Conflict Resolution Using Logic Programming

Jan Chomicki, *Member*, *IEEE Computer Society*,
Jorge Lobo, *Member*, *IEEE*, and
Shamim Naqvi

Abstract—This paper addresses some issues involved in applying the *event-condition-action* (ECA) rule paradigm of active databases to *policies*—collections of general principles specifying the desired behavior of a system. 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 range of monitors that filter the output of the policy to satisfy the constraints.

Index Terms—Policy, action constraint, conflict resolution, active rules, event, action.

1 Introduction

The simple *event-condition-action* rule paradigm has proved very useful in many AI and database applications [33], 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 [21], electronic commerce [18], security and access management [26], 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* $\mathcal{PDL}$ [25], in which policies are formulated as sets of *event-condition-action* rules of the form

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

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 server described in [32] provides an implementation for such transducers. A formal description of the syntax and semantics of the subset of $\mathcal{PDL}$ used in the present paper 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: There is an automatic reservation system for a conference room in a department. The reservations are controlled by a simple first-in-first-served policy captured by the following rule:

$$requestRes(User)\ \mathbf{causes}\ procRes(User). \qquad (2)$$

The procedure $procRes$ 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 $procRes$ calls is cancelled. We introduce *action constraints* into $\mathcal{PDL}$ to capture this type of conflict. The constraints describe under which circumstances a set of actions cannot be executed simultaneously. For the reservation policy, the appropriate action constraint is

$$\mathbf{never}\ procRes(U_1) \wedge procRes(U_2)\ \mathbf{if}\ U_1 \neq U_2.$$

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. A new abstraction, *monitor*, is defined as a filter that is applied to the output of a policy (i.e., a set of actions) by cancelling some actions to obtain a result consistent with the constraints. Given a policy and a set of constraints on the simultaneous execution of actions, the framework produces a monitor for the policy. However, a monitor cannot cancel actions arbitrarily. First, it should cancel as few actions as possible. Second, action cancellation should be based on the assumptions about execution atomicity. We have identified two distinct assumptions of this kind. Under the first one, all the actions are independent and the cancellation of an action has no influence on other actions (*action-cancellation monitors*). Under the second one, the actions caused by the same event fail or succeed together (*event-cancellation monitors*).

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 be subsequently modified by the user. The latter option allows for more flexibility in handling conflicts. The logic programming formulation provides a formal framework to describe the semantics of policies. It has many desirable properties by being both executable (using well-established logic programming techniques) and easy to modify and analyze formally.

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 data 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.

- *J. Chomicki is with the Department of Computer Science and Engineering, The State University of New York at Buffalo, NY 14260-2000. E-mail: chomicki@cse.buffalo.edu.*
- *J. Lobo is with Teltier Technologies, 60 Walnut Avenue, #300, Clark, NJ 07066. E-mail: jorge@teltier.com.*
- *S. Naqvi is with Winphoria Networks, 3 Hoghwood ZDrive West, Tewksbury, MA 01876. E-mail: shamim@winphoria.com.*

Manuscript received 13 June 2000; revised 22 Feb. 2002; accepted 23 May 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112265.

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 *and & is interpreted as a conjunction of events. 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 (negated events have no attributes). 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 constant or a variable that appears in the event part of the rule. An* action *is a ground action term.*

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 a constant or a variable that appears in the event part. The condition represents the conjunction of the predicates.*

2.2 Semantics

We adopt the view that a policy expects as input a set of instances of primitive events (an *epoch*) and assumes that the events in the set occurred simultaneously. The granularity of time for the epochs 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.

In the following, assume P is a policy.

Definition 5. *A finite set of event instances is an* epoch. *The set of all possible epochs (given the event symbols of P) is denoted by $Epochs(P)$. 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 finite sets of ground action terms (built with the action symbols of P) is denoted by $ActionSets(P)$, and its elements are called* action sets.

Formally, the semantics of P is given by a transducer $\pi_P : Epochs(P) \longrightarrow ActionSets(P)$. 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

$$occ(E) = \{occ(g) | g \in E\} \cup \{occ(!e) | e \text{ has no instances in } E\}.$$

The intuitive meaning of $occ(e(t_1, \ldots, t_n))$ is that the instance $e(t_1, \ldots, t_n)$ of the primitive event e *occurred* in the current epoch. Then, the semantics of each rule of the form "$e_1\& \ldots \& e_l$ **causes** a **if** C" in the policy is specified as the implication

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

Notice that the above implication has the same variables as the original rule.

