A Logic Programming Approach to Conflict Resolution in Policy Management

Jan Chomicki*
Monmouth University
chomicki@monmouth.edu

Jorge Lobo, Shamim Naqvi
Bell Labs
{jlobo,shamim}@research.bell-labs.com

Abstract

The simple *event-condition-action* (ECA) rule paradigm of active databases has proved very useful in many AI and database applications. However, its applicability goes beyond data management. ECA rules can be used in network management and monitoring, electronic commerce, security and access management, and other application areas to express *policies* – collections of general principles specifying the desired behavior of a system. In this paper we use a declarative policy description language $\mathcal{PDL}$, in which policies are formulated as sets of ECA rules. The main contribution of the paper is a framework for detecting action conflicts and finding resolutions to these conflicts. Conflicts are captured as violations of action constraints. The semantics of rules, and conflict detection and resolution are defined axiomatically using logic programs. Given a policy and a set of action constraints the framework defines a monitor that filters the output of the policy to satisfy the constraints. We briefly describe the architecture of a $\mathcal{PDL}$-based policy server being used to provide centralized administration of a soft switch in a communication network and show how it can be augmented to handle conflict resolution.

1 Introduction

The simple *event-condition-action* rule paradigm has proved very useful in many AI and database applications [23, 4], from constraint maintenance to the general encoding of expert rules. However, the applicability of the *event-condition-action* rule paradigm goes beyond data management or expert systems. Such rules can be used in network management and monitoring [11], electronic commerce [9], security and access management [17], and other application areas to express *policies* – collections of general principles specifying the desired behavior of a system. For instance, network management is mainly carried out by following policies about the behavior of the resources in the network. The policies are usually formulated as sets of low-level rules that describe how to (re)configure a device or how to manipulate the different network elements under different network conditions. Analogous policies occur in areas such as electronic commerce ("orders from established customers should receive higher priority") and computer security. Usually, policies are coded in an imperative programming language like Java. This makes for implementation ease and efficiency but limits what can be done with policies. For instance, it is difficult to modify, verify or analyze such policies. In this paper, we pursue a different approach. We use a declarative *policy description language* $\mathcal{PDL}$ [15], in which policies are formulated as sets of *event-condition-action* rules of the form

$$\textit{event}\ \textbf{causes}\ \textit{action}\ \textbf{if}\ \textit{condition}. \qquad (1)$$

A policy rule can be read as: if the *event* occurs in a situation where the *condition* is true, then the *action* is executed. A policy in $\mathcal{PDL}$ defines a *transducer:* a function that maps sets of events into sets of actions. The $\mathcal{PDL}$ policy manager provides an implementation for such transducers.

A formal description of the syntax and semantics of $\mathcal{PDL}$ can be found in Section 2. However, our interest in this paper is not in the language per se but in controlling policies written in the language. In particular, we address the issues of *conflict detection* and *resolution.*

Example 1 Consider the following simple scenario:

* Work done while visiting Bell Labs.

There is an automatic reservation system for the conference room in a department. The reservations are controlled by a simple first-in-first-served policy captured by the following rule:

$$\textit{requestRes}(\textit{User}) \ \textbf{causes} \ \textit{processRes}(\textit{User}). \qquad (2)$$

The procedure *processRes* reserves the room for the user if it is available. Notice, however, that if there are simultaneous reservation requests from different users, there will be two instances of the rule triggered and the policy will not work unless one of the *processRes* calls is cancelled. We introduce *action constraints* into $\mathcal{PDL}$ to capture this type of conflicts. The constraints describe under which circumstances a set of actions cannot be executed simultaneously. For the reservation policy the appropriate action constraint is

$$\textbf{never} \ \ \textit{processRes}(U_1) \wedge \textit{processRes}(U_2) \ \textbf{if} \ \ U_1 \neq U_2.$$

Typically the burden of conflict resolution is left in the hands of the policy administrator, who must provide the resolution of the conflicts in the code that implements the policies. The process is *ad hoc* with no guarantees on the properties of the solution.

This paper introduces a formal framework for detecting rule conflicts and finding resolutions to these conflicts. Given a policy and a set of constraints on the concurrent execution of actions, the framework produces a monitor for the policy. A policy monitor filters the output of a policy (i.e., a set of actions) by cancelling some actions to obtain a result consistent with the constraints. However, actions cannot be cancelled arbitrarily. First, we would like to cancel as few actions as possible. Second, we propose that policy monitors should be *unobtrusive:* For any set of input events E, an unobtrusive monitor outputs the same set of actions as the original policy applied to a *subset* E' of E. Thus, if an event e is in E', every action caused by e succeeds. If an event e is in $E \setminus E'$, every action caused by e fails unless it is also caused by a different event which is in E'. This is analogous to the *atomicity* requirement for standard database transactions. In our context, it means that the execution of an unobtrusive monitor corresponds to *some* conflict-free execution of the original policy.

In this paper, the semantics of policy rules, and conflict detection and resolution are defined axiomatically using disjunctive logic programs. Such programs are generated automatically from $\mathcal{PDL}$ specifications and can subsequently be modified by the user. The latter option allows for more flexibility in handling conflicts. The logic programming formulation has many desirable properties: It is both executable (using well-established logic programming techniques) and easy to modify and analyze formally.

The plan of the paper is as follows. In Section 2, we define the syntax and semantics of $\mathcal{PDL}$ (the latter using a translation to Horn rules). In Section 3, which constitutes the core of the paper, we address the issue of conflict detection and resolution using a logic programming framework. The construction described deals with unobtrusive monitors. In this section we restrict the monitors to policies without negative events (i.e. a policy cannot be based on the absence of an event). In Section 4, we study relevant computational complexity issues. In Section 5, we discuss the problems of having negated events and present a solution. In Section 6, we show how the unobtrusiveness requirement can be partially relaxed using an elaboration of our basic framework. In Section 7, we outline the existing implementation of a $\mathcal{PDL}$ server and show how it can be augmented to support policy monitors. In Section 8, we survey related work. In Section 9, we conclude and outline the directions for further research. Due to space limitations the examples used are simplified and only selected proofs are given.

