Reasoning about Policies using Logic Programs

Tran Cao Son*
New Mexico State University
Computer Science Department
PO Box 30001
Las Cruces, NM 88003-8001
tson@cs.nmsu.edu

Jorge Lobo
Bell Labs - Lucent Technologies
600 Mountain Avenue
Murray Hill, New Jersey 07974
jlobo@research.bell-labs.com

Abstract

We use a simplified version of the Policy Description Language $\mathcal{PDL}$ introduced in (Lobo, Bhatia, & Naqvi 1999) to represent and reason about policies. In $\mathcal{PDL}$ a policy description is a collection of Event-Condition-Action-Rules that defines a mapping from event histories into action histories. In this paper we introduce *the generation problem*: finding an event history generating an action history, and state its complexity. Because of its high complexity we present a logic programming-based semantics for policies and encode sound and complete solution of the generation problem using the stable model semantics of logic programs. We also identify a subclass of policies where the generation problem can be solved in polynomial time.

Introduction

In AI, a policy is usually defined as a complete mapping from states (of the world) to actions (Russell & Norvig 1996). 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. Policies can be specified at different levels of abstraction. Often, they are in the form of laws or rules of thumb and can be described by a set of *event-condition-action* (ECA) rules such as:

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

Intuitively, a rule of the form (1) says that if *event* occurs in a situation in which *condition* is true then *action* must be executed. In other words, *event* triggers *action* to occur if *condition* is true.

For example, the law says that we have to yield the way for an approaching ambulance. This can be described by the ECA rule:

$$ambulance\_approaching \ \textbf{causes} \ \ yield\_way \qquad (2)$$

Another example and good policy to follow is to lock our office PC's screen whenever we go home. This "rule of thumb" can be expressed by the following rule:

$$leaving\_office \ \textbf{causes} \ \ lock\_screen \qquad (3)$$

*Work done while at Bell-Lab.

In (2) and (3), *ambulance_approaching* and *leaving_office* are events while *yield_way* and *lock_screen* are actions. While (2) represents a law, (3) does not. It only suggests a useful behavior.

There have been many proposals on how to specify policies and languages to implement them (see, for example, (Moffett & Sloman 1993) for policies in general, (Eiter, Subrahmanian, & Pick 1999; Eiter & Subrahmanian 1999) for policies in agent programming, and in network management (Wies 1994).) Our motivation is driven by the specification of policies for network management. The Internet Engineering Task Force (IETF) has proposed a standard language for the specification of network policies (IETF 1998), arguing for an object-oriented approach. Yet, none of the commercial network management tools implements it in full, perhaps because of the vagueness of the computational model behind the language.

In (Lobo, Bhatia, & Naqvi 1999), $\mathcal{PDL}$, a simple but expressive policy description language was introduced.[1] The distinguishing feature of $\mathcal{PDL}$ is its domain-independent and declarative semantics. In this language, a policy is a set of ECA rules which maps event histories into action histories.

A network management system based on $\mathcal{PDL}$ has been implemented to do Operations, Administration, Maintenance and Provisioning of telephone networks and it is described in (Virmani, Lobo, & Kohli 1999). The core of this system is a policy server that manages (one or more) policies by accepting registrations of policies (their events and rules), observing events happening in the system, and then issuing commands according to the policies. A similar policy server has been used to implement a simple workflow management system. In all of these applications, policies are carefully developed to avoid

1. issuing conflict commands;
2. issuing redundant commands; and

[1] Although there are no formal comparisons between $\mathcal{PDL}$ and the standards being proposed at IETF (the standard is work in progress, still incomplete), it is not difficult to check that policies written in the proposed IETF model can be captured in $\mathcal{PDL}$.

3. having policy rules that are never used.

Consider the following scenario.

