From: AAAI-99 Proceedings. Copyright © 1999, AAAI (www.aaai.org). All rights reserved.

A Policy Description Language

Jorge Lobo Randeep Bhatia Shamim Naqvi
Network Computing Research Department
Bell Labs
600 Mountain Av, Murray Hill, NJ 07974
{jlobo,randeep,shamim}@research.bell-labs.com

Abstract

A policy describes principles or strategies for a plan of action designed to achieve a particular set of goals. We define a policy as a function that maps a series of events into a set of actions. In this paper we introduce $\mathcal{PDL}$, a simple but expressive language to specify policies. The design of the language has been strongly influenced by the action languages of Geffner and Bonet (Geffner & Bonet 1998) and Gelfond and Lifschitz (Gelfond & Lifschitz 1993) and the composite temporal event language of Motakis and Zaniolo (Motakis & Zaniolo 1997). The semantics is founded on recent results on formal descriptions of action theories based on automata and their application to active databases. We summarize some complexity results on the hardness of evaluating polices and briefly describe the implementation of a policy server being used to provide centralized administration of a soft switch in a communication network.

Introduction

In AI, a policy is usually defined as a complete mapping from states (of the world) to actions (Russell & Norvig 1995). In system management, a policy describes principles or strategies for a plan of action designed to achieve a particular set of goals identified by the managers of the system. This high level description of management applies to very general situations such as establishing policies to increase inter-departmental interactions in a company to more technical situations such as defining traffic policies to reduce overload in a computer network. Policies can be specified at different levels of abstraction. At a very elementary level we find policies specified using production rules as in the standard AI definition. At a very high level we can find policies specified using natural language. In this document we are interested in some intermediate point for which we still have an effective computational model.

We define a policy as a function that maps a series of events into a set of actions. In contrast to the standard AI definition of policies, we assume that there is an intermediate service (with sensors) between the policy server and the environment that continuously polls the environment and communicates to the policy server only the changes in the environment (events) that may require the enforcement of a policy. This concept is not new, and it has been informally used in network management. Network management requires the definition of policies to specify configuration parameters, to handle faults, to ensure certain level of performance, to handle security and accounting. As the examples in the paper will show, this model highly simplifies the description of policies.

Below we describe $\mathcal{PDL}$, a simple but expressive language to specify policies. The design of the language has been strongly influenced by the *state language* of actions of Geffner and Bonet (Geffner & Bonet 1998), the action description language $\mathcal{A}$ of Gelfond and Lifschitz (Gelfond & Lifschitz 1993) and the composite temporal event language of Motakis and Zaniolo (Motakis & Zaniolo 1997). It uses the *event-condition-action* rule paradigm of active databases (Widom & Ceri 1995), a successor of the production rule paradigm of languages such as OPS5 (Brownston *et al.* 1985). In fact, *our language can be described as a real-time specialized production rule system to define policies.*

An *event-condition-action* rule is triggered when the event occurs, and if the condition is true the action is executed. In general, besides the informal definition of an active rule, there is no consensus about the definition of events, how to process rules when several of them are triggered simultaneously, and how to characterize the set of possible actions. There is a tendency to consider events modifications or access to the database, conditions to be queries to the database, and actions either modifications to the database or remote procedure calls. The operational semantics is informally described with wide variations from system to system and it is tightly coupled to the concept of transaction, making the database semantics the central point for the interpretation of rules. In contrast, we have developed a domain-independent semantics of our rules by making the events, conditions and actions parameters of the language. We have also concentrated in a precise description of the execution since we are interested in predicting the efficiency of the implementation of policies.

Copyright ©1999, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

We envision a system manager defining a policy in two steps. First, the manager will consult a policy server to obtain: 1) the set of events that the system is able to monitor (e.g. a router went down), 2) the set of actions that can be invoked by a policy (e.g. send e-mail to a user) and 3) the set of functions that system supports to evaluate the status of the environment (e.g. disk is 90% full). Then the manager will write policies (i.e. write a set of rules) by combining events, actions and functions from these sets. The policy server will take this policy and implement it in the system.

The most salient feature of the language is its declarative semantics. In our approach, a policy description defines a transition function that maps a series of events into a set of actions to be executed by the policy enforcer. This function is implementation-independent. The semantics is founded on recent results on formal descriptions of action theories based on automata (Baral, Gelfond, & Provetti 1997; Gelfond & Lifschitz 1993) and their application to active databases (Baral, Lobo, & Trajcevski 1997). In the next section we describe the syntax of the language. Next, we present a series of examples to informally describe the semantics of the language. Then we present the formal semantics in terms of transition functions and briefly describe how the semantics can be casted in terms of logic programs. The next section describes complexity issues associated with the implementation of a policy server. We currently have an implementation of a server that supports a sub-set of the language. The server is being used to monitor software switches for telephone communication. Most of the examples in the paper have been inspired by the application. Some concluding remarks are found in the last section.