2 The policy description language $\mathcal{PDL}$

2.1 Syntax

The language we consider consists of three basic classes of symbols: *primitive event* symbols, *action* symbols and *constant* symbols. These symbols are system-dependent and are given to the user that defines the policies. There is also a set of standard ordered types such as integers, floats, character strings, etc. Action and primitive event symbols may be of any nonnegative arity. Each action symbol of arity n denotes the name of a procedure that takes n arguments (also called parameters) of a particular type. Every event argument and constant symbol also has an associated type. The arguments of event symbols represent event attributes.

Definition 1 A *policy* is a finite collection of well-typed policy rules of the form (1), where the *event*, *action* and *condition* parts of a rule are defined below.

Definition 2 The *event* part of a policy rule is an expression of the form $e_1 \& \ldots \& e_n$ where each e_i is an *event literal*. An *event literal* is either a typed *event term* of the form $e(t_1, \ldots, t_n)$, where e is a primitive event symbol of n arguments and each t_i is a constant or a variable, or a primitive event symbol e preceeded by ! (!e) representing the negation of e. An *event instance* is a ground event term.

Definition 3 The *action* part of a policy rule is a typed *action term* of the form $a(t_1, \ldots, t_n)$, where a is an action symbol of n arguments and each t_i is a variable that appears in the event part of the rule or a constant.

Definition 4 The *condition* part of a policy rule is an expression of the form $p_1, \ldots, p_n$, where each p_i is a predicate of the form $t_1 \theta t_2$, θ is a relation operator from the set $\{=, \neq, <, \leq, >, \geq\}$ and each t_i is either a variable that appears in the event part or a constant. The condition represents the conjunction of the predicates.

2.2 Semantics

We adopt the view that a policy expects as an input a set of instances of primitive events assuming the events in the set occurred simultaneously. (The granularity of time is application dependent: It may be a minute in some contexts, a day - in others. We assume that the selection of the time granularity is made outside of our policy framework.)

Definition 5 A finite set of event instances is an *epoch*. The set of all possible epochs is denoted by *Epochs*. We say that the event e *occurs* in an epoch if an instance of the event term $e(X_1, \ldots, X_n)$ is a member of the epoch. We say that the event literal !e *occurs* in an epoch (with a single instance) if there are no instances of the primitive event e in the epoch. The set of all ground action terms is denoted by *Actions* and its finite subsets are called *action sets*.

In the following, assume P is a policy. Formally, the semantics of P is given by a transducer $\pi_P : \mathit{Epochs} \longrightarrow 2^{\mathit{Actions}}$. We define this transducer using a Horn logic program Π_P of a special form. The minimal model of this program, which can be computed using well-known logic programming techniques, represents the transducer defined by the policy. To compute the actions that are triggered by an epoch, we transform the epoch into a set of ground atoms and add them to the program Π_P. The actions will appear in the minimal model of the expanded program. For an epoch E, let

$$\begin{aligned} \mathit{occ}(E) = \quad & \{\mathit{occ}(g) | g \in E\} \cup \\ & \{\mathit{occ}(!e) | e \text{ has no instances in } E\}. \end{aligned}$$

The intuitive meaning of $\mathit{occ}(e(t_1, \ldots, t_n))$ is that the instance $e(t_1, \ldots, t_n)$ of the primitive event e *occurred* in the current epoch. Then, each rule of the form

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

in the policy is translated into the implication

$$\mathit{exec}(a) \leftarrow \mathit{occ}(e_1) \wedge \ldots \wedge \mathit{occ}(e_l) \wedge C.$$

We denote by Π_P the set of rules that result from the translation of each policy rule in P. It is easy to see that Π_P is a non-recursive Horn logic program.

Definition 6 $\pi_P : \mathit{Epochs} \longrightarrow 2^{\mathit{Actions}}$ is the transducer defined by P if for every epoch E, $a \in \pi_P(E)$ iff $\Pi_P \cup \mathit{occ}(E) \models \mathit{exec}(a)$.

By the virtue of Π_P being a non-recursive Horn program, the evaluation of π_P can be done in time linear with respect to the size of the input epoch.

3 Conflict detection and resolution

3.1 Action constraints and monitors

A policy generates a conflict when its output contains a set of actions that the policy administrator has specified *cannot occur together*. The conflicts are captured as violations of action constraints.

Definition 7 An *action constraint* is an expression of the form

$$\textbf{never} \quad a_1 \wedge \ldots \wedge a_m \textbf{ if } \quad C.$$

Each a_i is an action term and C is a condition such that variables appearing in C also appear in one of the a_is. The meaning is: *"never allow the simultaneous execution of the actions $a_1, \ldots, a_m$ if the condition C holds."* It formally represents the formula

$$\forall \neg (a_1 \wedge \ldots \wedge a_n \wedge C).$$

Definition 8 Given an action set S consisting of ground action terms, we say that S *satisfies* an action constraint ac (resp. a set of action constraints AC) if S is a model of ac (resp. of all the constraints in AC) in the standard model theoretic sense (with action terms viewed as literals).

A policy monitor of a set of action constraints generates only action sets without conflicts, i.e., satisfying all given action constraints.

Definition 9 Given a set of action constraints AC, a *policy monitor* ω_{AC} is a transducer $\omega_{AC} : \mathit{Epochs} \longrightarrow 2^{\mathit{Actions}}$ such that for every epoch E, $\omega_{AC}(E)$ satisfies AC.