Example 1 *An Internet Service Provider (ISP) uses a policy server to balance access to its servers. The goal is to connect users to servers according to their location and time of connection. The ISP uses the following rule*

$$\begin{array}{l} \mathit{request\_access}(id)\ \textbf{causes}\ \ \mathit{connect}(id, srv_1) \\ \quad \textbf{if}\ \ \mathit{location}(id, eastCoast), Time > 12{:}00 \end{array} \quad (4)$$

Here, id *represents a connection request containing the information about* user-id *and* machine-id *from which the user is trying to connect to the server.* location(id,eastCoast) *is a predicate that determines whether or not the user is located in the East coast. The rule can be read as: if a user requests a connection then connect her/him to the server 1 if s/he is located in the East coast and s/he is trying to connect after 12 PM.*

In addition to load balancing the policy server is also used by the ISP to handle access control. The policy that the ISP wishes to implement is to deny simultaneous connections of a single user. It is implemented by the following rule:

$$\begin{array}{l} \mathit{request\_access}(id) \\ \qquad \textbf{causes}\ \ \mathit{deny\_access}(id) \\ \qquad \textbf{if}\ \ \mathit{active\_connection}(id) \end{array} \quad (5)$$

This rule, while trying to achieve the goal of not allowing a user to have more than one connection, causes the policy server to issue conflicting commands if it is combined with the load balancing policy since the load balancing policy reacts independently of the access control policy.

The above conflict is simple for a policy developer to recognize. However, we can easily write policies where conflict detection becomes much harder to discover. This is even more problematic in $\mathcal{PDL}$, where policies could depend not only on events generated by the environment but also on events generated by the policy itself. For example, it is not immediately obvious whether the following policy can cause the policy server to issue conflict commands at the same time; and if so, when will it happen.

$$\begin{array}{l} seq(e_1(i_1), e_2(i_2), e_3(i_3), e_4(i_4), e_5(i_5)) \\ \qquad \textbf{causes}\ \ up(i_1 + i_2 + i_3 + i_4 + i_5) \\ \qquad \textbf{if}\ \ f(i_1, i_2, i_3, i_4, i_5) \\ seq(e_1(i_1), e_2(i_2), e_3(i_3))\ \textbf{triggers}\ \ e_4(i_1 + i_2 \times i_3) \\ seq(e_2(i_2), e_3(i_3), e_4(i_4))\ \textbf{triggers}\ \ e_5(i_2 + i_3 \times i_4) \\ e_1(i_1) \& e_2(i_2)\ \textbf{causes}\ \ down(i_1 \times i_2) \end{array} \quad (6)$$

In this policy, e_1, e_2, e_3 are external events with one integer parameter whose value is in the range from 1 to 9; e_4 and e_5 are two internal events which are generated by the system, $f(i_1, i_2, i_3, i_4, i_5)$ is a (predefined) Boolean function, and for every n, $up(n)$ and $down(n)$ are two conflicting actions. The first rule says that if the events e_1 through e_5 occur in sequence and the evaluation of f returns true then the action *up* is executed with the appropriate parameter. The second rule says that if the events e_1, e_2, e_3 occur in this order then event e_4 is generated by the policy (with the appropriate attributes). The next rule is similar. The last rule says that if the events e_1 and e_2 occur simultaneously the action *down* is executed. To find a situation in which the policy server would generate conflict actions, one needs to solve the following equation:

$$2(i_1 + i_2) + i_3 + i_2 i_3 + i_1 i_3 + i_2 i_3^2 = j_1 j_2 \quad (7)$$

and checking whether $f(i_1, i_2, i_3, i_1 + e_2 i_3, i_2 + i_3(i_1 + e_2 i_3)$ is true, where i_1, i_2, i_3, j_1 and j_2 are integers in the range from 1 to 9. It is not obvious to see if this can happen or not.

The above example suggests that an *automatic conflict detection tool can be useful in the developing process of policies* (i.e. finding out under which circumstances *down* and *up* are executed simultaneously).

We are interested in the question of finding an event history (i.e. sequence of events) that causes the policy server to issue a particular action history (i.e. sequence of actions). We call this *the generation problem.* Note that conflict detection (or redundancy detection) is an instance of the generation problem. For example, if a_1 and a_2 are two conflicting commands (or commands with the same effects) and there is an event history that makes the policy server to issue a_1 and a_2 simultaneously, then the rules containing a_1 and a_2 are in conflict (or redundant). Similarly, a rule is useless or redundant when there is no event history which makes the event-condition part of the rule true.

