mathit{User_1} \neq \mathit{User_2}.$$

Reserved resources: bandwidth, airport runway,...

# Action constraints

Syntax:

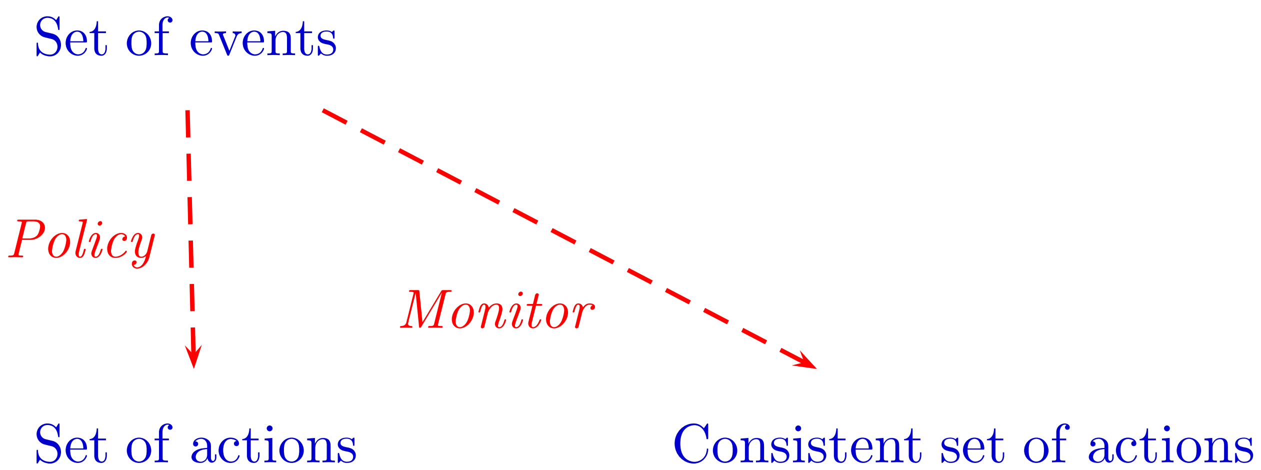
$$\textbf{never} \;\; A_1 \wedge A_2 \wedge \cdots \wedge A_n \;\textbf{if}\;\; C$$

Logical reading:

$$\forall \neg (A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge C)$$

# Conflict resolution

A **monitor** detects and resolves conflicts among the actions generated by a policy.



# Classes of monitors

Different resolution strategies.

**Action-based** monitors:

- blocking a conflicting action (**action cancellation** monitor)
- delaying a conflicting action (**action delay** monitor)

**Event-based** monitors:

- ignoring the events causing a conflicting action(**event cancellation** monitor)
- postponing the events that cause a conflicting action (**event delay** monitor)

# Example

Policy:

$defectiveProduct$ **causes** $stop$

$orderReceived$ **causes** $mailProduct$

$orderReceived$ **causes** $chargeCreditCard$

Constraint:

**never** $stop \wedge mailProduct$

The customer does not want to be charged if an ordered defective product is not mailed!