Note that a monitor, being a transducer, is semantically identical to a policy. However, unlike policies, monitors will not be defined in $\mathcal{PDL}$ but rather specified indirectly using logic programs of a special form (defined later in this section).

Next, we identify several important properties of policy monitors. They capture the intuition that monitors should be chosen to behave as close as possible to the policy which output they filter and that the effect of conflict resolution should be maximally transparent to the user.

In the following definitions, assume that P is a policy and AC is a set of action constraints. They will be omitted when they are clear from the context.

Definition 10 An epoch E is P-*consistent* with AC if $\pi_P(E)$ satisfies AC. A (P, AC)-*consistent reduction* E' of an epoch is an epoch such that $E' \subseteq E$ and E' is P-consistent with AC. The reduction is *maximal* if there is no (P, AC)-consistent reduction E'' of E such that $E' \subseteq E''$ and $E' \neq E''$.

Definition 11 A policy monitor ω_{AC} is:

1. a *conservative* monitor of P if it is identical to the policy for P-consistent epochs.

2. an *action-cancellation* monitor of P if does not generate any actions beyond those of P, i.e., for every epoch E, $\omega_{AC}(E) \subseteq P(E)$.

3. a *maximal action-cancellation* monitor of P if for every epoch E, $\omega_{AC}(E)$ is a maximal subset of $P(E)$ that satisfies AC.

4. an *event-cancellation* monitor of P if for every epoch E, there exists a (P, AC)-consistent reduction E' of E such that $\omega_{AC}(E) = P(E')$.

5. a *maximal event-cancellation* monitor of P if for every epoch E, there exists a maximal (P, AC)-consistent reduction E' of E such that $\omega_{AC}(E) = P(E')$.

Conservativeness is a basic requirement that all monitors should satisfy. All the monitors described in this paper work by cancelling actions to eliminate conflicts. It is easy to see that in the absence of negated events every event-cancellation monitor is also an action-cancellation monitor. However, there is a difference between action- and event-cancellation monitors. The latter satisfy additionally a property that we call *unobtrusiveness*. To appreciate the importance of unobtrusiveness, consider the following example.

Example 2 Assume we have the policy P_1

defectiveProduct **causes** *stop*
orderReceived **causes** *mailProduct*
orderReceived **causes** *chargeCC*

and the constraint

$$\textbf{never} \quad stop \wedge mailProduct.$$

If the events *defectiveProduct* and *orderReceived* occur together, the resulting conflict may be eliminated by cancelling the action *mailProduct*. However, the action *chargeCC* can still be executed without conflict, although this is intuitively incorrect (at least from the customer's point of view!). An event-cancellation monitor avoids this problem by ignoring the event *orderReceived* and therefore also indirectly cancelling both actions it causes.

Example 3 To see the need for action-cancellation monitors which are not unobtrusive, consider the rule

$$request(X) \ \textbf{causes} \ acknowledge(X).$$

If there are other rules involving the *request* event that lead to conflict and as the result some other actions caused by this event are cancelled, the above rule should still be executed. Thus, the *request* event cannot simply be ignored.

The next question to ask is if we can forgo event-cancellation and only work with action-cancellation monitors. If we go back to Example 2, at first glance it seems that we could make an explicit connection between *mailProduct* and *chargeCC* to solve the problem. One possibility would be to have rules with multiple actions such as

orderReceived **causes** *mailProduct*, *chargeCC*

and when one of the two actions is cancelled the other should also be cancelled. A similar effect can be achieved by adding the constraint

$$\textbf{never} \quad stop \wedge chargeCC.$$

However, both of those approaches may unnecessarily or even incorrectly cause the cancellation of an action. This may occur in the example above if there is another event also causing *chargeCC*. We could, for example, have the rule

serviceCallCompleted **causes** *chargeCC*.

If the event *serviceCallCompleted* occurs simultaneously with *defectiveProduct* and *orderReceived*, an action-cancellation monitor will not charge for the service call.

A comprehensive policy management system should provide both unobtrusive and non-unobtrusive monitors, as well as their combinations. In this paper, we concentrate first on *event-cancellation monitors* which are unobtrusive. Afterwards we show how to combine unobtrusive and not unobtrusive monitors for different parts of a policy.

3.2 Translation to logic programs

Here we define policy monitors for policies that do not involve negated events by extending the Datalog translation of policies, defined in the previous section. The idea is to capture conflict resolution *at the level of rule execution.* In addition to the two predicate symbols *occ* and *exec*, we introduce three new predicate symbols: *block* (an action), *ignore* (an event), and *accept* (an action).

Conflict rules Ψ_{AC}. Each constraint in AC of the form

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

is translated into a *conflict* rule

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

Blocking rules B_P and accepting rules A_P. Each policy rule in P of the form

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

is translated into a *blocking* rule

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

and an *accepting* rule

$$accept(a) \leftarrow \quad occ(e_1) \wedge \ldots \wedge occ(e_n) \wedge C \wedge \\ \neg ignore(e_1) \wedge \ldots \wedge \neg ignore(e_n).$$

Example 4 The constraint

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

is translated to

$$block(stop) \vee block(mailProduct) \leftarrow \\ exec(stop) \wedge exec(mailProduct).$$

Example 5 The constraint

$$\textbf{never}\ \ processRes(U_1) \wedge processRes(U_2) \ \textbf{if}\ \ U_1 \neq U_2$$

is translated to

$$block(processRes(U_1)) \vee block(processRes(U_2)) \\ \leftarrow \ exec(processRes(U_1)) \wedge exec(processRes(U_2)) \\ \wedge U_1 \neq U_2.$$

Example 6 The rule

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

is translated to

$$ignore(dial) \vee ignore(charge) \leftarrow \\ occ(dial) \wedge occ(charge) \wedge block(connect) \\ accept(connect) \leftarrow \\ occ(dial) \wedge occ(charge) \wedge \\ \neg ignore(dial) \wedge \neg ignore(charge).$$