Our goal in this paper is to establish the foundations to represent and reason about $\mathcal{PDL}$ policies using logic programs with the stable model semantics (Gelfond & Lifschitz 1988). We use logic programming for the following reasons: (i) logic programming is a powerful knowledge representation language with well-understood declarative semantics; (ii) there are efficient tools for computing stable models of logic programs (Citrigno *et al.* 1997; Niemelä & Simons 1997); (iii) the generation problem can be naturally formulated as a stable model computation problem, i.e., there exists an effective reduction from the generation problem to the stable model computation problem.

In the first part of the paper we introduce $\mathcal{PDL}$ and the general semantic framework of the language in terms of logic programs. In the second part, we define the generation problem in reasoning about policies and show that in general, the problem is harder than the evaluation of policies by proving that for a simple class of policies, the problem is PSPACE-complete. We then show how the generation problem can be solved using logic programs under the stable model semantics. We also identify classes of policies for which polynomial solutions exist and develop algorithms to solve them. Discussion of future work can be found in the last section of the paper.

The Language $\mathcal{PDL}$

The design of $\mathcal{PDL}$ has been strongly influenced by the state language of actions of (Geffner & Bonet 1998), the different action description languages (Gelfond & Lifschitz 1993; Giunchiglia & Lifschitz 98), and the composite temporal event language of (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). It is worth noting that $\mathcal{PDL}$ is mainly concerned with specifying the reaction of a policy given an event history, while action languages are designed to reason about effects of actions and planning. This leads to an important difference between $\mathcal{PDL}$ and action languages: effects of actions are not specified in $\mathcal{PDL}$. This departure from action languages is based on the fact that policies are reactive systems used mostly to monitor environments where actions are usually not executed by the policy. The policy only requests to an external entity the execution of the action. In fact, *$\mathcal{PDL}$ can be described as a real-time specialized production rule system to define policies.*

However, like many action description languages, a policy in $\mathcal{PDL}$ defines a transition system between states and its computational model can be naturally captured using logic programs. Since our goal in this paper is not the language per se, we simplify the syntax of $\mathcal{PDL}$ and describe its semantics directly in terms of logic programs skipping an independent definition of the transition function. We translate a $\mathcal{PDL}$ policy description P into a logic program Π_P and extract the transition function from the stable models of P.[2]

We assume a sorted finite first order language $\mathcal{L}$ with finite sets of function and predicate symbols. Function symbols are partitioned into four sets: *internal event* symbols, *external event* symbols, *action* symbols and *standard* symbols. Standard symbols are of arity 0 and we call them constants. The other symbols may or may not be of arity zero.

The basic components of a policy description are events, actions, and rules. A *primitive event* is a term whose principal functor is an event symbol and its arguments are either variable or constant symbols. It will be an internal event if the functor is an internal event symbol; otherwise it will be an external event. An *action* is a term whose principal functor is an action symbol.

Intuitively, an external event is generated by the environment, it can occur at anytime and is independent of the policies controlling the system. On the other hand, internal events are those that are deliberately generated by a policy. Actions represent the response of the policies to changes in the environment. The effect of an action can result in the intervention of a system operator (human), or the execution of a program, etc.

An *event literal* is either a primitive event or a event symbol preceded by $\neg$. A ground primitive event of the form $e(t_1,\ldots,t_k)$ is called an *instance* of the event e. An *epoch* is a set of event instances representing the simultaneous occurrences of the events.[3] We say that a negative event literal $\neg e$ occurs in an epoch if there are no instances of the event e in the epoch. Given the event literals $e_1, e_2, \ldots, e_n$ $(1 \leq n)$, we denote the conjunction by $e_1 \& e_2 \& \ldots \& e_n$, the sequence by $seq(e_1, e_2, \ldots, e_n)$ and the relax-sequence by $[e_1, e_2, \ldots, e_n]$. These are called *complex* events. An instance of $e_1 \& e_2 \& \ldots \& e_n$ consists of an instance of each e_i in the same epoch. An instance of $seq(e_1, e_2, \ldots, e_n)$ in an event history $\mathcal{H}_E = (E_1, \ldots, E_m)$ (i.e. a sequence of epochs) consists of an instance e_n in the last epoch E_m of $\mathcal{H}_E$, plus an instance of e_{n-1} in E_{m-1}, …, plus an instance of e_1 in E_{m-n+1}. An instance of $[e_1, e_2, \ldots, e_n]$ in an event history $\mathcal{H}_E = (E_1, \ldots, E_m)$ consists of instances e_1 through e_n occurring in epochs E'_1 through E'_n respectively such that each E'_i occurs before E'_{i+1} in $\mathcal{H}_E$, there are no instances of e_{i+1} in any epoch between E'_i and E'_{i+1} in $\mathcal{H}_E$, and $E'_n = E_m$. A rule in $\mathcal{PDL}$ is a proposition of the form