Adding the constraint

$$\textbf{never} \quad stop \wedge chargeCreditCard$$

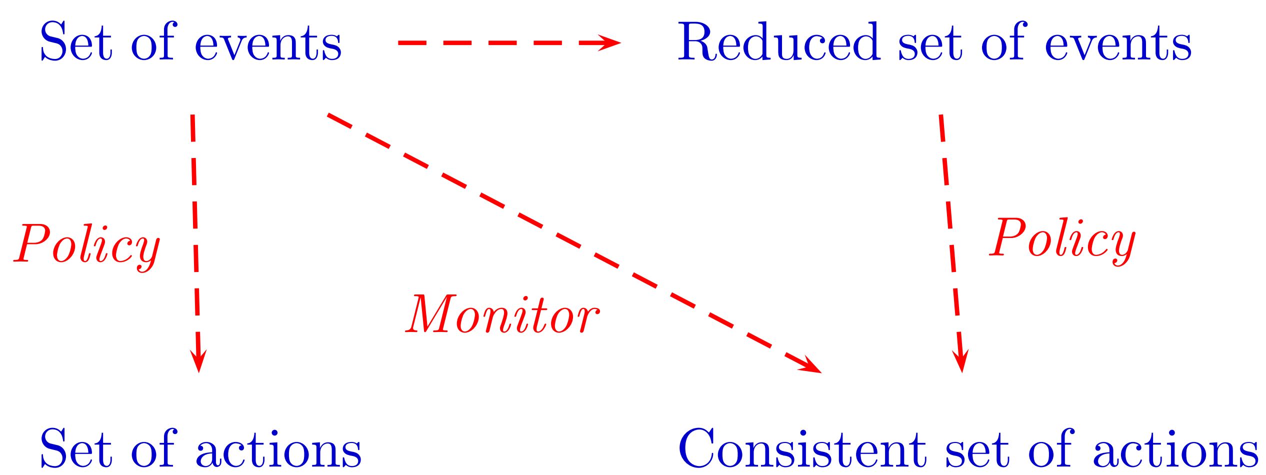
does not work for a slightly modified policy:

$$defectiveProduct \ \textbf{ causes } \ stop$$
$$orderReceived \ \textbf{ causes } \ mailProduct$$
$$orderReceived \ \textbf{ causes } \ chargeCreditCard$$
$$callCompleted \ \textbf{ causes } \ chargeCreditCard$$

# Unobtrusiveness

An unobtrusive monitor mimicks the policy on some subset $E'$ of input events:

- input event $e \in E'$: every action caused by $e$ succeeds
- input event $e \notin E'$: every action caused by $e$ fails unless it is also caused by an event in $E'$



# “Transactional” semantics for policies

**transaction $\equiv$ action:**

- actions independently blocked or delayed
- the user can detect a conflict
- implemented by action-based monitors

**transaction $\equiv$ event + actions caused:**

- actions connected through common events
- conflicts invisible to the user
- implemented by event-based monitors

# Highlights of the talk

1. syntax and semantics of PDL
2. definition of monitors (*event conjunction only*):
   - axiomatic
   - algorithmic
3. extensions:
   - negation, history-based policies
4. implementation
5. related work:
   - policies, events, rules, agents
6. conclusions and further work

# PDL: syntax

**Policy:** a set of **Event-Condition-Action** rules.

**Events:**

- application-defined
- atomic or composite
- atomic events can have attributes
- composite events: conjunction, negation, sequence, relax-sequence.

**Conditions:** built-in predicates for comparing attribute values.

**Actions** are uninterpreted: they correspond to arbitrary procedure calls.

# PDL: semantics

*Event conjunction and negation only.*

**Epoch:** a finite set of simultaneous input events.

The **semantics** of a policy $P$ is a mapping $\pi_P$ associating with every possible epoch a set of actions.

This mapping is specified using a translation to (a variant of) **Datalog**.

# Translation to Datalog

A PDL rule

$$e_1 \& \cdots \& e_n \ \textbf{causes} \ a \ \textbf{if} \ C(t_1, \ldots, t_k)$$

is translated to:

$$occ(e'_1) \wedge \cdots occ(e'_n) \wedge C(t'_1, \ldots, t'_k) \rightarrow exec(a(t'_1, \ldots, t'_k))$$

Event and term translation($e'_i$ and $t'_j$):

- attribute notation $\rightarrow$ positional
- negation elimination: $!e \rightarrow not\_e$

# Example

The PDL rule

$$requestRes\ \textbf{causes}\ processRes(requestRes.user)$$
$$\textbf{if}\ \ requestRes.user \neq intruder$$

is translated to

$$occ(requestRes(U)) \wedge U \neq intruder \rightarrow exec(processRes(U))$$

## How to define monitors?

**Axiomatically:** disjunctive logic programs.

**Algorithmically:** nondeterministic imperative programs.

# Axiomatic conflict resolution

Limitations:

- *event conjunction only.*

Cancellation monitors are defined by augmenting the Datalog translation of a policy by:

- **conflict** rules
- **blocking** rules (not needed for action cancellation)
- **accepting** rules.

The **accepted** actions are output.