Remarks: Note that all the rules above are *safe*. That is, variables that appear in the consequents of the implications or in negated literals in the antecedent also appear in literals with no negation in the antecedents. The rules are not, strictly speaking, in Datalog, since actions and events are encoded as function symbols. However, those symbols are not nested and thus the resulting program can easily be translated into Datalog by introducing new predicate symbols. Also, for a policy P and a set of constraints AC, the program $\Pi_P \cup \Psi_{AC} \cup A_P \cup B_P$ is a *hierarchical (i.e. non-recursive) disjunctive logic program.*

Theorem 1 For any epoch E, E' is a maximally (P, AC)-consistent reduction of E iff $E' = \{e | occ(e) \in M \wedge ignore(e) \notin M\}$ for some minimal model M of $\Pi_P \cup \Psi_{AC} \cup A_P \cup B_P \cup occ(E)$.

This theorem provides us with sufficient information to build a maximal event-cancellation monitor. We only need any of the standard algorithms to compute the minimal models [21] of disjunctive logic programs.

Definition 12 Let $\Omega_{P,AC} : Epochs \longrightarrow 2^{Actions}$ be a transducer defined for any epoch E in the following way:

1. Select a minimal model M of $\Pi_P \cup \Psi_{AC} \cup A_P \cup B_P \cup occ(E)$.
2. Return the set $\{a | accept(a) \in M\}$ as output (i.e. $\Omega_{P,AC}(E) = \{a | accept(a) \in M\}$).

Corollary 1 $\Omega_{P,AC}$ is a maximal event-cancellation monitor of P. Moreover, all maximal event-cancellation monitors of P can be obtained in this way.

It is also immediate to see that $\Omega_{P,AC}$ is conservative (if no conflicts occur, no actions are blocked and no events ignored).

3.3 Tuning the translation

The logic program translation defined above encodes all maximal event-cancellation monitors that can be associated with a policy. However, in real situations there are monitors that can be considered more appropriate than others. When there is a choice of blocking one of several actions to resolve a conflict, the application domain may suggest a *priority ordering* among the actions.

Example 7 Let's return to Example 2. The constraint says that we can never execute simultaneously the *stop* and *mailProduct* actions. Naturally, if the product is defective we should block *mailProduct* and let *stop* proceed. In other words, we give *priority* to *stop* over *mailProduct*. Priority can be encoded by changing the conflict rule to

$$block(mailProduct) \leftarrow exec(stop) \wedge exec(mailProduct)$$

eliminating the disjunction. Thus, for each constraint the user can select the action with the least priority and remove all the others from the head of the corresponding conflict rule. User choices for different constraints need to be coordinated to make sure the resulting monitor is still maximal. How to do it is beyond the scope of this paper.

A similar situation arises when choosing events to ignore, since in many cases it is impossible, incorrect, or undesirable to ignore certain events, e.g., time events. We call such events *persistent.*

Example 8 In Example 6 the *dial* event is persistent. Therefore, the blocking rule should be changed to

$$ignore(charge) \leftarrow occ(dial) \wedge occ(charge) \wedge block(connect).$$

In general, users can selectively remove ignored events from the consequent of the blocking rules to make them persistent until at least one is left.

4 Computational complexity

Assuming a class $\mathcal{C}$ of policy monitors, a fixed $\mathcal{PDL}$ policy P and a fixed set AC of action constraints, we study the complexity of simulating monitors from $\mathcal{C}$ as a function of the size of a given input epoch E and a given set of actions A. The simulation consists of determining whether $A \subseteq \omega_{AC}(E)$ for some monitor ω_{AC} from $\mathcal{C}$.

Theorem 2 Simulating maximal action-cancellation monitors can be done in PTIME.

Proof: The simulation returns *yes* iff $A \subseteq P(E)$ and A satifies AC. If this is the case, one can obtain in PTIME the action set $\omega_{AC}(E)$ for some maximal action-cancellation monitor ω_{AC} using a greedy approach: Start with A and keep adding actions in $P(E) - A$ until you obtain a set that cannot be further expanded without violating the constraints AC.

□

Theorem 3 Simulating maximal event-cancellation monitors is NP-complete.

Proof:

Upper bound. From the general results about the complexity of minimal models for Disjunctive DATALOG programs, it follows that the simulation problem is in Σ_2^p [6]. This bound can be sharpened to NP as follows. First, note that as soon as the set of events to be ignored is determined, the set of accepted actions can be computed by a single application of accepting rules and thus in PTIME. Second, for a given epoch the set S of *occ* and *exec* atoms can be computed in linear time. Once this is done, we take the ground instantiation of the entire logic program and keep only those rules whose all *occ* and *exec* atoms in the body are in S and (ground) conditions are true. For those rules the atoms in S and the conditions are erased. We are left thus with two kinds of clauses:

$$b_1 \vee \cdots \vee b_n$$

and

$$i_1 \vee \cdots \vee i_m \vee \neg b_0$$

where b_i are *block* atoms and i_j are *ignore* atoms. Moreover, for a fixed policy and a fixed set of action constraints the number of literals in those clauses is bounded. By exhaustively resolving the clauses of the first kind with those of the second, we obtain an equivalent set S^+ of polynomial size, containing only positive clauses. For any guessed set I of *ignore* atoms it can be verified in PTIME if it forms, together perhaps with some *block* atoms, a minimal model of S^+. Similarly, it can be checked in PTIME whether, given I, the set of accepted actions contains the given set of actions A.

Lower bound. Reduction from 3-COLORABILITY. We need four unary event symbols (for the different colors): r, b and g and one binary event symbol *arc*.