The syntax

$\mathcal{PDL}$ consists of three basic classes of symbols: *primitive event* symbols, *action* symbols and *function* symbols. The primitive event symbols are partitioned into two sets, the set of *system defined* primitive event symbols and the set of *policy defined* primitive event symbols. Action and function symbols and system defined symbols are pre-defined and are given to the user that defines the policies. Policy defined primitive event symbols will be defined by the user. There is also a set of standard domains and types such as integers, floats, string of characters, etc. and possibly complex types such as stacks, queues, etc. depending on the domain of application.

Action and function symbols are of different arities. There might be symbols of arity 0. Each action symbol of arity n denotes the name of a procedure that takes n arguments (also called parameters) each of a particular type. A function symbol of arity n denotes a function that takes n arguments of a particular type and returns the value of another type. If $n = 0$, the function symbol represents a constant from one of the given domains. The type of a function is the type of the value returned by the function.

Policies are described by a collection of propositions of of two types, policy rule propositions and policy defined event propositions. *Policy rule* propositions are expressions of the form:

event **causes** *action* **if** *condition*

The intuitive reading of this proposition is: if the *event* occurs in a situation where the *condition* is true then the *action* will be executed.

As an example, suppose we have a pool of modems to provide two customers access to Internet services. We have assigned the number 5559991 to Customer1 and the number 5559992 to Customer2. We have 20 modems in our pool. There can be simultaneous connections from the same customer to the pool. All modems in the pool are shared by both customers but the server can be configured to limit the connections per number. We would like to allow a maximum of 15 connections to Customer1 during the day and 5 to Customer2. During the night we will allow a maximum of 10 to each customer.

The event to monitor is time. We will have a symbol associated with this event, say $CoarseTimeEvent$. This symbol represents a class of events for which instances occur four times a day, at 6:00am, 12:00pm, 6:00pm and 00:00am. In the *condition* part of the propositions describing the policy we will need to check when an instance of the *event* occurs to take the appropriate action. Thus, we will extend the $CoarseTimeEvent$ with an attribute named $Time$. In general, events will have a set of attribute names associated with them, each one having an associated type. For our example, the type of $Time$ will be the enumerated type {"morning", "noon", "evening", "midnight"}. We will use the standard dot "." notation to refer to the attributes of an event. An *instance* of an event is given by a complete denotation of its attributes. We will use one action in the example, $ModemPoolAssigment$. The signature of this action is Telephone_Numbers$\times\{1, 2, \ldots, 20\}$. When this action is executed the configuration of the pool is changed to limit the maximum number of connections of the telephone number given in the first argument to be the number given in the second argument.

The following two propositions cover the actions required in the morning:

1) $CoarseTimeEvent$
   causes $ModemPoolAssigment(5559991, 15)$
   if $(CoarseTimeEvent.Time = \text{"}morning\text{"})$.
2) $CoarseTimeEvent$
   causes $ModemPoolAssigment(5559992, 5)$
   if $(CoarseTimeEvent.Time = \text{"}morning\text{"})$.

Similarly, there will be two more rules in the policy that will set the number of modems to 10 for both customers in the evening. The four propositions together define the policy.

Before we introduce policy defined event propositions we need to take a closer look at the notion of event. Policies may depend on several events or the lack of

certain events happening, or even on events happening in the past. We can have the following situations:

1. A policy must be enforced if two events e_1 and e_2 occur simultaneously.
2. A policy must be enforced if an event e does not occur.
3. A policy must be enforced if an event e_2 immediately follows an event e_1.
4. A policy must be enforced if an event e_2 occurs after an event e_1 occurs.

Thus, policy decisions are made after a pre-determined stream of primitive event instances is observed by the policy service running the policy. We will call the streams of event instances *event histories*. There may be several instances of one or more primitive events occurring at the same time. Each set of primitive event instances occurring simultaneously in a stream is called an *epoch*. An *event literal* is an event symbol e or an event symbol preceded by !. The event literal $!e$ occurs in an epoch if there are no instances of the event e in the epoch.

Definition 1 *A basic event is an expression of the form*

1. $e_1 \& \ldots \& e_n$ *representing the occurrence of instances of* e_1 *through* e_n *in the current epoch (i.e. the simultaneous occurrence of the n events) where each* e_i *is an event literal, or*
2. $e_1 | \ldots | e_n$ *representing the occurrence of an instance of one of the* e_i*s in the current epoch. Each* e_i *is an event literal.*

Basic events only refer to instances of events that occur in a single epoch, but we could have composite events that refer to several epochs simultaneously. For example, the sequence $loginFail, loginFail, loginFail$ may represent the event: "three consecutive attempts to login that result in failure." In general $e_1, \ldots, e_{n-1}, e_n$ may represent the moment when an instance of the basic event e_n occurred in the current epoch, immediately preceded by an instance of the basic event e_{n-1} (i.e. an instance of e_{n-1} occurred in the previous epoch), ... , with an instance of the basic event e_1 occurring $n-1$ epochs ago.