# Conflict rules

Constraint

$$\textbf{never} \;\; a_1 \wedge \ldots \wedge a_n \;\textbf{if}\;\; C$$

is translated into the **conflict** rule:

$$exec(a_1) \wedge \ldots \wedge exec(a_n) \wedge C \rightarrow block(a_1) \vee \ldots \vee block(a_n)$$

Example:

$$\textbf{never} \;\; stop \wedge mailProduct$$

translated to

$$exec(stop) \wedge exec(mailProduct) \rightarrow block(stop) \vee block(mailProduct)$$

# Action cancellation

For each action $a$ occurring in a policy rule, there is an **accepting** rule:

$$exec(a) \wedge \neg block(a) \rightarrow accept(a)$$

# Event cancellation

Each policy rule of the form

$$e_1 \& \ldots \& e_n \ \textbf{causes} \ a \ \textbf{if} \ \ C$$

is translated into a **blocking** rule

$$occ(e_1) \wedge \ldots \wedge occ(e_n) \wedge block(a) \wedge C \rightarrow ignore(e_1) \vee \ldots \vee ignore(e_n)$$

and an **accepting** rule

$$occ(e_1) \wedge \ldots \wedge occ(e_n) \wedge C \wedge \neg ignore(e_1) \wedge \ldots \wedge \neg ignore(e_n) \rightarrow accept(a)$$

Policy translation:

$$occ(defectiveProduct) \longrightarrow exec(stop)$$

$$occ(orderReceived) \longrightarrow exec(mailProduct)$$

$$occ(orderReceived) \longrightarrow exec(chargeCreditCard)$$

Conflict rule:

$$exec(stop) \wedge exec(mailProduct) \longrightarrow block(stop) \vee block(mailProduct)$$

Blocking rules:

$$occ(defectiveProduct) \wedge block(stop) \longrightarrow ignore(defectiveProduct)$$

$$occ(orderReceived) \wedge block(mailProduct) \longrightarrow ignore(orderReceived)$$

$$occ(orderReceived) \wedge block(chargeCreditCard) \longrightarrow ignore(orderReceived)$$

Accepting rules:

$$occ(defectiveProduct) \wedge \neg ignore(defectiveProduct) \longrightarrow accept(stop)$$

$$occ(orderReceived) \wedge \neg ignore(orderReceived) \longrightarrow accept(mailProduct)$$

$$occ(orderReceived) \wedge \neg ignore(orderReceived) \longrightarrow accept(chargeCreditCard)$$

# Conjunction of events

The rule:

$$dial\ \&\ charge\ \ \textbf{causes}\ \ connect$$

is translated to:

$$occ(dial) \wedge occ(charge) \wedge block(connect) \rightarrow ignore(dial) \vee ignore(charge)$$

$$occ(dial) \wedge occ(charge) \wedge \neg ignore(dial) \wedge \neg ignore(charge) \rightarrow accept(connect)$$

# Correspondence result

**Theorem 1.**

For both action and event cancellation, every minimal model of the augmented Datalog translation specifies a maximal monitor of the policy (and vice versa).

**Maximal** monitor: preserves as many actions (events) as possible without violating constraints.

# Algorithm for action cancellation

**Algorithm 1** `Action Cancellation Monitor`

**begin**

$A := \emptyset$

$U := \pi_P(E)$

**while** true **do**

select $a \in U - A$ such that $A \cup \{a\} \models AC$

**if** select successful **then** $A := A \cup \{a\}$

**else** break

**end**

**end**

# Algorithm for event cancellation

**Algorithm 2** `Event Cancellation Monitor`

**begin**

$E' := \emptyset$

**while** true **do**

select $e \in E - E'$ such that $\pi_P(E' \cup \{e\}) \models AC$

**if** select successful **then** $E' := E' \cup \{e\}$

**else** break

**end**

$A := \pi_P(E')$

**end**

# Negation

Problems:

- a policy may fail to have a monitor at all
- ignoring events may trigger new actions