The policy P_0 is defined as follows:

$$
\begin{array}{l}
r(X)\&b(X)\ \textbf{causes}\ fail\\
r(X)\&g(X)\ \textbf{causes}\ fail\\
b(X)\&g(X)\ \textbf{causes}\ fail\\
arc(X,Y)\&r(X)\&r(Y)\ \textbf{causes}\ fail\\
arc(X,Y)\&b(X)\&b(Y)\ \textbf{causes}\ fail\\
arc(X,Y)\&g(X)\&g(Y)\ \textbf{causes}\ fail\\
r(X)\ \textbf{causes}\ ok(X)\\
b(X)\ \textbf{causes}\ ok(X)\\
g(X)\ \textbf{causes}\ ok(X)\\
arc(X,Y)\ \textbf{causes}\ ok\_arc(X,Y).
\end{array}
$$

The set of action constraints AC_0 consists of the single action constraint

$$\textbf{never}\quad fail.$$

A graph $G = (V, E)$ is 3-colorable iff there exists a maximal (P_0, AC_0)-consistent reduction E' of the epoch

$$
\begin{array}{l}
\{r(v)|v \in V\} \cup \{b(v)|v \in V\} \cup \{g(v)|v \in V\}\\
\cup\{arc(u,v)|(u,v) \in E\}
\end{array}
$$

such that $P_0(E')$ contains

$$\{ok(v)|v \in V\} \cup \{ok\_arc(u,v)|(u,v) \in E\}.$$

□

We would like to remark that if the set of actions A is fixed, it can be shown that the simulation problem for event-cancellation monitors is in PTIME.

Tuning the translation of $\mathcal{PDL}$ policy monitors, as described above, reduces the number of disjuncts in clause heads. If we are left with a (non-recursive) logic program, the simulation problem is in LOGSPACE.

5 Negated events

There are several problems with extending our approach to policies with negated events. First, in the presence of negated events, policies may not have *any* policy monitors.

Example 9 Consider the following policy P_3:

$$
\begin{array}{l}
e_1\ \textbf{causes}\ a_0\\
!e_1\ \textbf{causes}\ a_1\\
e_1\ \textbf{causes}\ a_2\\
!e_1\ \textbf{causes}\ a_3
\end{array}
$$

and the constraints $\{\textbf{never}\quad a_0 \wedge a_2, \textbf{never}\quad a_1 \wedge a_3\}$. If e_1 occurs there is a conflict. If it is ignored there is also a conflict.

Even more complications arise if the events have attributes since two instances of the same event can occur simultaneously and ignoring one instance does not imply ignoring the second instance. In general, testing consistency is computationally difficult.

Theorem 4 Determining the existence of an event-cancellation monitor for a given set of $\mathcal{PDL}$ rules (where events have no attributes) and a given epoch is co-NP-complete.

The second problem is due to the *nonmonotonic* character of negation. Our monitors work by action cancellation. In the presence of negation, cancelling an action and ignoring an event causing it may trigger some new actions.

Example 10 The following policy P_4:

$$
\begin{array}{l}
!e_1\ \textbf{causes}\ a_1\\
e_1\ \textbf{causes}\ a_2\\
e_1\ \textbf{causes}\ a_3
\end{array}
$$

and the constraint $\{\textbf{never}\quad a_2 \wedge a_3\}$ has a maximal event-cancellation monitor: one that outputs a_1 for every epoch. However, this monitor is not an action-cancellation monitor because $a_1 \notin \pi_{P_4}(\{e_1\}) = \{a_2, a_3\}$. Moreover, the logic programming translation specifies a different monitor: one that returns an empty set of actions for every epoch. This monitor is no longer maximal.

To deal with negative events, we are thus forced to loosen the definition of an epoch and let some events be undefined (i.e., neither the event nor its negation are known to have occurred).

Definition 13 An *extended epoch* is a collection of instances of events plus a subset of $\{!e : e \text{ is a primitive event symbol}\}$. An extended epoch E is *coherent* if for every negated event $!e$ in E there is no instance of e in E; *complete* if for every primitive event e that has no instances in the epoch, $!e$ is a member of E.

Under this definition, there are three possible coherent epochs in Example 9: $\{\}$, $\{e_1\}$ and $\{!e_1\}$. This definition naturally induces a new definition of *extended policies*. The input of an extended policy is an extended epoch and the output a set of actions. If we apply the policy in Example 9 to the extended empty epoch the policy outputs no actions. We can also define *extended conservative* monitors, regular and maximal *extended action-cancellation* monitors, and regular and maximal *extended event-cancellation* monitors with the appropriate modifications in the original definitions.

Note that the existence of the empty epoch guarantees that there is always an extended action-cancellation monitor, namely the extended monitor that maps every epoch to the empty set of actions. Note that the logic programming translation of the policy does not change. The only change occurs in the input where epochs are replaced with extended epochs. Furthermore, the (unchanged) logic program specifies now a maximal extended event-cancellation monitor. For an extended epoch E, let

$$occ'(E) = \quad \{occ(e) | e \in E\}.$$

Theorem 5 For any coherent extended epoch E, E' is a maximally consistent reduction of E iff $E' = \{e | occ(e) \in M \wedge ignore(e) \notin M\}$ and M is a minimal model of $\Pi_P \cup \Psi_{AC} \cup A_P \cup B_P \cup occ'(E)$. Furthermore, E' is coherent since $E' \subseteq E$.

Define the monitor $\Omega'_{P,AC}$ like $\Omega_{P,AC}$ in Definition 12, with $occ'(E)$ substituted for $occ(E)$. From Theorem 5, it follows that $\Omega'_{P,AC}$ is a maximal extended event-cancellation monitor. In addition observe that if we extend a (non-extended) epoch E to an extended epoch $ext(E) = \{g | g \in E\} \cup \{!e | e$ has no instances in $E\}$, $\Omega_{P,AC}(E) = \Omega'_{P,AC}(ext(E))$. It should also be clear that the complexity bounds (Theorem 3) do not change.

Concerning Example 10, note that although the monitor that outputs a_1 for every epoch is a maximal event-cancellation monitor, it is not a maximal *extended* event-cancellation monitor. $\Omega'_{P_4,AC}$, which outputs the empty set for $\{e_1\}$ is both an *extended* event-cancellation and action-cancellation monitor. Similarly, in Example 9, there is now an extended event-cancellation monitor: one that outputs an empty set of actions for every epoch.

We conclude the discussion of negation by addressing an *expressiveness* issue. At first glance it appears that allowing negated events in the rules may obviate the need for action constraints. In Example 2, adding !*defectiveProduct* to the second and third rules makes the action constraint superfluous. However, the elimination of action constraints is not always possible. In Example 1, eliminating the constraint would require adding existential quantification to the event language.

6 Rule clusters

As we have seen earlier, in some cases the unobtrusiveness requirement for monitors is desirable, while in other it is not. Moreover, it appears useful to be able to enforce this requirement only for selected parts (rules) of a policy.

6.1 Clusters and partitions

We define a *rule cluster* of a $\mathcal{PDL}$ policy P to be a subset of the set of rules of P. A *cluster partition* of P is a set of disjoint rule clusters of P that cover all of P. For a cluster partition $\mathcal{P} = (P_1, \ldots, P_k)$ of P and a set of action constraints AC, a *partitioned monitor* produces for each cluster P_i an action set A_i that has to satisfy the following properties:

- $\bigcup_{1 \le i \le k} A_i$ satisfies AC,
- for every epoch E and every i, $1 \le i \le k$, there is an epoch $E_i \subseteq E$ such that $A_i = \pi_{P_i}(E_i)$,

The first condition guarantees that the partitioned monitor is indeed a monitor, i.e., satisfies action constraints. The second condition provides unobtrusiveness *within* a single rule cluster. Separate clusters are independent and an event can be ignored only in some clusters.

For a given cluster partition $\mathcal{P} = (P_1, \ldots, P_k)$ and a given epoch E, the partitioned monitors corresponding to $\mathcal{P}$ can be ordered using the component-wise order. Consider the partition $\mathcal{P}_0$ in which every cluster consists of a single rule of the policy P. A partitioned monitor corresponding to $\mathcal{P}_0$ which additionally is maximal in the above ordering is a maximal action-cancellation monitor of P. On the other hand, a maximal partitioned monitor corresponding to the partition $\mathcal{P}_1$ consisting of a single cluster is a maximal event-cancellation monitor of P. So by adjusting the partition of P we can achieve the desired degree of unobtrusiveness in the monitor.

6.2 Translation to logic programs

We show here how to obtain an executable specification of a maximal partitioned monitor of a policy by modifying the translation to logic programs defined earlier. We simulate the partitioning of rules into clusters by introducing an additional subscript, corresponding to the cluster number, into the monitor meta-predicates. However, only some predicates: *exec*, *ignore*, and *accept* need to be subscripted in that way. The remaining predicates: *occ* and *block* are interpreted identically in each cluster and thus do not need to be subscripted.

We show now how to obtain the translation of the rules of a policy P. Consider the policy translation first. Each rule

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

in a cluster P_i is translated into the implication

$$exec_i(a) \leftarrow occ(e_1) \wedge \ldots \wedge occ(e_l) \wedge C.$$

Additionally, for each cluster P_i there is a rule that collects the actions to be executed (A is a logical variable here):

$$exec(A) \leftarrow exec_i(A).$$

Denote the set of implications obtained in this way by Π_{P_i}. Second, the *conflict rules* Ψ_{AC} are defined as before. Thus, each constraint in AC of the form

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

is translated into the rule

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

Third, for each rule

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

in a cluster P_i, we obtain a *blocking rule*

$$ignore_i(e_1) \vee \ldots \vee ignore_i(e_n) \leftarrow occ(e_1) \wedge \ldots \wedge occ(e_n) \wedge C \wedge block(a)$$

and an *accepting rule*

$$accept_i(a) \leftarrow occ(e_1) \wedge \ldots \wedge occ(e_n) \wedge C \wedge \neg ignore_i(e_1) \wedge \ldots \wedge \neg ignore_i(e_n).$$

For a cluster P_i, denote the set of blocking rules by B_{P_i} and the set of accepting rules by A_{P_i}.

Theorem 6 Let $\mathcal{P} = (P_1, \ldots, P_k)$ be a cluster partition of a policy P. Let $\Omega_{\mathcal{P},AC} : Epochs \longrightarrow (2^{Actions})^k$ be a transducer defined for any epoch E in the following way:

1. Select a minimal model M of
$$\bigcup_i \Pi_{P_i} \cup \Psi_{AC} \cup \bigcup_i (A_{P_i} \cup B_{P_i}) \cup occ(E).$$
2. Return the vector $(A_1, \ldots, A_k)$ where $A_i = \{a | accept_i(a) \in M\}$ as output (i.e., $\Omega_{\mathcal{P},AC}(E) = \{a | accept_i(a) \in M\}$).

Then $\Omega_{\mathcal{P},AC}$ is a maximal partitioned monitor corresponding to $\mathcal{P}$.

If we have a policy that is to be monitored unobtrusively together with a rule for which this requirement is undesirable (as in Example 3), we just break the policy into two clusters and use the corresponding partitioned monitor.

Example 11 Consider again Example 2. If each rule is in a separate cluster and *mailProduct* is blocked, then the event *orderReceived* is ignored in the second cluster but not in the third one and the action *chargeCC* is accepted, negating unobtrusiveness.