We can describe many classes of sequences if we borrow the notion of a sequence of zero or more events from regular expressions. We will denote zero or more occurrences of an event E by "$\hat{}E$". Formally,

Definition 2 *An event is either* **(a)** *a basic event,* **(b)** *group(E) where E is a basic event, or* **(c)** *an expression that can be formed by a finite number of applications of the following rules:*

1. *If* E_1 *through* E_n *are events then* $E_1, \ldots, E_n$ *is a (complex) event representing the sequence: event* E_1*, immediately followed by* E_2*, ... , immediately followed by* E_n*.*
2. *If E is an event then* $\hat{}E$ *is a (complex) event representing the sequence of zero or more occurrences of the event E.*
3. *If E is an event then (E) is a (complex) event.*

To motivate the meaning of *group* note that if we have a history where there are n instances of event e_2 in the current epoch and m instances of event e_1 in the previous epoch, there will be a total of $n \times m$ instances of the event (e_1, e_2) in the history. There are situations in which we are not interested in single instances of a basic event but on a global property of all the instances in a epoch. For example, we can describe "a network failure (nf) followed by at least one disc crash (dc)" with $(nf, group(dc))$. In this case if we have m occurrences of nf and n of dc there will be only m occurrences of the complex event. If we have $(group(nf), group(dc))$ then there will be only one occurrence of the event. We will see examples of the "$\hat{}$" and *group* operators in the next section.

The last class of events we will consider are *policy defined* primitive events. So far, an event is defined as a combination of primitive events that have occurred in an event history. Policy defined primitive events allow us to mark incoming epochs with new event instances that we can later use as a component of a complex event. They provide a policy with a limited amount of memory. To define an event we will use policy defined event propositions, but we need to define terms first. Recall that it was assumed that there is set of types known by the user.

Definition 3 *A term is either*

1. *A constant from one of the known types, or*
2. *An expression of the form* $f(t_1, \ldots, t_n)$*, where f is a function symbol of arity n and each* t_i *is a term of the appropriate type, or*
3. *An expression of the form* $e[k].m$*, where e is a primitive event, k is positive integer, m is an attribute associated with e.*

The index in the event term is to distinguish different occurrences of the same primitive event symbol in a complex event expression. The index is assigned left to right in increasing order. If there is no ambiguity in the reference the index can be omitted. The denotation of a term of the form $f(t_1, \ldots, t_n)$ *is the result of calling a function (or script) that will be associated with the function symbol f with the values that the internal terms denote as parameters. We will use infix notation for function symbols with well-understood meanings such as +, *, etc. We will also assume a cast system translation from types such as the one in C or Java.*

A *policy defined event* proposition is an expression of the form:

event **triggers** *pde*$(m_1 = t_1, \ldots, m_k = t_k)$
if *condition*

pde is a policy defined primitive event symbol. m_1 through m_k are the attributes associated with the de-

fined event symbol *pde* and each t_i is a term of the appropriate type. The intuitive reading of this expression is: If the *event* occurs in a situation where the *condition* is true, an instance of the primitive event *pde* will occur in the immediately following epoch with the valuation of each t_i as the value assigned to each attribute m_i of *pde*.

Complex events in policy propositions can include policy defined or a system defined primitive events.

With the definition of terms we can also give a precise characterization of actions and conditions.

Definition 4 *An* action *is an expression of the form* $a(t_1, \ldots, t_n)$*, where* a *is action symbol of* n *arguments and each* t_i *is a term of the appropriate type.*

A condition is a expression of the from $p_1, \ldots, p_n$*, 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* t_1 *and* t_2 *are terms of the same type.*

Examples

Soft switch overload control: Suppose we characterize overload as an excessive number of signaling network time-outs over calls made, let say at a ratio of t. If an overload occurs some call requests must be rejected until the time-out rate goes down to a reasonable number, say t'. We can define this policy as follows.

Events:
normal_mode : policy defined event.
restricted_mode : policy defined event.
call_made
time_out

The events do not have attributes.

Actions: *restrict_calls*, *accept_all_calls*

Policy description:

1) *normal_mode*, ^(*call_made*|*time_out*)
   triggers *restricted_mode*
   if $Count(time\_out) > t * Count(call\_made)$.
2) *restricted_mode* **causes** *restrict_calls*.
3) *restricted_mode*, ^(*call_made*|*time_out*)
   triggers *normal_mode*
   if $Count(time\_out) < t' * Count(call\_made)$.
4) *normal_mode* **causes** *accept_all_calls*.

We assume that when the system starts the primitive event *normal_mode* is triggered. This can be accomplished by adding the event proposition "*power_on* **triggers** *normal_mode*" to the policy description. The complex event triggering the first policy defined event proposition occurs in an event history in which there is an epoch with an instance of the *normal_mode* event, followed by a sequence of epochs where there is an instance of either the event *call_made* or *time_out* in each epoch in the sequence. The *restricted_mode* event will be triggered the first time that the condition in the event definition is true. Note that as far as this policy is concerned there are only four types of events.