$$event_1 \textbf{ causes } action \textbf{ if } condition \quad or \quad (8)$$

$$event_1 \textbf{ triggers } event_2 \textbf{ if } condition \quad (9)$$

where $event_1$ is a complex event, $event_2$ is an internal event whose arguments are constants or variables that appear in $event_1$, *action* is an action term whose arguments are also constants or variables that appear in $event_1$. The *condition* is an expression of the form $p_1, \ldots, p_n$, where each p_i is a predicate of the form $t_1 \theta t_2$, θ is a relation operator from the set $\{=, \neq, <, \leq, >, \geq\}$ and each t_i is either a constant or a variable that appears in $event_1$. Rules of the forms (8) and (9) are called action and event generation rules, respectively. A *policy description* is a finite set of rules of the form (8) and (9).

Let an E-*history* be a finite (and possibly empty) sequence of epochs and an A-*history* be a non-empty finite sequence of sets of ground action terms. In general, a *policy* P is a prefix-closed function that maps E-histories into A-histories. That is, for any history $\mathcal{H}_E$=$(E_1, \ldots, E_{n+1})$, if $P(\mathcal{H}_E)$=$(A_1, \ldots, A_m)$ then $P(E_1, \ldots, E_n)$ is a prefix of $(A_1, \ldots, A_m)$.

We describe the semantics of a policy description P using a locally stratified logic program Π_P of a special form. The minimal model of this program, which can be computed using well-known logic programming computational techniques, represents the function defined by the policy. To compute the actions that are triggered given an E-*history*, we transform the epochs 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. We will use

[2] We must note that the translation Π_P is different from the one proposed in (Lobo, Bhatia, & Naqvi 1999). For the class of policies defined in this paper, the two translations are equivalent though.

[3] This set is finite since the number of constants in the language is finite.

the term $seq(e_n, seq(e_{n-1}, \ldots, seq(e_1)\ldots)$ to represent $seq(e_1, \ldots, e_n)$. Likewise, the list $[e_n, \ldots, e_1]$ will be used to represent $[e_1, \ldots, e_n]$.

We introduce a new sort: *time*, represented by the set of positive integers,[4] and three new predicates: $occ(X,T)$, $h(X,T)$, and $r\_h(X, Rseq, T)$. Intuitively, each integer represents a time in the history, $occ(X,T)$ denotes that the primitive event (or action) X occurs in epoch number T while $h(X,T)$ denotes that the complex event X occurs in epoch number T, and $r\_h(X, Rseq, T)$ represents that X has occurred after T and that we need to check that the relax sequence $Rseq$ occurs at T.

An instance $e(t_1, \ldots, t_m)$ of an event e in an epoch i is represented by the ground atom $occ(e(t_1, \ldots, t_m), i)$. The set of rules that define the predicates h and r_h is given next.

$$\begin{array}{lcl} h(Ev,T) & \leftarrow & occ(Ev,T). \\ h(Ev_1\&Ev_2,T) & \leftarrow & h(Ev_1,T), \\ & & h(Ev_2,T). \\ h(seq(Ev),T) & \leftarrow & occ(Ev,T). \\ h(seq(Ev,Seq),T{+}1) & \leftarrow & occ(Ev,T{+}1), \\ & & h(Seq,T). \\ h([],T) & \leftarrow & \\ h([Ev],T) & \leftarrow & occ(Ev,T). \\ h([Ev|Rs],T{+}1) & \leftarrow & occ(Ev,T{+}1), \\ & & r\_h(Ev,Rs,T). \end{array} \tag{10}$$