We denote by Π_P the set of rules that result from the semantic specification of each policy rule in P. It is easy to see that Π_P is a nonrecursive Horn logic program.

Definition 6. *$\pi_P : Epochs(P) \longrightarrow ActionSets(P)$ is the transducer defined by P if, for every epoch E,*

$$a \in \pi_P(E) \textit{ iff } \Pi_P \cup occ(E) \models exec(a).$$

By the virtue of Π_P being a nonrecursive Horn program, the computation of $\pi_P(E)$ can be done in time polynomial with respect to the cardinality $|E|$ of the input epoch E.

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 "***never** *$a_1 \wedge \ldots \wedge a_m$* **if** *$C$." 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 (respectively, a set of action constraints AC) if S is a model of ac (respectively, of all the constraints in AC) in the standard model theoretic sense (with action terms viewed as literals). We use the standard notation $S \models ac$ (respectively, $S \models AC$) to denote this relationship.*

A 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, an AC-monitor ω_{AC} is a transducer $\omega_{AC} : Epochs(P) \longrightarrow ActionSets(P)$ 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). We adopted this approach because it is often inconvenient (or even impossible) to define monitors using $\mathcal{PDL}$. Also, if the set of constraints AC is clear from the context, we will use the term "monitor" instead of "AC-monitor."

Next, we identify several important properties of monitors. They capture the intuition that monitors should be chosen to behave *as close as possible* to the policy whose 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. An epoch E' is a (P, AC)-consistent reduction of an epoch E if $E' \subseteq E$ 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. *An AC-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 it does not generate any actions beyond those of P, i.e., for every epoch E, $\omega_{AC}(E) \subseteq \pi_P(E)$.*
3. *A* maximal action-cancellation *monitor of P if, for every epoch E, $\omega_{AC}(E)$ is a maximal subset of $\pi_P(E)$ 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 $\omega_{AC}(E) = \pi_P(E')$.*
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 $\omega_{AC}(E) = \pi_P(E')$.*

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 work by producing a consistent reduction of the input epoch, to which they then apply the original policy. We call such monitors *unobtrusive* because, if the user has only access to the actions generated by the monitor but not to the input epoch, she cannot observe constraint violations. We motivate the need for unobtrusive monitors by noticing that the behaviors produced by a policy on subsets of the input epoch are guaranteed to be correct since such subsets could occur as actual epochs. The correctness of any other behaviors has to be established with respect to some assumptions extraneous to the policy.

To further appreciate the importance of unobtrusiveness, consider the following example.

Example 2. Assume we have the policy P_1

$$defectiveProduct \textbf{ causes } stop$$
$$orderReceived \textbf{ causes } mailProduct$$
$$orderReceived \textbf{ causes } chargeCC$$

and the constraint "**never** $stop \wedge mailProduct$."

If the events *defectiveProduct* and *orderReceived* occur together in an epoch, 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* (building a maximal consistent reduction of the epoch) 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

$$orderReceived \textbf{ causes } mailProduct, chargeCC$$

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

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

or, if constraints can contain negative literals, the constraint:

$$\textbf{never } \neg mailProduct \wedge chargeCC.$$

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

$$serviceCallCompleted \textbf{ causes } chargeCC.$$

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 nonunobtrusive monitors.

3.2 Semantics of Conflict Resolution

Here, we formally define the semantics of monitors for policies that do not involve negated events by augmenting the Datalog specification 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). The input epoch of a monitor is specified using occ atoms, the output of the policy using $exec$ atoms, and the output of a monitor using $accept$ atoms. The specifications below typically describe not just a single monitor but rather a whole family of them.

In the specification, we will have several kinds of rules: *conflict* rules, *blocking* rules, and *accepting* rules. Conflict rules will be the same for action and event-cancellation monitors. Action monitors will not use blocking rules and will use a simplified version of accepting rules.

Note: In the rules below, the negation is interpreted as negation-as-failure, not logical negation.

3.2.1 Action-Cancellation Monitors

Conflict rules Ψ_{AC}. Each constraint in AC of the form "**never** $a_1 \wedge \ldots \wedge a_n$ **if** C" is captured as a *conflict* rule

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

Accepting rules A^a_P. For each action a occurring in a policy rule, there is an accepting rule:

$$accept(a) \leftarrow exec(a) \wedge \neg block(a).$$

Notice that for action cancellation, only the actions generated by the policy matter. It is irrelevant which events caused those actions and which policy rules were used in the generation. Neither are reflected in the above rules.

3.2.2 Event-Cancellation Monitors

Conflict rules Ψ_{AC}. The same as for action-cancellation monitors.

Blocking rules B_P and **accepting rules** A^e_P. For each policy rule in P of the form

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

conflict resolution is specified using the *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 the *accepting* rule

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

Notice that in event cancellation, the causal relationships between events and actions are taken into account.

3.2.3 Examples

Example 4. The constraint "**never** $stop \wedge mailProduct$" is captured using the conflict rule

$$\begin{aligned}&block(stop) \vee block(mailProduct) \leftarrow exec(stop)\\ &\quad \wedge exec(mailProduct).\end{aligned}$$

Example 5. The constraint

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

is captured using the conflict rule

$$\begin{aligned}&block(procRes(U_1)) \vee block(procRes(U_2)) \leftarrow\\ &exec(procRes(U_1)) \wedge exec(procRes(U_2)) \wedge U_1 \neq U_2.\end{aligned}$$

Example 6. Conflict resolution using event cancellation for the rule

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

is specified as

$$\begin{aligned}&ignore(dial) \vee ignore(charge) \leftarrow occ(dial) \wedge occ(charge)\\ &\quad \wedge block(connect)\\ &accept(connect) \leftarrow occ(dial) \wedge occ(charge) \wedge \neg ignore(dial)\\ &\quad \wedge \neg ignore(charge).\end{aligned}$$

Example 7. We now present the complete specification of conflict resolution based on event cancellation for the policy P_1 from Example 2. The conflict rule:

$$\begin{aligned}&block(stop) \vee block(mailProduct) \leftarrow exec(stop)\\ &\quad \wedge exec(mailProduct).\end{aligned}$$

The blocking rules:

$$\begin{aligned}&ignore(defectiveProduct) \leftarrow occ(defectiveProduct) \wedge block(stop)\\ &ignore(orderReceived) \leftarrow occ(orderReceived) \wedge block(mailProduct)\\ &ignore(orderReceived) \leftarrow occ(orderReceived)\\ &\quad \wedge block(chargeCreditCard).\end{aligned}$$

The accepting rules:

$$\begin{aligned}&accept(stop) \leftarrow occ(defectiveProduct) \wedge \neg ignore(defectiveProduct)\\ &accept(mailProduct) \leftarrow occ(orderReceived) \wedge \neg ignore(orderReceived)\\ &accept(chargeCreditCard) \leftarrow occ(orderReceived)\\ &\quad \wedge \neg ignore(orderReceived).\end{aligned}$$

3.3 General Properties

The logic program $L^a_{P,AC} = \Pi_P \cup \Psi_{AC} \cup A^a_P$ specifies conflict resolution based on action cancellation. The program $L^e_{P,AC} = \Pi_P \cup \Psi_{AC} \cup A^e_P \cup B_P$ specifies conflict resolution based on event cancellation.

Note that $L^a_{P,AC}$ and $L^e_{P,AC}$ are *safe*. That is, variables that appear in the consequent of the implications or in negated literals in the antecedent also appear in positive literals in the antecedent. The programs 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, $L^a_{P,AC}$ and $L^e_{P,AC}$ are hierarchical (i.e., nonrecursive) disjunctive logic programs.

We show now that the correspondence between the stable models [15] of $L^a_{P,AC}$ (respectively, $L^e_{P,AC}$) together with an input epoch, and maximal action-cancellation (respectively, event-cancellation) monitors. In the case of action-cancellation monitors, the correspondence is immediate.