There is a special function symbol in the conditions of the policy description: "*Count*". This function symbol is a *temporal aggregate* that evaluates over the sequence of primitive event occurrences that form the complex event. In the example, the aggregate is used to count the number of occurrences of the events *time_out* and *call_made* in the sequence. In general, a temporal aggregate has the structure of a function call with a primitive event symbol or an attribute of a primitive event symbol as its argument. Typical temporal aggregates are *Sum*, *Avg*, *Min*, *Max*, etc. The condition in the first policy defined event proposition is true as soon as in the instance of the complex event the number of *time_out* events is larger than the number of *call_made* events times t.

The example above assumes that there can only be one *call_made* event or *time_out* event in each epoch. If several calls or time outs happen simultaneously there will be several instances of the complex event in the history. For example, in the history with the sequence of epochs

$\{normal\_mode\}$, $\{call\_made\}$,
$\{time\_out, call\_made, call\_made\}$,
$\{call\_made, call\_made\}$

there are six instances of the complex event *normal_mode*, ^(*call_made*|*time_out*) occurring at the last epoch of the history.[1] The intention of the policy is to group these instances into a single instance and count the timeouts and the calls made in the whole group. This is captured if the policy is defined in terms of the *group* operator as followed:

normal_mode, *group*(^(*call_made*|*time_out*))
  triggers *restricted_mode*
  if $Count(time\_out)/Count(call\_made) > t$.
restricted_mode **causes** *restrict_calls*.
restricted_mode, *group*(^(*call_made*|*time_out*))
  triggers *normal_mode*
  if $Count(time\_out)/count(call\_made) < t'$.
normal_mode **causes** *accept_all_calls*.

We use the notation $group(E)$, for a complex event E, as the shorthand of the event that results after replacing any basic event e in E with $group(e)$.

Routing: Suppose that in a communication network if the average gapping of a trunk group is larger than a given threshold (say g) in the last five hours re-route calls to a different trunk. Gapping is the interval necessary between requests sent to a telephone switch service control point (SCP) in order to process the requests on time.

Events:
window5 : policy defined event { *Start* : Date }

[1] There are three instances of the event (*call_made*|*time_out*) in the epoch {*time_out*, *call_made*, *call_made*}, and two in {*call_made*, *call_made*}.

hour :{ *Time* : Date }

Instances of *hour* will occur every hour on the hour. The attribute *Time* is set with the time when the event occurred. We are assuming the existence of a standard type "Date". Instances of the *window5* event will be triggered by the policy every hour (after the first four hours since the policy was enabled in the system). This event will mark the end of a 5 hour sliding window. *Start* is set to the time when the window begins.

Actions: *reroute* : Trunk Name.

When this action is executed the configuration of the server is changed to limit the calls that go through the trunk that is passed as argument of the action.

Policy description:

hour, ˜*hour*, *hour*
triggers *window5*(*Start* = *hour*[1].*Time*)
if *hour*[3].*Time* − *hour*[1].*Time* = 5.
hour, ˜(*call_gapping_t1*), *window5*
causes *reroute*(*trunk1*)
if *ave*(*call_gapping_t1.gap*) > *g*,
window5.Start = *hour.Time*.

The expression ˜*e* denotes a sequence of zero or more events ending in the basic event *e*. This is a shorthand for the complex event ˆ(ˆ!*e*, *e*).[2] The event *hour* occurs on the hour. The complex event *hour*, ˜*hour* occurs in a history any time an instance of the *hour* event has occurred in previous epoch and another instance of *hour* has occurred in the current epoch. The *window5* event is triggered the first time the second *hour* event instance occurs five hours apart from the first *hour* event. *call_gapping* occurs when this message is sent from an off the SCP to a soft switch.

The semantics

Policies are interpreted over *event histories*. An event history is a sequence of zero or more epochs. An *epoch* is a set of primitive event instances. The instance of a primitive event is a denotation for all attributes of the primitive event. There may be zero, one or more instances of a primitive event in a given epoch. Every epoch, in addition to the denotation of the attributes of each primitive event occurring in the epoch, will also have a denotation for all function symbols in the language. We will denote an epoch by a pair (S, D), with S the set of instances and D the denotation of function symbols. We denote the empty history by ϵ and assume there are no primitive event instances in ϵ.