And for each k-ary event symbol e we have the rules

$$\begin{array}{lcl} r\_h(e(\vec{X}),Rs,T{+}1) & \leftarrow & occ(\neg e,T{+}1), \\ & & r\_h(e(\vec{X}),Rs,T). \\ r\_h(e(\vec{X}),Rs,T) & \leftarrow & h(Rs,T). \\ r\_h(\neg e,Rs,T{+}1) & \leftarrow & occ(e(\vec{X}),T{+}1), \\ & & r\_h(\neg e,Rs,T). \\ r\_h(\neg e,Rs,T) & \leftarrow & h(Rs,T). \end{array} \tag{11}$$

In all of the rules in (11) and (10), Ev, Rs, or T is a variable of the type event[5], relax-sequence, or time, respectively. The intention is to define when a complex event is satisfied (or occurs). For that, the first rule is for a single event, and the next rule for a conjunction of event instances is satisfied. The third and fourth rules are for a sequence of events, and the last two rules, together with the rules in (10), are for a relax-sequence.

We're now ready to define the program Π_P associated with a policy description P. For each rule of the form

$$E \textbf{ causes } a \textbf{ if } t_1^1\theta_1 t_2^1, \ldots, t_1^m \theta_m t_2^m$$

in P, Π_P contains a rule of the form

$$occ(a,T) \leftarrow h(E,T), t_1^1\theta_1 t_2^1, \ldots, t_1^m\theta_m t_2^m. \tag{12}$$

For each rule of the form

$$E \textbf{ triggers } e \textbf{ if } t_1^1\phi_1 t_2^1, \ldots, t_1^m\phi_m t_2^m$$

in P, Π_P contains a rule of the form

$$occ(e,T+1) \leftarrow h(E,T), t_1^1\theta_1 t_2^1, \ldots, t_1^m\theta_m t_2^m. \tag{13}$$

The program Π_P consists of rules in (10)-(11), rules (12) and (13), and the rules representing the Closed-World-Assumption for k-ary event symbol e:

$$\begin{array}{l} occ(\neg e,T) \leftarrow \textit{not } event(e,T). \\ event(e,T) \leftarrow occ(e(X_1,\ldots,X_k),T). \end{array} \tag{14}$$

which says that at anytime, if no instance of the event e occurs then the event $\neg e$ occurs. Here *event* stands for *external* or *internal*.

Example 2 *Let us denote the policy description in Example 1 by P. P is translated into Π_P as follows. There is one event symbol:* request_access, *and two action symbols:* connect *and* deny_access. *The event has one attribute:* id. connect *has two parameters of type* id *and* server *while* deny_access *has only one parameter of type* id. *Besides (10)-(11) and (14), Π_P contains the following rules:*

$$\begin{array}{l} occ(connect(Id,srv_1),T) \leftarrow \\ \quad h(request\_access(Id),T), \\ \quad location(Id,eastCoast), \\ \quad Time > 12{:}00. \\ occ(deny\_access(Id),T) \leftarrow \\ \quad h(request\_access(Id),T), \\ \quad active\_connection(Id). \end{array}$$

Notice that if there are several occurrences of the same primitive event in a rule, their attributes must be renamed accordingly to avoid clashes. We demonstrate this problem in the next example.

Example 3 *Consider a policy rule*

$$e(I_1), e(I_2) \textbf{ triggers } k(I_1,I_2) \textbf{ if } I_1{>}I_2, I_1{\geq}100,$$

where e is an event with one attribute whose domain is the set of integers. This rule is translated into the following rule:

$$\begin{array}{lcl} occ(k(I_1,I_2),T{+}1) & \leftarrow & h(seq(e(I_1),e(I_2)),T), \\ & & I_1{>}I_2, I_1{+}I_2{\geq}100. \end{array}$$

For an event history $\mathcal{H}_E = (E_1, \ldots, E_m)$, let $fact(\mathcal{H}_E, j) = \{occ(e,i) \mid e \in E_i,\ 1 \leq i \leq j\}$. We say that $\mathcal{H}_E$ is *plausible* for a policy description P iff either $\mathcal{H}_E$ is empty or $\mathcal{H}_E = (E_1, \ldots, E_m)$ and for every internal event e, the following holds:

(*) **for every** i, $2 \leq i \leq m$,

$$e \in E_i \quad iff \quad \Pi_P \cup fact(\mathcal{H}_E, i-1) \models occ(e,i).$$

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

[4] We use integers for convenience but we only need to refer to terms of the form 0, $S(0)$, $S(S(0))$,..., etc. in the spirit of Datalog$_{1S}$ (Chomicki 1993).

