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:

\[
\text{arrivedFaxOff \ causes sendFaxJoeOff (arrivedFaxOff.content)} \quad \text{if} \quad \text{arrivedFaxOff.from} = \text{“Chicago”}.
\]
Action conflicts

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

Example:

\[ requestRes \textbf{causes} processRes(requestRes.user) \]

Two simultaneous reservation requests cannot both be satisfied:

\[ \textbf{never } processRes(\text{User}_1) \land processRes(\text{User}_2) \textbf{ if } \text{User}_1 \neq \text{User}_2. \]

Reserved resources: bandwidth, airport runway,...
Action constraints

Syntax:

\textbf{never} \quad A_1 \land A_2 \land \cdots \land A_n \textbf{ if } C

Logical reading:

\forall \neg (A_1 \land A_2 \land \cdots \land A_n \land C)
Conflict resolution

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

Set of events

Policy

Monitor

Set of actions Consistent set of actions
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:

\[\text{defectiveProduct causes stop}\]
\[\text{orderReceived causes mailProduct}\]
\[\text{orderReceived causes chargeCreditCard}\]

Constraint:

\[\text{never } stop \land mailProduct\]

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

\textbf{never} \hspace{0.2cm} \texttt{stop} \land \texttt{chargeCreditCard}

does not work for a slightly modified policy:

\texttt{defectiveProduct} \hspace{0.2cm} \textbf{causes} \hspace{0.2cm} \texttt{stop}

\texttt{orderReceived} \hspace{0.2cm} \textbf{causes} \hspace{0.2cm} \texttt{mailProduct}

\texttt{orderReceived} \hspace{0.2cm} \textbf{causes} \hspace{0.2cm} \texttt{chargeCreditCard}

\texttt{callCompleted} \hspace{0.2cm} \textbf{causes} \hspace{0.2cm} \texttt{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 $E'$

\[ \begin{array}{c}
\text{Set of events} \
\text{Policy} \\
\text{Monitor} \\
\text{Set of actions}
\end{array} \rightarrow \begin{array}{c}
\text{Reduced set of events} \\
\text{Policy} \\
\text{Consistent set of actions}
\end{array} \]
“Transactional” semantics for policies

\textbf{transaction} \equiv \textbf{action}:

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

\textbf{transaction} \equiv \textbf{event} + \textbf{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


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

\[ e_1 \& \cdots \& e_n \text{ causes } a \text{ if } C(t_1, \ldots, t_k) \]

is translated to:

\[ \text{occ}(e'_1) \land \cdots \land \text{occ}(e'_n) \land C(t'_1, \ldots, t'_k) \rightarrow \text{exec}(a(t'_1, \ldots, t'_k)) \]

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

- attribute notation \(\rightarrow\) positional
- negation elimination: \(!e \rightarrow \text{not}_e\)
Example

The PDL rule

\[ \textit{requestRes causes processRes(requestRes.user)} \]
\[ \textbf{if } \textit{requestRes.user } \neq \textit{intruder} \]

is translated to

\[ \text{occ(requestRes(U))} \land U \neq \textit{intruder} \rightarrow \text{exec(processRes(U))} \]
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

\[
\text{never } a_1 \land \ldots \land a_n \text{ if } C
\]

is translated into the conflict rule:

\[
\text{exec}(a_1) \land \ldots \land \text{exec}(a_n) \land C \rightarrow \text{block}(a_1) \lor \ldots \lor \text{block}(a_n)
\]

Example:

\[
\text{never } \text{stop} \land \text{mailProduct}
\]

translated to

\[
\text{exec}(\text{stop}) \land \text{exec}(\text{mailProduct}) \rightarrow \text{block}(\text{stop}) \lor \text{block}(\text{mailProduct})
\]
Action cancellation

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

$$exec(a) \land \neg block(a) \rightarrow accept(a)$$
Event cancellation

Each policy rule of the form

\[ e_1 & \cdots & e_n \text{ causes a if } C \]

is translated into a \textbf{blocking} rule

\[ \text{occ}(e_1) \land \cdots \land \text{occ}(e_n) \land \text{block}(a) \land C \rightarrow \text{ignore}(e_1) \lor \cdots \lor \text{ignore}(e_n) \]

and an \textbf{accepting} rule

\[ \text{occ}(e_1) \land \cdots \land \text{occ}(e_n) \land C \land \neg \text{ignore}(e_1) \land \cdots \land \neg \text{ignore}(e_n) \rightarrow \text{accept}(a) \]
Policy translation:

\[
\text{occ(defectiveProduct)} \rightarrow \text{exec(stop)} \\
\text{occ(orderReceived)} \rightarrow \text{exec(mailProduct)} \\
\text{occ(orderReceived)} \rightarrow \text{exec(chargeCreditCard)}
\]

Conflict rule:

\[
\text{exec(stop)} \land \text{exec(mailProduct)} \rightarrow \text{block(stop)} \lor \text{block(mailProduct)}
\]

Blocking rules:

\[
\text{occ(defectiveProduct)} \land \text{block(stop)} \rightarrow \text{ignore(defectiveProduct)} \\
\text{occ(orderReceived)} \land \text{block(mailProduct)} \rightarrow \text{ignore(orderReceived)} \\
\text{occ(orderReceived)} \land \text{block(chargeCreditCard)} \rightarrow \text{ignore(orderReceived)}
\]

Accepting rules:

\[
\text{occ(defectiveProduct)} \land \neg \text{ignore(defectiveProduct)} \rightarrow \text{accept(stop)} \\
\text{occ(orderReceived)} \land \neg \text{ignore(orderReceived)} \rightarrow \text{accept(mailProduct)} \\
\text{occ(orderReceived)} \land \neg \text{ignore(orderReceived)} \rightarrow \text{accept(chargeCreditCard)}
\]
Conjunction of events

The rule:

\[ \text{dial} \& \text{charge} \text{ causes } \text{connect} \]

is translated to:

\[ \text{occ}(\text{dial}) \land \text{occ}(\text{charge}) \land \text{block}(\text{connect}) \rightarrow \text{ignore}(\text{dial}) \lor \text{ignore}(\text{charge}) \]

\[ \text{occ}(\text{dial}) \land \text{occ}(\text{charge}) \land \lnot \text{ignore}(\text{dial}) \land \lnot \text{ignore}(\text{charge}) \rightarrow \text{accept}(\text{connect}) \]
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

Event history \[\rightarrow\] Reduced event history

Action history \[\rightarrow\] Consistent action history
Monitors for history-based policies

Axiomatic approach: $Datalog_{1S}$.

Algorithmic approach: easy extensions.
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:

\[ \text{pour}_\text{request} \text{ causes } \text{hold}_\text{cup} \]
\[ \text{pour}_\text{request} \text{ causes } \text{tilt}_\text{bottle} \]
\[ \text{serve}_\text{request} \text{ causes } \text{hold}_\text{plate} \]

\text{never } \text{hold}_\text{cup} \land \text{hold}_\text{plate}

What happens if \text{pour}_\text{request} and \text{serve}_\text{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:

