Monitors for History-based Policies

Jan Chomicki[1] and Jorge Lobo[2]

[1] Dept. of Computer Science and Engineering
University at Buffalo
Buffalo, NY 14260-2000
chomicki@cse.buffalo.edu
[2] Network Computing Research Dept.
Bell Labs
Murray Hill, NJ 07974
jlobo@research.bell-labs.com

Abstract. We investigate the issue of conflict detection and resolution for policies formulated as sets of event-condition-action rules. We focus on the temporal dimension of policies. In particular, we consider sequence events in rules, conflict resolution through cancellation or delay, and temporal action constraints. We formally define monitors – procedures for resolving conflicts. We present algorithms for the computation of optimal monitors.

1 Introduction

Policies are common in many software application areas: electronic commerce, network management, telecommunications, security etc. Recently, there has been a significant growth of interest in languages for formulating policies and mechanisms for their implementation.

In [14] a declarative policy language $\mathcal{PDL}$ was proposed. $\mathcal{PDL}$ programs are sets of ECA (Event-Condition-Action) rules. A paramount issue for such programs is how to detect and resolve action conflicts [7, 9, 11]. We characterized action conflicts as violations of *action constraints* in [7] and [9], in the context of a subset of $\mathcal{PDL}$ containing rules for which the event part of the rule consisted of primitive events and boolean operators. The actions produced by such rules depend only on the current set of input events; past events are irrelevant. Thus policies specified by such rules may be termed *stateless.* (They can be formally defined as mappings from event sets to action sets.) We proposed the *cancellation* of conflicting actions as a basic technique for resolving conflicts. We also showed how to implement cancellation monitors–the procedures for detecting and resolving conflicts.

However, the flow of time is important for policies. Not only concurrent but also sequential occurrence of events is meaningful. There are rules with *sequence events* in the full $\mathcal{PDL}$. The actions produced by such rules depend not only on the current set of input events but also on the history of past events. Time adds also another dimension to conflict resolution. Conflicting actions can be *delayed* until they can be executed without conflict. This observation leads to another class of monitors, namely *delay* monitors. To illustrate the difference between cancellation and delay monitors, consider the following example.

Example 1. Suppose we want to restrict concurrent access to a resource, so if there are two or more concurrent conflicting requests for the resource, only one can be granted. A cancellation monitor will ignore all but one of such requests. A delay monitor will delay the ignored requests instead, until they can be executed without conflict.

The criteria to select the appropriate monitor in the context of a specific application must consider the properties of conflicting actions in more detail. If a conflicting action is unlikely to be repeated and makes sense even when it is delayed, then it is better to use a delay monitor. Otherwise, a cancellation monitor is more appropriate.

Actions should not be cancelled (or delayed) arbitrarily. Clearly, a monitor should be as close to the policy as possible. This can be interpreted in two ways. If we focus on canceling (or delaying) a minimal set of conflicting actions, we get *action* monitors. On the other hand, if we choose to apply the original policy to a maximal consistent reduction of the original input, we get *event* monitors.

To appreciate the need for event monitors, consider the following observation. Sometimes a group of actions is always executed in tandem and if one of them is cancelled, all of them must be cancelled. This is similar to a database transaction. For example, suppose a piece of merchandise ordered by a customer arrived at the stock location. After this event happens two actions are executed: the piece of merchandise is shipped to the customer and the customer credit card is charged. If the shipment is stopped by a conflict with another action (such as a recall of the product due to a manufacturing defect), the charge should not be made. This dependency is captured by event monitors. In action monitors, two actions caused by the same event (or events) can be cancelled or delayed independently.

The last dimension of time addressed in the present paper deals with *temporal* action constraints. Such constraints make it possible to restrict not only concurrent but also sequential execution of actions.

To sum up, we extend in this paper the approach to conflict resolution proposed in [7] and [9] to wider classes of policies and monitors. We show how to define monitors for policies with rules that refer to *sequence events*. We also show how to specify *delay monitors*. Finally, we address the issue of more general action constraints formulated in *Past Temporal Logic*. All the above extensions are founded on a more general notion of policy: a mapping from sequences of event sets to sequences of action sets. In fact, the monitors defined in this paper are applicable not only to $\mathcal{PDL}$ but also to any other policy language with the formal semantics of the same kind.