An event history $\mathcal{H} = I_1 \ldots, I_n$, $0 \leq n$, is a *minimal* history of an event E iff one of the following conditions holds:

1. E is a primitive event, $n = 1$ (i.e. the history is just an epoch) and there is an instance i of E in $I_1 = (S, D)$. $((\{i\}, D), E)$ is called a *trace* of E in $\mathcal{H}$.
2. $E =!e$, $n = 1$ and there are no instances of e in $\mathcal{H}$. $(\mathcal{H}, E)$ is the (only) *trace* of E in $\mathcal{H}$.
3. $E = e_1 \& \ldots \& e_m$, $n = 1$ (i.e. the history is just an epoch), each e_j is an event literal, and there is a trace $((S_j, D), e_j)$ of e_j in $I_1 = (S, D)$ for every j, $1 \leq j \leq m$. $((\bigcup_{l=1}^{m} S_l, D), E)$ is a *trace* of E in $\mathcal{H}$.
4. $E = e_1 | \ldots | e_m$, $n = 1$ (i.e. the history is just an epoch), each e_j is an event literal, and there is an instance of e_j in I_1 for some j, $1 \leq j \leq m$. T is a *trace* of E in I_1 if T is a trace of e_j in I_1.
5. $E = group(E')$, E' is a basic event, $n = 1$, $\mathcal{H}$ is a minimal history of E', and the (only) trace of E in $\mathcal{H}$ is $((\cup\{S : (S, D) \text{ is a trace of } E' \text{ in } I_1\}, D), group(E'))$.
6. $E = E_1, \ldots, E_m$, and there exists a minimal history $\mathcal{H}_i$ for each E_i such that $\mathcal{H} = \mathcal{H}_1, \ldots, \mathcal{H}_m$. If T_1 is a trace of E_1 in $\mathcal{H}_1$, ... , T_m is a trace of E_m in $\mathcal{H}_m$, then $\mathcal{T} = T_1, \ldots, T_m$ is a *trace* of E in $\mathcal{H}$.
7. $E = \hat{}E'$ and either $\mathcal{H} = \epsilon$ and $(\mathcal{H}, null)$ is a *trace* of E in $\mathcal{H}$, or $\mathcal{H}$ is a minimal history of E', E and any trace of E', E in $\mathcal{H}$ is a *trace* of E in $\mathcal{H}$.
8. $E = (E')$ and $\mathcal{H}$ is a minimal history of E'. Any trace of E' in $\mathcal{H}$ is a trace of E in $\mathcal{H}$.

Before we define the satisfaction of a condition in a policy proposition we need to precisely define the denotation of terms. Since terms involve attributes of primitive events the denotation of a term will also involve traces of events. To simplify the presentation we will assume for the rest of this section that there is at most one occurrence of a primitive event symbol in a complex event.[3] We will describe the denotation of two aggregate functions, *Count* and *Avg*. Other aggregate functions can be defined similarly.

Definition 5 *Let $\mathcal{T}$ be a trace of an event in an event history. If $\mathcal{T} = ((S_1, D_1), e_1)), \ldots, ((S_n, D_n), e_n)$, and $e_n \neq null$, the denotation $t^{\mathcal{T}}$ of any term t in $\mathcal{T}$ is:*

1. *t, if t is a constant from one of the known types.*
2. *$f^{D_n}(t_1^{\mathcal{T}}, \ldots, t_k^{\mathcal{T}})$, if $t = f(t_1, \ldots, t_k)$ and f is a function symbol of arity k and f^{D_n} is the denotation of f in D_n.*
3. *$e.m^s$, if $t = e.m$ and m is an attribute associated with the event symbol e, there exists a unique j, $1 \leq j \leq n$ such that $e_j = e$, and s is the only instance of e in S_j. If there is no j such that $e_j = e$ in $\mathcal{T}$, or if there is more than one instance of e in S_j, the denotation of t is undefined.*
4. *The number of instances of e in $S_1, \ldots, S_n$, if $t = Count(e)$, and e is a event symbol, otherwise the denotation of t is undefined.*
5. *the average of $e.x^s$ for every instance s of e in $S_1, \ldots, S_n$, if $t = Ave(e.x)$ and $e.x$ is an integer attribute associated with e and the number of instances of e in $S_1, \ldots, S_n$ is $\neq 0$; otherwise the denotation of t is undefined.*

[2] If e is a non-primitive event, !e can be transformed into a basic event by repeated applications of De Morgan rules and a rule that cancels two consecutive !.

[3] Multiple occurrences can be handled by renaming or indexing the events (see the definition of terms).

If e_n = null and $n > 1$, the denotation of t in $\mathcal{T}$ is the same as the denotation of t in $((S,D_1),e_1)),\ldots,((S_{n-1},D_{n-1}),e_{n-1})$; otherwise $t^{\mathcal{T}}$ is undefined.

Truth values of predicates are obtained using the standard definition of the comparison operators. The only special consideration is when the denotation of a term that appears in a predicate is undefined. In that case the predicate will be undefined. A condition is true if every predicate in the condition is true; it is unknown if at least one predicate is unknown; otherwise is false. However, we will limit any reference made in a condition of a primitive event that occurs under the scope of a caret ^, or a *group* operator, or as part of a basic event of the form "$e_1|\ldots|e_n$" with $n > 0$, to only appear as an argument of an aggregate operator. It is easy to see that under this restriction a condition will never be undefined because of references to terms with undefined denotation. This restriction can be enforced syntactically. We will assume all policy descriptions obey the restriction.