Definition 12. *Let $\Omega^a_{P,AC}$: $Epochs(P) \longrightarrow ActiondSets(P)$ be a transducer defined for any epoch E in the following way:*

1. *Select a stable model M of $L^a_{P,AC} \cup occ(E)$.*
2. *Return the set $A_M = \{a | accept(a) \in M\}$ as output (i.e., $\Omega^a_{P,AC}(E) = \{a | accept(a) \in M\}$).*

Theorem 1. *$\Omega^a_{P,AC}$ is a maximal action-cancellation AC-monitor of P. Moreover, all maximal action-cancellation AC-monitors of P can be obtained in this way.*

Proof. Let M be a stable model of $L^a_{P,AC} \cup occ(E)$. Assume that there is a ground instance "**never** $a_1 \wedge \ldots \wedge a_n$" of an action constraint that is violated in the action set A_M. Thus, for every $i = 1, \ldots n$: $accept(a_i) \in M$, $exec(a_i) \in M$, and $block(a_i) \notin M$; otherwise, $accept(a_i)$ is not supported since the only rule where $accept(a_i)$ occurs in the head is

$$accept(a_i) \leftarrow exec(a_i) \wedge \neg block(a_i).$$

But, this is not possible since then M will not be a model of

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

To show that A_M is a maximal consistent subset of $\pi_P(E)$, assume it's not and let's say the action $a \in \pi_P(E)$ can be added to A_M without violating consistency. Therefore, $block(a) \in M$ since $exec(a) \in \pi_P(E)$ and $accept(a)$ is not supported in the program. Hence, there is a rule

$$\begin{aligned}&block(a) \vee block(a_1) \vee \ldots \vee block(a_n)\\ &\quad \leftarrow exec(a) \wedge exec(a_1) \wedge \ldots \wedge exec(a_n)\end{aligned}$$

in $L^a_{P,AC} \cup occ(E)$, such that $block(a) \in M$, and for every $i = 1, \ldots, n$, $exec(a_i) \in M$ and $block(a_i) \notin M$; otherwise, M is not a minimal model. Hence, for every $i = 1, \ldots, n$, $accept(a_i) \in M$, and a cannot be added to the set of accepted actions without violating a constraint.

Let Max be a maximal action-cancellation monitor of P. Given an epoch E, let

$$\begin{aligned}M = &\{accept(a) | a \in Max(E)\} \cup \{exec(a) | a \in \pi_P(E)\}\\ &\cup \{block(a) | a \in \pi_P(E) \setminus Max(E)\} \cup occ(E).\end{aligned}$$

We need to show that M is a stable model of $L^a_{P,AC} \cup occ(E)$. It is easy to see that $occ(E) \cup \{exec(a) | a \in \pi_P(E)\}$ is in any stable model of $L^a_{P,AC} \cup occ(E)$.

If $block(a) \in M$, it means that $a \in \pi_P(E)$ but $a \notin Max(E)$, and since Max is a maximal cancellation monitor, this means that there is an action constraint "**never** $a \wedge a_1 \wedge \ldots \wedge a_n$" such that $a_i \in Max(E)$ for every $i = 1, \ldots, n$. Hence,

$$exec(a), exec(a_1), \ldots, exec(a_n) \in M,$$

$block(a_i) \notin M$ for any $i = 1, \ldots, n$. Thus, $block(a)$ is supported by the rule

$$\begin{aligned}&block(a) \vee block(a_1) \vee \ldots \vee block(a_n)\\ &\quad \leftarrow exec(a) \wedge exec(a_1) \wedge \ldots \wedge exec(a_n)\end{aligned}$$

in $L^a_{P,AC} \cup occ(E)$. An $accept(a)$ is in M only if $exec(a) \in M$ and $block(a) \notin M$. M is clearly minimal since $L^a_{P,AC} \cup occ(E)$ is recursion free. □

The situation is a bit more complicated in the case of event-cancellation monitors. To get maximal event-cancellation, we need to minimize the set of *ignore* atoms. However, the stable model construction—along the above lines but applied to $L^e_{P,AC} \cup occ(E)$—will minimize not this set but rather the set of *ignore* and *block* atoms together. Therefore, there may be stable models of $L^e_{P,AC} \cup occ(E)$ which do not correspond to maximal event-cancellation monitors. To avoid that situation, the logic program $L^e_{P,AC}$ needs to be transformed by eliminating all the occurrences of the *block* predicate symbol.