The plan of the paper is as follows. In Section 2, we provide the basic definitions. We introduce the syntax of $\mathcal{PDL}$ and define action constraints. In Section 3, we define monitors, introduce several basic classes of monitors and show how such monitors may be computed. In Section 4, we introduce temporal action constraints and show how to accommodate such constraints in the monitors. Section

5 describes our current implementation of a $\mathcal{PDL}$ policy server. In Section 6, we briefly survey related work. We conclude the paper in Section 7.

2 Definitions

In [14], Lobo, Bhatia and Naqvi introduced the language $\mathcal{PDL}$, a policy description language that is being used as a programming language of a policy-based network management system [17]. We consider here a subset of $\mathcal{PDL}$.

2.1 Policies

A policy can be described as a reactive system that observes events happening in the environment and reacts to them by trying to affect the environment through the execution of actions (procedure calls). The reaction could be caused by the occurrence of a single event or a number of events.

To write programs that implement policies we assume that both the events observed by a policy and the actions that the policy generates have the structure of terms. We fix two disjoint set of function symbols: *primitive event* symbols and *action* 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. We refer to the arguments of a primitive event as its *attributes* and to the arguments of an action as its *parameters*. Every attribute and parameter has an associated type.

Definition 1. *A* policy *is a finite collection of well-typed policy rules of the form*

$$\textit{event}\ \textbf{causes}\ \textit{action}\ \textbf{if}\ \ \textit{condition}, \tag{1}$$

where the event, action *and* condition *parts of a rule are defined below.*

Definition 2. *The* event *part of a policy rule is either an expression of the form:*

1. $e_1 \& \ldots \& e_n$ *where each* e_i *is an event term (defined below) and* & *is interpreted as a* conjunction *of events; or an expression of the form*
2. $e_1, \ldots, e_n$ *where each* e_i *is a conjunction of event terms and "," is interpreted as a* sequence *of events; or an expression of the form*
3. $[e_1, \ldots, e_n]$ *where each* e_i *is a conjunction of event terms and the sequence is interpreted as a* relax-sequence *of events.*

An event term *is a typed 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. An* event instance *is a ground event term (i.e., an event term without variables).*

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 either (1) a variable that appears in the event part of the rule, (2) a constant, or (3) a well-formed expression of variables, constants and operations from the standard types. 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 either (1) a variable that appears in the event part of the rule, (2) a constant, or (3) a well-formed expression of variables, constants and operations from the standard types. The condition represents the conjunction of the predicates.*

In general, several events can occur simultaneously in the environment. We refer to the collection of event instances that are considered to occur simultaneously as an *epoch*. From the policy point of view, the epoch defines simultaneity. The implementation of the concept though is domain-dependent. In some cases the right epoch granularity is an hour, in other a day, or even an arbitrarily defined period of time. An implementation of epochs is described in Section 5. To a policy the environment is presented as a finite sequence of epochs to which the policy responds by generating actions.

Definition 5. *A finite set of event instances is an* epoch. *A finite set of actions is an* action set. *A finite sequence of epochs is called an* E-history, *and a finite sequence of action sets an* A-history.

Definition 6. *A substitution is a function that maps typed variables to constants of the appropriate type. The application of a substitution σ to a term t, denoted by $t\sigma$, is the simultaneous replacement of the variables in t that are in the domain of σ by the constants assigned to the variables by σ.*

Note: Our notion of substitution corresponds to that of the *ground* substitution in logic programming.

We say that:

- an event e *occurs* in an epoch if an instance of the event term $e(X_1, \ldots, X_n)$ is member of the epoch[1];
- a conjunction $e_1 \& \ldots \& e_m$ of event instances occurs in an epoch if each e_i, $1 \leq i \leq m$, occurs in the epoch.

Definition 7. *Given an E-history $\mathcal{H}_E = (E_1, \ldots, E_n, E_{n+1})$, we say that*

1. *the conjunction of event instances $e_1 \& \ldots \& e_m$ occurs in $\mathcal{H}_E$ if it occurs in E_{n+1},*
2. *the sequence of conjunctions of event instances $e_1, \ldots, e_m$ occurs in $\mathcal{H}_E$ if every e_i, $1 \leq i \leq m$, occurs in $E_{n+1-(m-i)}$,*
3. *the relax-sequence of conjunctions of event instances $[e_1, \ldots, e_m]$ occurs in $\mathcal{H}_E$ if there is a sequence of epochs $E_{j_1}, \ldots, E_{j_m}$ such that (1) $1 \leq j_1 < \cdots < j_m = n+1$, (2) for all i, $1 \leq i \leq m$, e_i occurs in E_{j_i}, and (3) for all i, $1 \leq i \leq m-1$, e_{i+1} does not occur in any epoch between E_{j_i} and $E_{j_{i+1}}$.*

The difference between sequence and relax-sequence of events is that in the latter the events do not have to appear in *consecutive* epochs.

[1] In the term the X_i's are distinct variables of the appropriate type.

Definition 8. *The semantics of a* policy P *is recursively defined as the following function* T_P *from E-histories to A-histories:*

1. *For an E-history* $\mathcal{H}_E = (E_1)$*, of length 1,* $T_P(\mathcal{H}_E) = (A_1)$ *iff for every* $a \in A_1$*, there is a policy rule "*E **causes** A **if** C*" in* P *and a substitution* σ *such that* $E\sigma$ *occurs in* $\mathcal{H}_E$*,* $C\sigma$ *is satisfied and* $A\sigma = a$*.*
2. *For an E-history* $\mathcal{H}_E = (E_1, \ldots, E_{n+1})$*, of length* $n+1$*,* $T_P(\mathcal{H}_E) = (A_1, \ldots, A_n, A_{n+1})$ *iff for every* $a \in A_{n+1}$*, there is a policy rule "*E **causes** A **if** C*" in* P *and a substitution* σ *such that* $E\sigma$ *occurs in* $\mathcal{H}_E$*,* $C\sigma$ *is satisfied,* $A\sigma = a$*, and* $T_P((E_1, \ldots, E_n)) = (A_1, \ldots, A_n)$*.*

Note that by definition policies are prefix-closed. That is, for any E-history

$$\mathcal{H}_E = (E_1, \ldots, E_n, E_{n+1}),$$

if

$$T_P(\mathcal{H}_E) = (A_1, \ldots, A_n, A_{n+1})$$

then

$$T_P((E_1, \ldots, E_n)) = (A_1, \ldots, A_n).$$

This is essential for policies to be evaluable incrementally. The output of a policy on a given E-history never changes when the history is extended with further epochs.

Example 2. A CD club service wants to implement its customer policy using $\mathcal{PDL}$. The club gives bonuses to clients that place orders in two consecutive months if the total cost of the orders in those months is above a certain threshold c. A client can close an account in any month, and orders are shipped as soon as they are received. This policy can be written with three $\mathcal{PDL}$ rules:

$$\begin{aligned}
order(Cust, Cost, Itm)\, \mathbf{causes}\ & ship(Cust, Itm). \\
order(Cust, Cost_1, Itm_1), order(Cust, Cost_2, Itm_2)\, \mathbf{causes}\ & bonus(Cust) \\
\mathbf{if}\ & Cost_1 + Cost_2 > c. \\
close(Cust)\, \mathbf{causes}\ & closeAcc(Cust).
\end{aligned}$$

2.2 Action constraints

Independently of the policies there might be some restrictions imposed on the kind of A-histories that are considered possible or correct in the system. For example, there might be restrictions, identified by the policy administrators, that do not let two particular actions appear in the same action set in the history (i.e., the actions cannot be executed simultaneously). The restrictions on A-histories are called *action constraints*. Thus, action conflicts are captured as violations of action constraints.

Definition 9. *An* action constraint *is an expression of the form*

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

Each a_i in the expression is an action term and C a condition like in (1). Variables in C must also appear as parameters of the action terms. The informal reading of the constraint is: "never allow the simultaneous execution of the actions $a_1, \ldots, a_m$ if the condition C holds." The constraint formally represents the formula $\forall \neg(a_1 \wedge \ldots \wedge a_n \wedge C)$.

Example 3. Returning to our CD club example we can specify the restriction that we cannot simultaneously close an account and process a shipment associated with the same account using the constraint:

$$\textbf{never}\ \ ship(Cust, Item) \wedge closeAcc(Cust).$$

Example 4. We extend the CD club example with several rules about enrollment. Assume that a customer receives an initial offer upon enrollment but can enroll only once. Subsequent attempts to enroll by the same customer result in a declination. This can be expressed in $\mathcal{PDL}$ as:

$$\begin{array}{r l} enroll(Cust)\ \textbf{causes} & offer(Cust). \\ [enroll(Cust), enroll(Cust)]\ \textbf{causes} & decline(Cust). \end{array}$$

and the constraint

$$\textbf{never}\ \ offer(Cust) \wedge decline(Cust).$$

To complete this example, we need to indicate that if both *offer* and *decline* are generated in an epoch, *decline* should have priority (being more specific). We will show how to do it in the next section.

Definition 10. *Let a be an action, $ac \equiv$ "**never** $a_1 \wedge \ldots \wedge a_m$ **if** C" an action constraint, and A an action set. Using the standard logical notation, we write $A \models \alpha$ to denote that A is a model of α. Specifically:*

1. *$A \models a$ iff $a \in A$;*
2. *$A \models ac$ if for every substitution δ, $C\delta$ is false or there is an i, $1 \leq i \leq m$, $A \not\models a_i\delta$.*

An A-history $\mathcal{H}_A = (A_1, \ldots, A_n)$ satisfies ac (resp. AC) if $A_n \models ac$ (resp. $A_n \models AC$). $\mathcal{H}_A$ is prefix-consistent with AC if for every $m \leq n$, $(A_1, \ldots, A_m)$ satisfies AC.

This definition generalizes to sets of constraints in an obvious way.

3 Monitors

A monitor of a set of action constraints generates only outputs without conflicts (without constraint violations).

Definition 11. *Given a set of action constraints AC, an AC-monitor ω_{AC} is a mapping from E-histories to A-histories of the same length such that for every E-history $\mathcal{H}_E$, $\omega_{AC}(\mathcal{H}_E)$ is prefix-closed and prefix-consistent with AC. (If the set of constraints AC is clear from the context, we will use the term "monitor" instead of "AC-monitor".)*

Notice that every monitor is prefix-closed, and thus – like a policy – can be evaluated incrementally. Our goal is to take policies together with action constraints and automatically generate monitors. We have identified two basic ways in which a monitor can handle conflicts. One is for the monitor to cancel some of the actions that generate the conflict. Such a monitor will be called a *cancellation monitor.* The other is to delay some conflicting actions until their execution does not cause conflicts. Such a monitor will be called a *delay monitor.* Within the classes of cancellation and delay monitors we introduce further subdivisions into *action* and *event monitors.* Intuitively, action monitors decide which actions to delay or cancel looking exclusively at the output of a policy (a set of actions). Event monitors, on the other hand, take also the input events into account.

Example 5. Consider again Example 2. Assume an *order* event and a *close* event occur in a single epoch. Therefore, because of the conflict between *ship* and *closeAcc*, one of those actions needs to be cancelled (or delayed). Assume *ship* is cancelled (delaying it does not make much sense). Assume also that another *order* event occurred in the preceding epoch and the condition for the *bonus* action to be executed is satisfied. The *bonus* action can be executed without conflict but it does not seem natural to do that since the order supporting the bonus was cancelled. An event-cancellation monitor avoids this problem by ignoring the event *order* and therefore also indirectly cancelling both actions it causes. Such a monitor selects a consistent reduction of the input epoch, in that case the reduction contains only the *close* event. Therefore, both *ship* and *bonus* are effectively cancelled.

In some cases ignoring all the actions caused by an event is not necessarily desirable.

Example 6. Consider Example 4 in the case when an enrollment event of a customer is followed by another enrollment event of the same customer. In this case the second event causes two actions: *offer* and *decline*, which result in a conflict. An event monitor cancels or delays that event, and consequently neither of the actions is output. Intuitively, this is not a correct behavior: we expect exactly one of them to be output in this situation.

A comprehensive policy management system should provide both action and event monitors, as well as their combinations. Consequently, we will have four basic classes of monitors: action-cancellation, event-cancellation, action-delay, and event-delay. The monitors will be defined through algorithms that construct them. Each algorithm is nondeterministic and thus defines a family of monitors. Each of those will, however, enjoy suitable maximality properties, as elaborated in the next section.

3.1 Action monitors

Notation: $\mathit{tail}((A_1, \ldots, A_n)) = A_n$.

The following algorithm computes

$$M^P_{ac}(E_1, \ldots, E_n) = (A_1, \ldots, A_n)$$

for some action-cancellation AC-monitor M^P_{ac} of P:

```
Algorithm 1 Action Cancellation Monitor
begin
   for i := 1 to n do
    A_i := ∅
    U := tail(T_P((E_1, ..., E_i)))
    while true do
      select a ∈ U − A_i such that A_i ∪ {a} ⊨ AC
      if select successful then A_i := A_i ∪ {a}
      else break
    end
   end
end
```

The following algorithm computes

$$M^P_{ad}(E_1, \ldots, E_n) = (A_1, \ldots, A_n)$$

for some action-delay AC-monitor M^P_{ad} of P:

```
Algorithm 2 Action Delay Monitor
begin
   D := ∅
   for i := 1 to n do
    A_i := ∅
    U := D ∪ tail(T_P((E_1, ..., E_i)))
    while true do
      select a ∈ U − A_i such that A_i ∪ {a} ⊨ AC
      if select successful then A_i := A_i ∪ {a}
      else break
    end
    D := U − A_i
    end
end
```

In some cases it is natural to define a priority ordering between actions. For example, if an action represents an exception, it should have priority over the more general action. In Example 4, we remarked that *decline*, being more specific, should have priority over *offer*. Action priorities can be easily added to the above algorithms: the actions in the **while** loop should be selected according to the priorities.

3.2 Event monitors

In this case, the set of actions is computed in two steps. First, a reduction of the input E-history is computed and then the policy applied to this set to obtain a set of actions without conflicts.

The following algorithm computes

$$M^P_{ec}(E_1, \ldots, E_n) = (A_1, \ldots, A_n)$$

for some event-cancellation AC-monitor M^P_{ec} of P:

`Algorithm 3` *`Event Cancellation Monitor`*

begin
  for $i := 1$ **to** n **do**
   $E' := \emptyset$
   while true **do**
    select $e \in E_i - E'$
    such that $tail(T_P(E_1, \ldots, E_{i-1}, E' \cup \{e\})) \models AC$
    if select successful **then** $E' := E' \cup \{e\}$
    else break
   end
   $A_i := tail(T_P(E_1, \ldots, E_{i-1}, E'))$
  end
end

The following algorithm computes

$$M^P_{ed}(E_1, \ldots, E_n) = (A_1, \ldots, A_n)$$

for some event-delay AC-monitor M^P_{ed} of P:

`Algorithm 4` *`Event Delay Monitor`*

begin
  $D := \emptyset$
  for $i := 1$ **to** n **do**
   $E' := \emptyset$
   while true **do**
    select $e \in D \cup E_i - E'$
    such that $tail(T_P(E_1, \ldots, E_{i-1}, E' \cup \{e\})) \models AC$
    if select successful **then** $E' := E' \cup \{e\}$
    else break

end
$A_i := tail(T_P(E_1, \ldots, E_{i-1}, E'))$
$D := D \cup E_i - E'$
end
end

3.3 Computational complexity

It is easy to see that all the above monitors can be computed in time polynomial in the number of events in the input E-history. In [9] we showed that the simulation problem for event-cancellation monitors of stateless $\mathcal{PDL}$ policies is NP-complete. That result does not contradict the above observation: The simulation problem requires that the monitor produce a given set of actions and thus is possibly more difficult than the problem of computing an arbitrary (maximal) monitor.