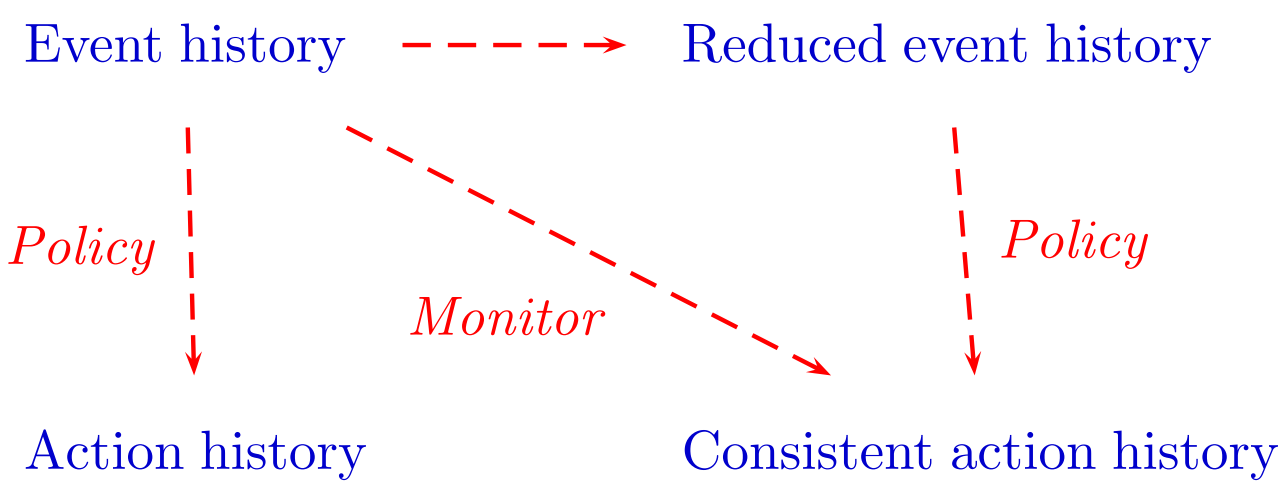
Solution:

- ignoring an event makes it undefined

# History-based policies

Temporal dimensions:

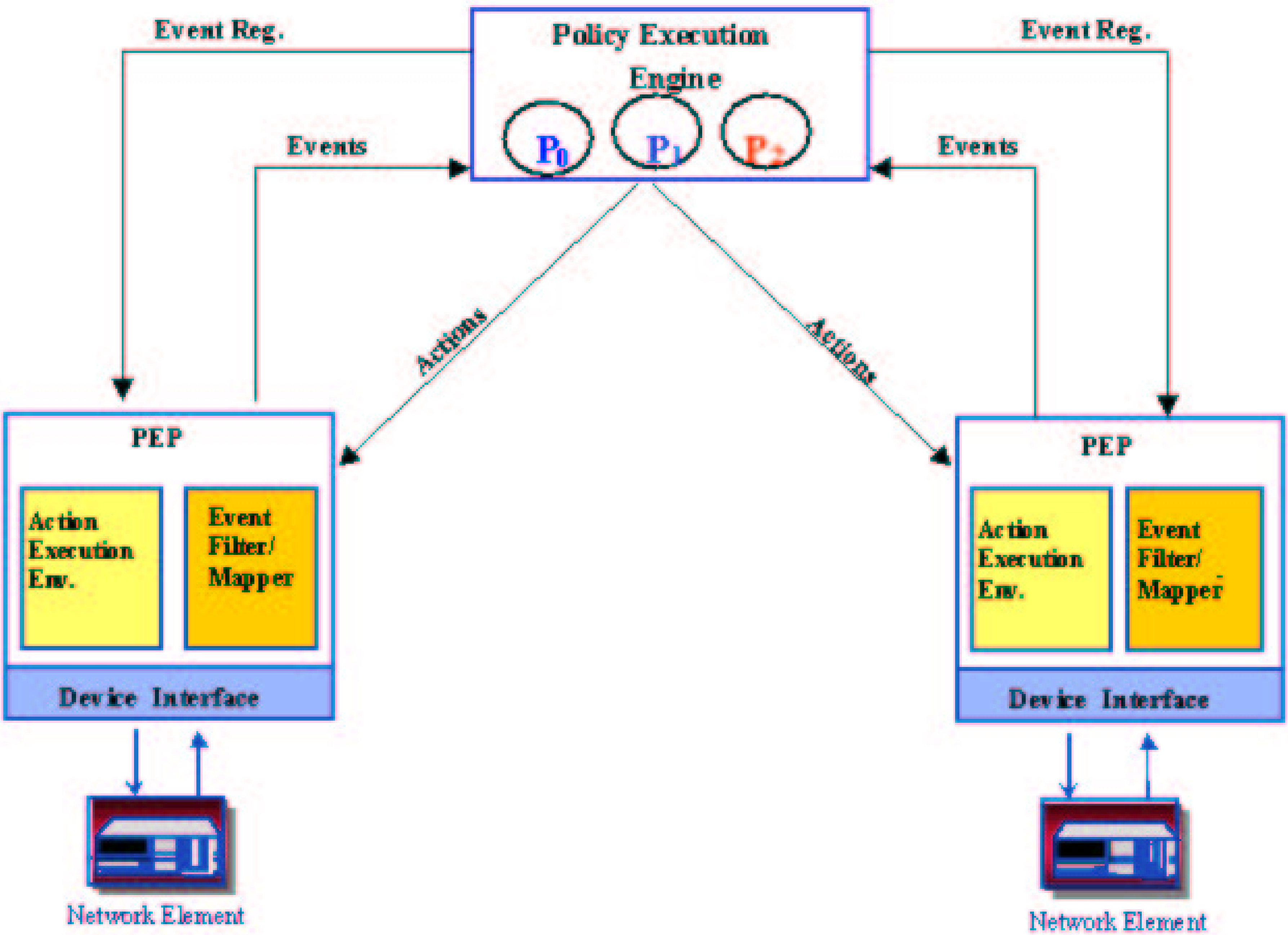
- **sequence** events
- **delay** monitors
- **temporal** action constraints