For the purpose of efficient conflict resolution, a set of single rule clusters may be viewed as a single policy. For this policy a PTIME maximal action-cancellation monitor is obtained, via an adaptation of the proof of Theorem 2. The output of this monitor is a set of *accept* atoms and can be combined with the outputs of the remaining clusters.

7 The policy server

A network management system based on full $\mathcal{PDL}$ has been incorporated into a commercial softswitch developed at Bell Labs that couples public telephony with IP technology [22]. The softswitch manages an unbounded number of policy servers that are able to run policies written in $\mathcal{PDL}$.[1] When a policy is loaded into a policy server the server creates a policy evaluator for the input policy, it contacts the devices (i.e., routers, hubs, computers, etc.) that can potentially generate instances of the events of interest to the policy and registers the interest with the devices. The registration happens at policy enabling points (PEPs) that wrap around the devices to act as interfaces between the devices and the policy servers. Events generated by a device are intercepted by its assigned PEP, translated into event terms and sent to the appropriate policy server. When an event arrives at the policy server, the server gives copies of the event to each policy evaluator that is running a policy that mentions the event. Each evaluator accumulates the events in a buffer and using a time constant T given to the evaluator during initialization, the evaluator groups events from the buffer into epochs based on the following criterion. An event e arriving at a buffer at time T_1 belongs to the same epoch than the previous event in the buffer if the difference between the time of the beginning of the previous event epoch and T_1 is less than or equal to T. Otherwise, the arriving event belongs to a new epoch and the beginning time of this new epoch is set to T_1. For the special case in which e is

[1] It is bounded only by the capacity of the computers used.

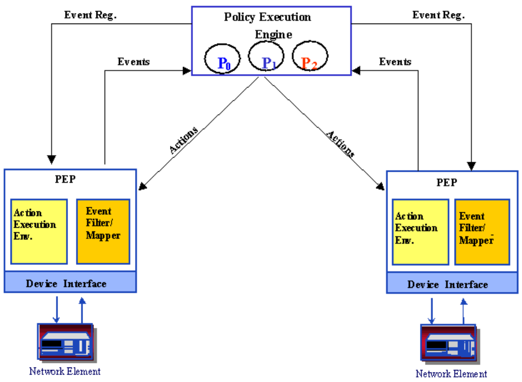

Figure 1: System Architecture

the first event sent to the evaluator, the first epoch is started with the beginning time also set to T_1. Each policy evaluator runs with the appropriate epoch as input. Following the policy evaluations, each policy evaluator returns a set actions. The server takes these actions and sends them to the appropriate PEPs which translate them into device-specific operations. The architecture of the server is depicted in Figure 1. The system is completely written in Java except for some parts of the PEPs which are device-dependent.

One possible way to add the conflict resolution capabilities described here to the current implementation is for the policy server to be extended as follows. For every set of actions A generated by the evaluation of a policy P with an input epoch E, take the epoch E and the set $exec(A) = \{exec(a) | a \in A\}$, and execute the following two steps:

1. Select a minimal model M of $exec(A) \cup \Psi_{AC} \cup A_P \cup B_P \cup occ(E)$.
2. Return the set $\{a | accept(a) \in M\}$ as output.

This new set will be used by the server as the final output of the evaluation of P. Observe that the difference between these two steps and the monitor $\Omega_{P,AC}$ introduced in 12 is that Π_P has been replaced by $exec(A)$. We are using the current implementation to evaluate the policy and leave only the resolution of conflicts to the logic program.

8 Related work

Several languages have been proposed for policy based network management (see for example [18, 24, 11, 14]). Much of the work in these papers is dedicated to implementation issues. Similarly, there are several implementations of event notification systems available but all have informal specifications (see for example [10] and the references therein). To address problems like conflict resolution it is essential to have languages with precise semantics like $\mathcal{PDL}$. However, the framework described in the present paper is not restricted to $\mathcal{PDL}$ and can be applied to other event languages. We will briefly discuss in the next section some general guidelines on how we can extend the framework to a more general class of policies.

More formal work on event notification languages has been developed specifically with the goal of network monitoring [25, 11]. The emphasis in these languages is on very expressive primitive events and event compositions. Actions are secondary, so the question of conflicts was never addressed. Our approach assumes that the user knows in advance the action constraints. However, if policies are developed by independent entities sometimes it might be difficult to obtain the constraints. Work on detecting statically potential conflicts is reported in [16]. This work might be useful to generate action constraints automatically.

Conflict resolution among production rules has been studied in AI and databases. For example, OPS5 [2] uses elaborate criteria depending on the form of the rules and the data to resolve such conflicts. Active database systems, on the other hand, typically use priorities to choose among conflicting rules [1]. A comprehensive framework for conflict resolution in this context was presented in [12]. The results about the computational complexity of testing consistency of production rules were presented in [3]. Those works are quite different from ours in that they assume interpreted actions (variable assignments or database updates) and mostly ignore the event part of event-condition-action rules. A recent work [13] deals with a model that is closer to ours, although the conflicts studied are still between the rules, not actions, and the events are not taken into account. This work proposes a meta-language for the control of rule executions. The rules themselves are viewed as black boxes. Using the meta-language of [13], one can often achieve similar effects to our framework. For example, the policy P_1 from Example 2 can be represented as a conjunction of the following relationships from [13]: the rules 1 and 2 are mutually exclusive, the rule 1 is preferred over the rule 2, the rules 2 and 3 require each other. The relative expressive power of the framework of [13] and our framework remains to be investigated. As should be clear from the above example, the framework of [13] is less declarative and lower-level than ours, and conflict resolution and rule dependencies have to be explicitly programmed-in.