3.4 General properties of monitors

Note that according to the definitions, a monitor that cancels all the actions for any input is an AC-monitor for any policy P and set of constraints AC. It is even an action delay and an event delay monitor since the effect of such a monitor can be characterized as delaying the actions or the events for ever. However, it does not make sense to cancel (or delay) an action or an event if it is not involved in a conflict. If we could order the monitors in such a way that the higher the monitor in the order the closer its behavior is to the original policy, we would certainly like monitors that are maximal in this order. Such monitors would cancel or delay the minimum number of actions or events needed to eliminate conflicts. Furthermore, in the case of delay monitor actions and events should be delayed as little as possible.

We have developed formal characterizations of the orders suitable for comparing monitors in all the classes discussed in this paper: action cancellation/delay and event cancellation/delay. We have proved that the monitors defined by Algorithms 1-4 are maximal in these orders. Details of the definitions and proofs can be found in [8].

4 Temporal action constraints

In many applications, it is natural to impose constraints not only on concurrent but also on sequential execution of actions. For example, some actions should (or shouldn't) appear in a specific order. This kind of constraint cannot be directly captured within the framework described so far. However, the appropriate extension is rather easy.

We add *temporal connectives* to the language of action constraints. We choose the past connectives of linear-time temporal logic [6]. The language of action constraints allows now not only action terms but also *action expressions*. An action expression is:

1. an action term,
2. **previous** a where a is an action expression (meaning "a was executed in the previous epoch"), or
3. a_1 **since** a_2 where a_1 and a_2 are action expressions (meaning "a_1 has been executed in every epoch since a_2 was executed").

Formally, we define satisfaction for temporal action constraints by extending Definition 10.

Definition 12. *Let a be an action expression, $ac \equiv$ "**never** $a_1 \wedge \ldots \wedge a_m$ **if** C" a temporal action constraint, and $\mathcal{H}_A = (A_1, \ldots, A_n)$ an A-history. Now*

1. *$\mathcal{H}_A \models a$ iff:*
   - *(a) a is an action term and $A_n \models a$, or*
   - *(b) $a \equiv$ **previous** a' and $n > 1$ and $(A_1, \ldots, A_{n-1}) \models a'$, or*
   - *(c) $a \equiv a'$ **since** a'' iff for some i, $1 \leq i < n$, $(A_1, \ldots, A_i) \models a''$ and for all j, $i < j \leq n$, $(A_1, \ldots, A_j) \models a'$.*
2. *$\mathcal{H}_A \models ac$ iff for every substitution δ, $C\delta$ is false or there is an i, $1 \leq i \leq m$, $\mathcal{H}_A \not\models a_i\delta$.*

Using this language, one can formulate the constraint that an action B shouldn't be executed before A as:

$$\textbf{never} \;\; A \wedge (\mathit{true} \; \textbf{since} \; B)$$

where *true* is an action executed in every epoch (there are several ways to define such an action).

The monitors defined in Section 3 should be appropriately generalized. Now action constraints need to be evaluated not only in the last state in a history but in the entire history. The techniques that automatically derive what kind of auxiliary historical information needs to be kept in every state to avoid looking at the entire history are well known [6].

5 System implementation

In this section we present an algorithm for evaluating policies specified using $\mathcal{PDL}$. The algorithm is implemented as the Policy Engine of a Policy Server embedded in the "softswitch", a next generation switch for circuit and packet telephony networks, and has been used to implement policies for detecting alarm conditions, fail-overs, device configuration and provisioning, service class configuration, congestion control etc., [17]. This implementation covers the full version of $\mathcal{PDL}$, which is more general than the language we consider in the paper. The softswitch manages an unbounded number of policy servers that are able to run policies written in $\mathcal{PDL}$.[2] When a policy is loaded into a policy server the server creates a policy evaluator for the input policy, contacts the devices (i.e., routers,

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

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 a 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 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. The evaluator works by simulating the finite automata encoded in the *event* part of a policy rules. The transitions of the automata are labeled by set of primitive event symbols. Sequences are translated directly: there is a transition and a state per each conjunction of events in the sequence plus the initial state. For relax-sequences there is also a transition and a state per each conjunction of events, but in addition there is a self-loop transition coming out from each state that skips irrelevant events until the next conjunction of events in the relax-sequence appears.