Definition 13. *The logic program $cl(L^e_{P,AC})$ consists of* block-*free clauses obtained by exhaustively resolving the clauses Ψ_{AC} with the clauses B_P in $L^e_{P,AC}$.*

Definition 14. *Let $\Omega^e_{P,AC} : Epochs(P) \longrightarrow ActionSets(P)$ be a transducer defined for any epoch E in the following way:*

1. *Select a stable model M of $cl(L^e_{P,AC}) \cup occ(E)$.*
2. *Return the set $B_M = \{a | accept(a) \in M\}$ as output (i.e., $\Omega^e_{P,AC}(E) = \{a | accept(a) \in M\}$).*

Theorem 2. *$\Omega^e_{P,AC}$ is a maximal event-cancellation AC-monitor of P. Moreover, all maximal event-cancellation AC-monitors of P can be obtained in this way.*

Proof Similar to the proof of Theorem 1. □

It is also immediate to see that $\Omega^a_{P,AC}$ and $\Omega^e_{P,AC}$ are conservative (if no conflicts occur, no actions are blocked and no events ignored). Also, both logic programs obtained, $L^a_{P,AC}$ and $cl(L^e_{P,AC})$, are *pseudo-head-cycle-free* and stable models of such programs can be computed nondeterministically in PTIME [4]. (The programs are not necessarily *head-cycle-free* since the same predicate can appear more than once in the head of a single rule.)

3.4 Modifying the Specification

3.4.1 Priority Ordering

The logic program specification of monitors (Section 3) defines all maximal action- or 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. For instance, 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 8. 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

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

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 minimal models of the resulting logic program still correspond to maximal monitors. If a global ordering on actions is provided, then the notion of monitor maximality itself becomes unclear. The ordering on actions can be lifted in several different ways to an ordering of action sets [11]. In addition, the sets are ordered by the subset ordering. How to combine those orderings in a semantically meaningful way is an issue for future research.

Note that a *prohibition* to execute an action a if some events $e_1, \ldots, e_n$ occur can be simulated using action constraints and priorities. We introduce the rule "$e_1 \& \ldots \& e_n$ **causes** not_a" plus the constraint "**never** $a \land not\_a$" into the policy. The action not_a is a new action and has no effect. We also give priority to not_a over a.

3.4.2 Persistent Events

In many cases, it is impossible, incorrect, or undesirable to ignore certain events, e.g., time events. We call such events *persistent*.

Example 9. In Example 6, the dial event is persistent. To capture persistent events, we can modify the blocking rules, i.e., when an event cannot be ignored it will never appear in the head of a blocking rule. Therefore, the blocking rule should be changed to

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

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.

3.5 Negation

If negated events are allowed in policy rules, the definition and implementation of action-cancellation monitors (which do not take events into accounts) are unaffected. However, there are several problems with applying event-cancellation monitors to policies with negated events. First, in the presence of negated events, policies may not have *any* event-cancellation monitors. 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. This is rather unintuitive.

To deal with negative events, we can loosen the definition of epochs and let some events be undefined (i.e., neither the event nor its negation is known to have occurred). If an event is ignored because of conflict resolution, the event becomes undefined and does not trigger any new actions. Similar to well-founded models of logic programs [20], epochs are now three-valued and we can extend the definition of *event-cancellation* monitors (including maximal ones) and the logic programmming specifications by appropriately modifying the original definitions [8].

We conclude this section by addressing the issue of the added *expressive power* of negation. 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.

4 Related Work

Several languages have been proposed for policy-based network management (see, for example, [27], [34], [21], [24], [10]). 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, [19] 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. More formal work on event notification languages has been developed specifically with the goal of network monitoring [35], [21]. 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.

Conflict resolution among production *rules* has been studied in AI and databases. For example, OPS5 [5] uses elaborate criteria depending on the form of the rules and 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 [22]. The results about the computational complexity of testing consistency of production rules were presented in [6]. 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. Moreover, priorities are associated with rules, not actions, and rule priorities lack formal semantics. A recent work [23] 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 metalanguage for the control of rule executions. The rules themselves are viewed as black boxes. Using the metalanguage of [23], one can often achieve similar effects to our framework, as in Example 2.

The notion of *action constraints* was independently introduced in [7] and [14] (a similar notion was also proposed in [18]). In the paper [14], Eiter et al. introduced a modal policy specification language for software agents. Conflict resolution is only one of the many issues addressed in [14]. 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 [23] and [14], are not *unobtrusive*. The paper [13] is a follow-up paper to [14] and contains numerous complexity results involving languages richer that the subset of $\mathcal{PDL}$ used in the present paper.