Let $\mathcal{H} = I_1,\ldots,I_n$, be an event history. A policy defined event proposition "E **triggers** e **if** c" is *satisfied* in $\mathcal{H}$ iff the following conditions hold:

1. There exists an i such that $I_i,\ldots,I_n$ is a minimal history of E, and a trace $\mathcal{T}$, of E in this minimal history.
2. The denotation of c in $\mathcal{T}$, $c^{\mathcal{T}}$, is true.
3. There is no trace $\mathcal{T}'$ of E in $I_i,\ldots,I_k$ such that $\mathcal{T} = \mathcal{T}',\mathcal{T}''$ (i.e. $\mathcal{T}'$ a prefix of $\mathcal{T}$), and the denotation of c in $\mathcal{T}'$, $c^{\mathcal{T}'}$, is true.

$\mathcal{T}$ is called a satisfying *trace* of the event definition in $\mathcal{H}$. We can similarly define when a history satisfies a policy proposition of the form "E **causes** a **if** c". The third condition on the definition ensures that once a trace of an event satisfies a proposition, extensions of the same trace, due to the occurrence of a caret "^", do not satisfy the same proposition or event definition. For example, take the following history with four epochs, $(\{e_1\},D_1),(\{e_2\},D_2),(\{e_1\},D_3),(\{e_2\},D_4)$.
Assume that the condition c is satisfied in the prefix $(\{e_1\},D_1)$, $(\{e_2\},D_2)$, in the suffix $(\{e_1\},D_3)$, $(\{e_2\},D_4)$ and in the whole history $(\{e_1\},D_1)$, $(\{e_2\},D_2)$, $(\{e_1\},D_3)$, $(\{e_2\},D_4)$. The proposition "$\hat{}(e_1,e_2)$ **causes** $a(x)$ **if** c" is satisfied once by this history. The minimal history satisfying the proposition is the second pair $(\{e_1\},D_3)$, $(\{e_2\},D_4)$. Although the whole history is a minimal history of $\hat{}(e_1,e_2)$, it will not satisfy the proposition since a prefix of the history has already "consumed" the beginning of the event.

A history $\mathcal{H}$ is *plausible* for a policy description P iff either $\mathcal{H}$ is empty or $\mathcal{H} = I_1,\ldots,I_n$ and for every sub-history $\mathcal{H}_j = I_1,\ldots,I_j$, $1 \leq j < n$, of $\mathcal{H}$, and every policy defined event e, the following holds:

$\mathcal{H}_j$ satisfies event definition "E **triggers** $e(m_1 = t_1,\ldots,m_k = t_k)$ **if** c" in P iff there is an instance of e in I_{j+1} for each satisfying trace $\mathcal{T}$ of E in $\mathcal{H}_j$ with the denotation of $e.m_i$ in I_{j+1} is $t_i^{\mathcal{T}}$.

The underlying idea behind plausible histories is twofold: 1) it ensures that an instance of a policy defined primitive event occurs in an event history only if it is triggered by the satisfaction of a policy event definition and 2) it ensures that the instance is not spontaneously generated (i.e. it is only generated by a triggering event).

A policy description P defines a partial mapping π_P from event histories to sets of action symbols.

Definition 6 *Let P be a policy description. π_P : Histories $\longrightarrow 2^{Actions}$ is the policy defined by P iff* **(a)** *for every plausible event history $\mathcal{H}$ the following conditions hold:*

1. *For any satisfying trace $\mathcal{T}$ of a proposition of the form*

$$E \textbf{ causes } a(t_1,\ldots,t_n) \textbf{ if } c$$

in P, $a(t_1^{\mathcal{T}},\ldots,t_n^{\mathcal{T}}) \in \pi_P(\mathcal{H})$.
2. *Nothing else is in $\pi_P(\mathcal{H})$.*