We present the algorithm for a single policy rule since rules can be evaluated independently of one another. At any epoch t the algorithm maintains for the policy rule, the set $R(t)$ of all its possible distinct partial evaluations in the event history that may lead to the triggering of an action in a rule in some future epoch. We refer to these distinct partial evaluations as *active threads* and they are built as follows. Let a "sub-epoch" of an epoch refer to a subset of the primitive events in the epoch. Note that a "sub-epoch" is by definition an epoch. Given an E-history $\mathcal{E} = (E_1, \ldots, E_n)$, a sequence of "sub-epochs" $\mathcal{E}' = (E'_1, \ldots, E'_n)$ is a sub-history of $\mathcal{E}$ if and only if $E'_i \subseteq E_i$ for $1 \leq i \leq n$.

An active thread $A(t)$ at epoch t is maintained as a tuple $(A_1(t), A_2(t))$ where $A_1(t)$ is a path in the automaton for the event E of the policy rule and $A_2(t)$ is a sub-history (called the partial trace) of a suffix of the history at epoch t. The path $A_1(t)$ starts from an initial state of the automaton and is the path taken when the automaton is simulated on input $A_2(t)$. An active thread also carries with it the attribute values of the different events that come from the partial trace $A_2(t)$ and are necessary to evaluate the condition and the action of the policy rule. Note that an active thread may have, in a future epoch, enough information to fully evaluate the policy rule. The algorithm ensures that every active thread leads to a distinct evaluation of the policy rule. Each time an epoch is evaluated new threads are started and active threads are moved or killed according to their location in the automata. Details of the algorithm and complexity results on policy evaluation can be found in [3]. Following the policy evaluations, each policy evaluator returns a set of 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

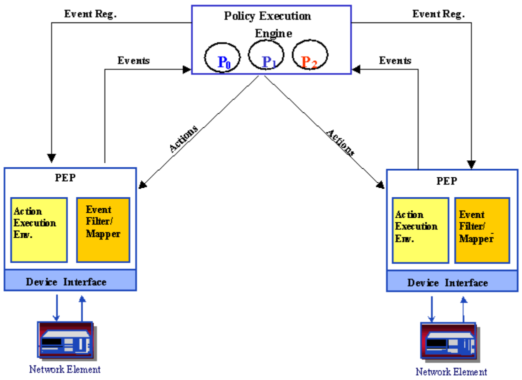

Fig. 1. System Architecture

the PEPs which are device-dependent.

Given that monitors, like policies, are prefix-closed, we can compute the A-history output by the monitor incrementally: when a new epoch is input only the new action set of the corresponding expanded A-history is computed.

To implement an action cancellation monitor we only need to intercept the set of actions generated by a policy evaluator each time an epoch is evaluated. Then, we run the **while** loop in Algorithm 1 setting U to be the set of actions returned by the evaluator and $i = n$. The actual action set sent to the PEPs is the resulting set A_n.

To implement an event cancellation monitor, following Algorithm 3, E_i is set to the current epoch, and $T_P((E_1, \ldots, E_{i-1}, E' \cup \{e\}))$ is replaced by a simulation of the policy evaluation with $E' \cup \{e\}$ as the input epoch. The simulation is done to trap the actions triggered by the epoch and check whether the set of action constraints AC is violated. If the AC set is violated, the selection fails and a new event has to be selected. After all the events from E_i have been covered and E' has been completed, an actual execution (not a simulation) of the policy evaluation is done with E'. The set of actions generated by the execution is sent to the PEPs.

The implementation of delay monitors is a little bit trickier since the information that events or actions are being delayed must be passed from the current epoch to the next epoch. This effect is achieved by introducing “fake” events and new rules triggered by these events into the policy. In the case of action delay we introduce a new event symbol e_a for every action symbol a with the same arity as a. Also, we extend the original policy P to an extended policy P' that contains all the rules of P plus a rule of the from “$e_a(X_1 \ldots, X_n)$ **causes** $a(X_1 \ldots, X_n)$” for each action symbol a. If an action $a(t_1, \ldots, t_n)$ is ignored, the corresponding event $e_a(t_1, \ldots, t_n)$ will be added to the next epoch by the policy evaluator.