Finally, we relate our approach to view updates in the presence of integrity constraints [36], [31]. We can view *occ* as an extensional predicate, *exec* as an intensional predicate, Π_P as a view definition, and action constraints as denial integrity constraints. The crucial difference from the standard assumptions is that the database (*occ* and *exec* together) does not have to ultimately satisfy the constraints. In fact, in event-cancellation, we are interested in constructing *repairs*—maximal subsets of the input epoch that do not produce integrity violations. This can be modelled as the revision of the database with the constraints. Several recent papers [2], [3], [17], [16] addressed this issue but didn't consider intensional predicates.

Alternatively, we can define an additional view capturing the integrity violations and require an update to delete all tuples from that view [31]. So, in principle, we could use any general view update method providing this kind of functionality, e.g., [36], [31]. However, those methods are usually complex (because of their generality) and their computational complexity has not been analyzed. Moreover, they do not provide a declarative specification of conflict resolution strategies and are not easily amenable to extensions and modifications.

5 Conclusions and Further Work

In this paper, we have presented a formal 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*, albeit indirectly [9]. In particular, we accommodate *sequence events* (like in [29]), already a part of full $\mathcal{PDL}$ [25]. 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. (This conflict resolution strategy does not seem to have an analogue in database integrity maintenance.) Moreover, we are considering a more general constraint language that permits *temporal* constraints. These extensions will make it possible to relax the simultaneity restrictions on events and actions imposed by the framework of the present paper. 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 nontrivial. Also, as we have mentioned earlier, in some cases, the unobtrusiveness requirement is not the expected behavior of a monitor. It appears useful to be able to enforce this requirement only for selected parts (rules) of a policy. We can extend the monitors to apply to policies for which rules are grouped into clusters and unobtrusiveness is restricted to each cluster. A *policy server* of a slightly larger subset of $\mathcal{PDL}$ than the one presented here has been incorporated into the PacketStart IP Services Platform software developed at Bell Labs [32]. The policy server is being used to implement policies for detecting alarm conditions, fail-overs, device configuration and provisioning, service class configuration, congestion control, etc.

Acknowledgments

A preliminary version of some of the results of this paper appeared in [8].

References

[1] R. Agrawal, R. Cochrane, and B.G. Lindsay, "On Maintaining Priorities in a Production Rule System," *Proc. VLDB Conf.*, pp. 479-487, 1991.

[2] M. Arenas, L. Bertossi, and J. Chomicki, "Consistent Query Answers in Inconsistent Databases," *Proc. ACM Symp. Principles of Database Systems,* pp. 68-79, 1999.

[3] M. Arenas, L. Bertossi, and J. Chomicki, "Specifying and Querying Database Repairs Using Logic Programs with Exceptions," *Proc. Int'l Conf. Flexible Query Answering Systems,* pp. 27-41, 2000.

[4] R. Ben-Eliyahu-Zohary, L. Palopoli, and V. Zemlyanker, "More on Tractable Disjunctive Datalog," *J. Logic Programming,* vol. 46, pp. 61-101, 2000.

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

[6] H. Kleine Büning, U. Löwen, and S. Schmitgen, "Inconsistency of Production Systems," *J. Data and Knowledge Eng.,* vol. 3, pp. 245-260, 1988/89.

[7] J. Chomicki, J. Lobo, and S. Naqvi, "Axiomatic Conflict Resolution in Policy Management," Technical Report ITD-99-36448R, Bell Labs, Feb. 1999.

[8] J. Chomicki, J. Lobo, and S. Naqvi, "A Logic Programming Approach to Conflict Resolution in Policy Management," *Proc. Seventh Int'l Conf. Principles of Knowledge Representation and Reasoning,* F. Giunchiglia, A.G. Cohn, and B. Selman, eds., pp. 121-132, 2000.

[9] J. Chomicki and J. Lobo, "Monitors for History-Based Policies," *Policies for Distributed Systems and Networks,* pp. 57-72, 2001.

[10] N. Damianou, N. Dulay, E. Lupu, and E. Sloman, "The Ponder Policy Specification Language," *Policies for Distributed Systems and Networks,* pp. 18-38, 2001.

[11] B.A. Davey and H.A. Priestley, *Introduction to Lattices and Order.* Cambridge Univ. Press, 1990.

[12] T. Eiter, G. Gottlob, and H. Mannila, "Disjunctive Datalog," *ACM Trans. Database Systems,* vol. 22, no. 3, pp. 364-418, 1997.

[13] T. Eiter and V.S. Subrahmanian, "Heterogeneous Active Agents, II: Algorithms and Complexity," *Artificial Intelligence,* vol. 108, pp. 257-307, Mar. 1999.