(b) *If $\mathcal{H}$ is not plausible, $\pi_P(\mathcal{H})$ is undefined.*

A logic program for π_P

The following is a fragment of a logic program LP that implements the transition function π_P for any policy P. To save space we restrict the definitions to the main predicates. We use the standard PROLOG list notation to represent histories and epochs. A list representing a history stores the epochs in reverse order, most recent epochs are at the beginning, old epochs are at the end. Policy rules of the form "`E causes A if C`" are assumed to be added to the program as facts of the form `policyrule(E causes A if P)`. Rules of the form "`E triggers E' if C`" are stored as facts of the form `trigger(E triggers E' if C)`. We do not specify how to represent instances of primitive events and the denotations of functions but this is not important for understating the program. Note that the logic program is hierarchical (Shepherdson 1997). Thus, Clark completion gives us an equivalent first order logic definition of π_P and PROLOG with negation as failure a correct implementation.

```
exec(History,A) <-
      plausible(History),
      policyrule(E causes A if C),
      fired(E causes A if C, History).

plausible([]).
plausible([(Epoch,D)]).
plausible([(Epoch,D)|History]) <-
      NOT ignoredtrigger(Epoch,History),
      plausible(History).

ignoredtrigger(Epoch,History) <-
      trigger(E triggers E' if C),
      fired(E triggers E' if C,History),
```

```
      Not member(E',Epoch).

fired(E causes A if C,History) <-
      holds(E,History,Trace),holds(C,Trace),
      noHoldsInPrefix(E,C,Trace).

noHoldsInPrefix(Event,C,[]).
noHoldsInPrefix(Event,C,Trace) <-
      holds(Event,Trace,Trace),
      NOT holds(C,Trace),Trace = [E|Trace'],
      noHoldsInPrefix(Event,C,Trace').
noHoldsInPrefix(Event,C,Trace) <-
      NOT holds(Event,Trace,Trace),
      Trace = [E|Trace'],
      noHoldsInPrefix(Event,C,Trace').

holds(E,[(Epoch,Denotation)|H]),
           [(Inst,Denotation)]) <-
      instance(E,Epoch,Inst).
holds((E1,E2),History,Trace) <-
      append(H1,H2,History),
      holds(E1,H1,T1),holds(E2,H2,T2),
      append(Trace,T1,T2).
holds(^E,_,[]).
holds(^E,History,Trace) <-
      holds((E,^E),History,Trace).
```

holds(C,Trace) evaluates the condition C (attribute comparisons of the events in Trace and function denotations) in the most recent state.

```
instance(E,Epoch,[E]) <- member(E,Epoch).
instance(!E,Epoch,Epoch) <-
      Not member(E,Epoch).
instance(E1&E2,Epoch,Inst) <-
      instance(E1,Epoch,Inst1),
      instance(E2,Epoch,Inst2),
      union(Inst1,Inst2,Inst).
instance(E1|E2,Epoch,Inst) <-
      instance(E1,Epoch,Inst).
instance(E1|E2,Epoch,Inst) <-
      instance(E2,Epoch,Inst).
instance(group(E),Epoch,S) <-
      getinstances(E,epoch,S).
```

getinstances(E,Epoch,S) gets from the Epoch all the occurrences of the event E.

Proposition 1 *Given a policy P and a history $\mathcal{H}$ =* (Ep1,D1),...,(Epm,Dm)*;* A $\in$ $\pi_P(\mathcal{H})$ *iff* LP$\models$ exec([(Epm,Dm),...,(Ep1,D1)],A).

Complexity and algorithm

This section discusses the complexity of the policy evaluation problem. We show that even restricted instances are quite intractable. These hardness results provide us with insights that help us design an efficient algorithm for evaluating policies under very realistic assumptions. A full paper describing these results is under preparation.

The algorithm is implemented as the engine of a *policy server*, in the PacketStar IP Services Platform software developed at Bell Labs. The policy server is being used to provide centralized administration in circuit and packet telephony networks. It has been used to implement policies for detecting alarm conditions, fail-overs, device configuration and provisioning, service class configuration, congestion control etc.

We have established the hardness of the easier decision version of the problem: given a policy P of description size n and an action A and history $\mathcal{H}$ of length h, is action A caused by policy P in any of the epochs of $\mathcal{H}$ (i.e. $A \in \pi_P(\mathcal{H})$)?

Theorem 1 *The decision version of the policy evaluation problem is NP-Hard for any of the following restricted class of policies:*

- *Policies with one rule in which* **event** $= e_1, e_2 \ldots e_h$*, at most two primitive events per epoch in the history $\mathcal{H}$, no policy defined primitive events and n a polynomial in h.*
- *Policies with one rule in which* **event** *is a sequence of h ^e-s for a system defined primitive event e, at most one primitive event per epoch in the history $\mathcal{H}$, no policy defined primitive events and n a polynomial in h.*
- *Policies for which the description size n is a bounded constant but we allow policy rules using policy defined primitive events.*
- *Policies for which the description size n is a bounded constant, we do not allow policy defined primitive events, however the* **event** *in the policy rules may use "double non-determinism" (e.g.* **event** $=$ $\hat{}(\hat{}e_1, \hat{}e_2)$*).*

Policy evaluation algorithm: Although very restricted instances of the policy evaluation problem are hard to solve, the hard cases seem to arise only for contrived instances of the problem, i.e. policies used in practice do not belong to the restricted classes of policies for which Theorem 1 holds. For example policy rules tend to have small size descriptions (n is a bounded constant). We are able to design an efficient algorithm for evaluating such practical policies. The policy evaluation algorithm **PE** has two phases:

(1) Initialization phase This phase involves, among other things, the construction of a non-deterministic finite automata (NDFA) for the complex event E in the rule. The algorithm simulates the transitions in the NDFA in the real-time evaluation phase.