# Monitors for history-based policies

Axiomatic approach: $Datalog_{1S}$.

Algorithmic approach: easy extensions.



Policy Execution Engine
$P_0$
$P_1$
$P_2$
Event Reg.
Event Reg.
Events
Events
Actions
Actions
PEP
Action Execution Env.
Event Filter/ Mapper
Device Interface
Network Element
PEP
Action Execution Env.
Event Filter/ Mapper
Device Interface
Network Element

# Related work

**Event notification** systems and languages:

- rich syntax (events), informal semantics
- explicit conflict resolution only recently addressed
- typically static, not dynamic, conflicts

**Production rules:**

- resolution of **rule conflicts**
- interpreted actions
- simple event model
- meta-language for controlling rule executions [Jagadish, Mendelzon, Mumick, PODS'96].

# Agent-based systems

Policies are simple **reactive agents**.

**Commands** can be viewed as **events**.

**Action constraints** determined by the physical or virtual environment.

**Unobtrusiveness:** no partially executed commands.

[Eiter, Subrahmanian, Pick, AI Journal, 1999]:

- deontic specifications of agent systems
- action cancellation monitors (w/o logical framework)
- only atomic events

# Unobtrusive agents

Waiter agent:

$pour\_request$ **causes** $hold\_cup$

$pour\_request$ **causes** $tilt\_bottle$

$serve\_request$ **causes** $hold\_plate$

**never** $hold\_cup \wedge hold\_plate$

What happens if *pour_request* and *serve_request* arrive simultaneously?

# Conclusions

A general formal framework for defining policy monitors:

- broad classes of policies and monitors
- results optimal: maximal monitors
- extensible:
  - negation
  - history-based policies

# Further work

Policy **analysis**:

- conflict-freeness

**More general** classes of policies:

- arbitrary event iteration
- temporal aggregation
- complex, long-duration actions
- preferences, deontic notions
- databases, XML documents
- agents: coordination, communication

Papers:

1. J. Lobo, R. Bhatia, S. Naqvi, *"A Policy Description Language."* AAAI'99.

2. J. Chomicki, J. Lobo, S. Naqvi, *"A Logic Programming Approach to Conflict Resolution in Policy Management."* KR'2000.

3. J. Chomicki, J. Lobo, *"Monitors for History-Based Policies."* POLICY'2001, January 2001, Bristol, UK.