[14] T. Eiter, V.S. Subrahmanian, and G. Pick, "Heterogeneous Active Agents, I: Semantics," *Artificial Intelligence,* vol. 108, pp. 179-255, Mar. 1999.

[15] M. Gelfond and V. Lifschitz, "Classical Negation in Logic Programs and Disjunctive Databases," *New Generation Computing,* vol. 9, nos. 3/4, pp. 365-386, 1991.

[16] G. Greco, S. Greco, and E. Zumpano, "A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases," *Proc. Int'l Conf. Logic Programming,* pp. 348-364, 2001.

[17] S. Greco and E. Zumpano, "Querying Inconsistent Databases," *Proc. Int'l Conf. Logic for Programming and Automated Reasoning,* pp. 308-325, 2000.

[18] B.N. Grosof, Y. Labrou, and H.Y. Chan, "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML," *Proc. First ACM Conf. Electronic Commerce,* M.P. Wellman, ed., Nov. 1999.

[19] R.E. Gruber, B. Krishnamurthy, and E. Panagos, "High-Level Constructs in the READY Event Notification System," *Proc. Eighth ACM SIGOPS European Workshop,* Sept. 1998.

[20] A. Van Gelder, K.A. Ross, and J.S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *J. ACM (JACM),* vol. 38, no. 3, pp. 619-649, 1991.

[21] M.Z. Hasan, "An Active Temporal Model for Network Management Databases," *Proc. IFIP/IEEE Fourth Int'l Symp. Integrated Network Management,* pp. 524-535, May 1995.

[22] Y.E. Ioannidis and T.K. Sellis, "Supporting Inconsistent Rules in Database Systems," *J. Intelligent Information Systems,* vol. 1, nos. 3/4, 1992.

[23] H.V. Jagadish, A.O. Mendelzon, and I.S. Mumick, "Managing Conflicts Between Rules," *Proc. 15th ACM SIGACT/SIGMOD Symp. Principles of Database Systems,* pp. 192-201, 1996.

[24] T. Koch, C. Krell, and B. Krämer, "Policy Definition Language for Automated Management of Distributed Systems," *Proc. Second Int'l Workshop Systems Management,* June 1996.
[25] J. Lobo, R. Bhatia, and S. Naqvi, "A Policy Description Language," *Proc. AAAI,* July 1999.
[26] N.H. Minsky and V. Ungureanu, "A Mechanism for Establishing Policies for Electronic Commerce," *Proc. 18th Int'l Conf. Distributed Computing Systems,* May 1998.
[27] J. Moffett and M.S. Sloman, "Policy Hierarchies for Distributed System Management," *IEEE JSAC,* vol. 11, no. 9, 1993.
[28] J.D. Moffett and M.S. Sloman, "Policy Conflict Analysis in Distributed System Management," *J. Organizational Computing,* vol. 4, no. 1, pp. 11-22, 1994.
[29] I. Motakis and C. Zaniolo, "Temporal Aggregation in Active Database Rules," *Proc. SIGMOD,* May 1997.
[30] *Policies for Distributed Systems and Networks,* M. Sloman, J. Lobo, and E.C. Lupu, ed. Springer-Verlag, 2001.
[31] E. Teniente and A. Olivé, "Updating Knowledge Bases while Maintaining Their Consistency," *VLDB J.,* vol. 4, pp. 193-241, 1995.
[32] A. Virmani, J. Lobo, and M. Kohli, "NETMON: Network Management for the SARAS Softswitch," *Proc. IEEE/IFIP Network Operations and Management Symp.,* Apr. 2000.
[33] J. Widom and S. Ceri, *Active Database Systems.* Morgan-Kaufmann, 1995.
[34] R. Wies, "Policies in Network and System Management—Formal Definition and Architecture," *J. Network and System Management,* vol. 2, no. 1, pp. 63-83, 1994.
[35] O. Wolfson, S. Sengupta, and Y. Yemini, "Managing Communication Networks by Monitoring Databases," *IEEE Trans. Software Eng.,* vol. 17, no. 9, pp. 944-953, Sept. 1991.
[36] B. Wüthrich, "On Updates and Inconsistency Repairing in Knowledge Bases," *Proc. IEEE Int'l Conf. Data Eng.,* 1993.