The notion of *action constraints* was independently introduced in [5] and [8] (a similar notion was also proposed in [9]). In the paper [8], Eiter *et al* introduced a modal policy specification language for software agents. Conflict resolution is only one of the many issues addressed in [8]. The authors propose, using an entirely different terminology, maximal action cancellation monitors, without considering event cancellation. In general, conflict resolution methods in the current active rule literature, including [13] and [8], are not *unobtrusive* and in many cases the result they produce does not necessarily correspond to conflict-free executions of the rules. The paper [7] is a follow-up paper to [8] and contains numerous complexity results. They all involve, however, languages richer that the subset of $\mathcal{PDL}$ used in the present paper.

Several recent works [8, 16, 19] have postulated a modal, deontic framework for specifying prohibitions and obligations of agents in a distributed environment. Simple obligations and prohibitions can be captured in our framework. First, notice that an ECA rule can be read as an *obligation* to execute its action part if the events in the event part occur. Second, a *prohibition* to execute an action a if some events $e_1, \ldots, e_n$ occur can be simulated using action constraints and priorities (Section 3.3). We introduce the rule

$$e_1 \& \ldots \& e_n \textbf{ causes } not\_a$$

plus the constraint

$$\textbf{never} \quad a \wedge not\_a$$

into the policy. The action not_a is a new action and has no effect. We also give priority to not_a over a.

9 Conclusions and further work

In this paper we have presented a logic programming framework for detecting action conflicts in policies and finding resolutions to these conflicts. The class of policies that we consider is limited to *stateless* transducers. We are currently extending our approach to deal with *state*. In particular, we accommodate *sequence events* (like in [20]), already a part of full $\mathcal{PDL}$ [15]. This extension requires generalizing the semantics of policies and monitors to mappings from *sequences* of epochs to *sequences* of sets of actions. Also, we are studying monitors that are not based on cancelling conflicting actions but rather on *delaying* them until no conflict occurs. Finally, we are considering a more general constraint language that permits *temporal* constraints. These extensions will enable us to deal with, among others, the issue of conflict resolution in active database rules. To address conflict resolution for full $\mathcal{PDL}$, we need to consider regular event expressions and aggregation. The generalized semantics, described above, is sufficient to handle this extension. However, the definition of the appropriate maximal monitors is non-trivial.

There is also another dimension of policy conflicts yet to be explored. Sometimes conflict between policies is resolved not by cancelling or delaying actions but by *composing* the policies into a new policy. Suppose, for example, clients make calls to a customer support line. Support agents are distributed around the world and a policy is written to route calls according to the geographic location of both the caller and the support agents, and the current congestion conditions of the network. There is also a policy that tries to match support agents with clients according to a profile of the client needs. A monitor, perhaps with some extra information, should make a decision on how to connect clients to agents taking into account both policies. Policies of this type are typical of telephone call centers. They are hard-coded in the center and conflicts are currently solved manually by modifying the policies involved.

References

[1] R. Agrawal, R. Cochrane, and B. G. Lindsay. On maintaining priorities in a production rule system. In *VLDB*, pages 479–487, 1991.

[2] L. Brownston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[3] H. Kleine Büning, U. Löwen, and S. Schmitgen. Inconsistency of production systems. *Journal of Data and Knowledge Engineering*, 3:245–260, 1988/89.

[4] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology.* Addison-Wesley, 1997.

[5] J. Chomicki, J. Lobo, and S. Naqvi. Axiomatic conflict resolution in policy management. Technical Report ITD-99-36448R, Bell Labs, February 1999.

[6] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3), 1997.

[7] T. Eiter and V.S. Subrahmanian. Heterogeneous active agents, II: Algorithms and complexity. *Artificial Intelligence*, 108:257–307, March 1999.

[8] T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108:179–255, March 1999.

[9] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: Courteous logic programs in XML. In Michael P. Wellman, editor, *Proc. of the First ACM Conference on Electronic Commerce*. ACM Press, November 1999.

[10] R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY event notification system. In *8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[11] M. Z. Hasan. An active temporal model for network management databases. In *IFIP/IEEE 4th International Symposium on Integrated Network Management*, pages 524–535, Santa Barbara, California, May 1995.

[12] Y. E. Ioannidis and T. K. Sellis. Supporting inconsistent rules in database systems. *Journal of Intelligent Information Systems*, 1(3/4), 1992.

[13] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick. Managing conflicts between rules. In *Proc. 15th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 192–201, 1996.

[14] T. Koch, C. Krell, and B. Krämer. Policy definition language for automated management of distributed systems. In *Second International Workshop on Systems Management*, Toronto, Canada, June 1996.

[15] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of AAAI*, Orlando, FL, July 1999.

[16] E. C. Lupu and M. Sloman. Conflict analysis for management policies. In R. Stadler A. Lazar, R. Saraco, editor, *Proc. 5th IFIP/IEEE International Symposium on Integrated Network Management*, pages 430–443, 1997.

[17] N. H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *Proc. of the 18th International Conference on Distributed Computing Systems*, May 1998.

[18] J. Moffett and M. S. Sloman. Policy hierarchies for distributed system management. *IEEE JSAC*, 11(9), 1993.

[19] J. D. Moffett and M. S. Sloman. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4(1):11–22, 1994.

[20] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. of SIGMOD*, Tucson, AZ, May 1997.

[21] I. Niemelä. Towards efficient default reasoning. In *Proc. of 14th International Joint Conference on Artificial Intelligence*, pages 312–318, Montreal, 1995.

[22] A. Virmani, J. Lobo, and M. Kohli. NETMON: Network management for the SARAS softsiwtch. In *Proc. of the IEEE/IFIP Network Operations and Management Symposium*, April 2000. To appear.

[23] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, 1995.

[24] R. Wies. Policies in network and system management - formal definition and architecture. *Journal of Network and System Management*, 2(1):63–83, 1994.

[25] O. Wolfson, S. Sengupta, and Y. Yemini. Managing communication networks by monitoring databases. *IEEE Transactions on Software Engineering*, 17(9):944–953, September 1991.