Similarly, for event delay, we introduce a new event symbol e' for every original event e. Also, we extend the original policy P to an extended policy P' that contains all the rules of P plus the copies of the rules in P in which every event e is replaced by e'. When an event is ignored, the event e' is added to the incoming epoch. If the event e' is ignored again, the same e' is added to the next epoch.

6 Related work

Conflict resolution for production rules in AI and databases has been addressed in [1, 4, 12]. Results about the complexity of testing consistency of production rules can be found in [5]. However, in contrast to our view, those works assume interpreted actions (variable assignments or database updates) and mostly ignore the event part of the rules. Also, conflicts are typically between rules, not actions. The work in [13] deals with a model that is closer to ours, although the conflicts studied are still between rules, not actions, and the events are not taken into account.

The notion of *action constraints* was independently introduced in [7] and [11]. Conflict resolution is only one of the many issues addressed in [11] and the authors limit themselves to proposing a construct equivalent to maximal action cancellation monitors for stateless policies. Event cancellation, sequence events, conflict resolution through delay, or temporal action constraints are not considered.

7 Conclusions

In this paper we have studied conflict resolution for history-based policies. We have defined monitors (procedures for resolving action conflicts) and identified several dimensions of monitors: action vs. event-based (introduced in [9] in the context of stateless policies), cancellation vs. delay-based (new). We have provided polynomial algorithms for computing maximal monitors in each class.

The techniques presented in this paper can be generalized to more general policies. For instance, handling negated events in history-based policies can be done along the same lines as in stateless policies [9]. The solution proposed there is based on making ignored events undefined, as opposed to making their negations true.

We envision several directions for future research. It should be worthwhile to study further classes of monitors, in particular *hybrid* monitors that allow cancellation of some events and delay of others. Another direction is static analysis of policies. Perhaps some policies never lead to conflicts – for them monitors represent an unnecessary overhead. This question becomes interesting in the presence of negated events or additional information about events. Work on detecting statically potential conflicts is reported in [15]. This work might be used to generate the action constraints we need as input for the monitors. Still another direction consists of studying the expressive power of different subsets of $\mathcal{PDL}$.

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. C. Baral, J. Lobo, and G. Trajcevski. Formal characterizations of active databases: II. In *Proc. of the International Conference on Deductive and Object Oriented Databases*, Lecture Notes in Computer Science. Springer, Switzerland, December 1997.
3. R. Bhatia, J. Lobo, and M. Kohli: Policy Evaluation for Network Management. In *Proc. of the 19th Conference on Computer Communication*, INFOCOM 2000. Israel, March 2000.
4. L. Brownston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.
5. H. Kleine Büning, U. Löwen, and S. Schmitgen. Inconsistency of production systems. *Journal of Data and Knowledge Engineering*, 3:245–260, 1988/89.
6. J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, 20(2):149–186, June 1995.
7. J. Chomicki, J. Lobo, and S. Naqvi. Axiomatic conflict resolution in policy management. Technical Report ITD-99-36448R, Bell Labs, February 1999.
8. J. Chomicki and J. Lobo. Monitors for History-Based Policies. Technical report, Lucent Bell Labs, 2000.
9. J. Chomicki, J. Lobo, and S. Naqvi. A Logic Programming Approach to Conflict Resolution in Policy Management. In *International Conference on Principles of Knowledge Representation and Reasoning*, Breckenridge, Colorado, April 2000.
10. T. Eiter and V.S. Subrahmanian. Heterogeneous active agents, II: Algorithms and complexity. *Artificial Intelligence*, 108:257–307, March 1999.
11. T. Eiter, V.S. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108:179–255, March 1999.
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. J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of AAAI*, Orlando, FL, July 1999.

15. 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.
16. I. Niemelä. Towards efficient default reasoning. In *Proc. of 14th International Joint Conference on Artificial Intelligence*, pages 312–318, Montreal, 1995.
17. A. Virmani, J. Lobo, and M. Kohli. NETMON: Network management for the SARAS softswitch. In *Proc. of the IEEE/IFIP Network Operations and Management Symposium*, April 2000.