(2) Real-time evaluation phase At any epoch t the algorithm maintains the set $R(t)$ of all possible distinct partial traces for event E in the event history. A partial trace is defined to be a prefix of a trace. That is a partial trace at epoch t may after appending a trace from epochs $t+1, t+2 \ldots$ be converted into a trace for event E. We refer to these distinct partial traces as *active threads.* An active thread at epoch t is maintained as a path A_1 in the NDFA for E that starts from the initial state and a sequence A_2 of "sub-epochs" of a suffix of the history at epoch t. In other words A_2

is a partial trace at epoch t for event E in the event history, such that the epochs over which A_2 is defined form a suffix of the epochs of the history at epoch t. An active thread also carries with it the partial information (attribute values, aggregated values etc) that can be obtained from the partial trace A_2 to evaluate the condition, the arguments of the action or the triggered event in the proposition.

The algorithm **PE** works as follows. Let us say the algorithm has just seen the events in epoch t. By our assumption the algorithm has the set of active threads $R(t-1)$. Let $A(t-1) \in R(t-1)$ be an active thread. Let s be the last state of the NDFA, in the path A_1 for $A(t-1)$. We will say that the active thread $R(t-1)$ is in state s of the NDFA. Let (s, s') be a transition in the NDFA which is labeled with an event $E(i)_{(s,s')}$. Note that $E(i)_{(s,s')}$ is of the form e or $!e$ or $e_1 \& e_2 \& \ldots$ or $e_1 | e_2 | \ldots$, where each e, e_i is a basic event. Let I be the set of instances of $E(i)_{(s,s')}$ in epoch t. For each instance $k \in I$ a new active thread is computed, which has $A_1 =$ *append*(A_1 of $A(t-1), s'$), $A_2 =$ *append*(A_2 of $A(t-1), k$) and the additional information obtained for evaluating the policy rule from the instance k. At this point it is checked if this new active thread is in a final state of the NDFA and it has all the information required to evaluate the policy rule. If it is then the policy rule is evaluated in this active thread. Otherwise the active thread is added to the set $R(t)$. This procedure is carried out for every active thread in $R(t-1)$. In addition a new active thread $\omega(0)$ is added to $R(t)$ with $A_1 =$ the initial state of the NDFA, an empty trace and no information to compute the policy rule.

In the current implementation of the policy server, a policy registers with network devices the system defined primitive events that it is interested in. Whenever such a registered event happens at a device, a java object for the event is streamed to the policy. This may cause actions by the policy at network devices. An action is streamed to the network device as a java object whose methods are evaluated at the device server leading to device specific commands sent to the device.

Final remarks

Formal descriptions of general polices is a challenging problem for the KR community. We probably need a fairly sophisticated language if we intent to cover all classes of polices. For example, Subrahmanian and some of his collaborators at the University of Maryland use a deontic logic based language to describe policies in the IMPACT project, a platform for agent collaboration (Subrahmanian *et al.* 1998) to express very complex policies. Our approach to the design of the language has been guided by two principles: 1) We would like policy specifications that can be (efficiently) implemented and 2) We would like succinct representations of policies. This is our initial proposal but we expect the language to evolve.

There is a direction of research orthogonal to the design of the language. Once polices are written one needs to reason about them. We would like to prevent a policy from executing conflicting actions, i.e. we would like consistent policies. We will also need to reason about inter-policy interactions. Take the rules:

FaxArrive **causes** *DeliverFaxMainFloor*
if *FaxArrive.Size*$> \kappa$

FaxArrive **causes** *DeliverFaxHome*
if *FaxArrive.Time*$> 5:00p.m.$

These rules may be considered to be in conflict since there are situations where a single fax is delivered to two different places. For conflict verification it is essential to have languages with precise semantics. Much of the work on policy languages in network management lacks any type of formal semantics (see for example (Moffett & Sloman 1993; Wies 1994)). We are preparing a paper addressing some of the consistency issues.

References

Baral, C.; Gelfond, M.; and Provetti, A. 1997. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming* 31(1-3):201–244.

Baral, C.; Lobo, J.; and Trajcevski, G. 1997. Formal characterizations of active databases: II. In *Proc. of the International Conference on DOOD.*

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

Geffner, H., and Bonet, B. 1998. High-level planning and control with incomplete information using POMDP's. In *working notes of the AAAI fall symposium on Cognitive Robotics.*

Gelfond, M., and Lifschitz, V. 1993. Representing action and change by logic programs. *JLP* 17:301–321.

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

Motakis, I., and Zaniolo, C. 1997. Temporal aggregation in active database rules. In *Proc. of SIGMOD.*

Russell, S., and Norvig, P. 1995. *Artificial Intelligence - A Modern Approach.* Prentice Hall.

Shepherdson, J. C. 1997. Negation as failure, completion and stratification. In D. M. Gabbay, and Robinson, J. A., eds., *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 5. London: Oxford Science Publications. 356–420.

Subrahmanian, V. S.; Bonatti, P.; Eiter, T.; Dix, J.; and Kraus, S. 1998. IMPACT: Interactive Maryland Platform for Agents aCting Together. URL: http://www.cs.umd.edu/~vs/agent/impact.html.

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

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