[5] Strictly speaking we need two copies of each rule, one for internal events and the other one for external.

Definition 1 *Let P be a policy description and $\mathcal{H}_E = (E_1, \ldots, E_m)$ be a plausible event history. $P(\mathcal{H}_E) = (A_1, \ldots, A_m, \ldots)$ iff $A_i = \{a | \Pi_P \cup fact(\mathcal{H}_E, m) \models occ(a, i)\}$.*

$P(\mathcal{H}_E)$ is the action history that is triggered by the event history $\mathcal{H}_E$.

Reasoning About Policies

The semantics of policy descriptions specifies the actions (and internal events) that are generated along a "history" of external events. Computing the semantics of a policy description amounts to computing the sequence of actions and internal events that are triggered along an event history and is called *policy evaluation.* Algorithms for policy evaluation have been discussed in (Bhatia, Lobo, & Kohli 2000) and implemented in the system Netmon (Virmani, Lobo, & Kohli 1999). It follows from Definition 1 that policy evaluation can be done in logic programming by computing the models of the program $\Pi_P \cup fact(\mathcal{H}_E, m)$. As our goal in this paper is not to study the evaluation of policies we will not discuss this topic further. Work on policy evaluation can be found in (Lobo, Bhatia, & Naqvi 1999) and (Bhatia, Lobo, & Kohli 2000).

Reasoning about policies is concerned with problems such as *tracing* (determining an event history which triggers an action given a complete history of the system), *hypothetical reasoning* (finding or completing an event history that explains partial observations of the history of the system), *intra- and inter-policy conflict detection and resolution*, and *policy composition* (constructing policies from other policies). Tracing is perhaps the most simple problem in reasoning about policies. It is similar to the projection problem in reasoning about actions. In fact, tracing can be done in polynomial time (with respect to the length of the given event history and the length of the policy). We have developed algorithms and implemented them in a system called **ExSPS** (**Ex**planation **S**ystem for **P**olicy **S**erver), which is currently in use at Bell-Labs to explain the behavior of a system given a complete history. Results on intra- and inter-policy conflict detection and resolution can be found in (Chomicki, Lobo, & Naqvi 2000; Chomicki & Lobo 2001). *In this paper, we concentrate on hypothetical reasoning. Particularly, we are interested in the problem of finding an explanation (in terms of an event history) for a given action history.*

Notice that an event history that triggers an action history can contain internal events, which cannot arbitrarily occur. Thus in order to generate an action or an action history, we should only consider plausible event histories. In what follows, we define the size of a policy description P by a triple (n_1, n_2, n_3) where n_1, n_2, and n_3 are the number of rules, the number of external events, and the number of internal events respectively. Before introducing the problem, we need the following definition.

Definition 2 *An action history $\mathcal{H}_A = (A_1, \ldots, A_n)$ occurs in an action history $\mathcal{H}'_A = (A'_1, \ldots, A'_m)$, denoted by $\mathcal{H}_A \prec \mathcal{H}'_A$, if there exists an integer k, $0 \leq k \leq m-n$, such that $A_j \subseteq A'_{j+k}$ for $1 \leq j \leq n$.*

In general, we are interested in the following problem.

Definition 3 (Generation) *Given a policy description P and an action history $\mathcal{H}_A$. The* generation problem *of $\mathcal{H}_A$, denoted by $G(P, \mathcal{H}_A)$, is the problem of finding a plausible history $\mathcal{H}_E$ such that $\mathcal{H}_A \prec P(\mathcal{H}_E)$. The size of the problem is defined by the size of the policy description P and the size of $\mathcal{H}_A$. $\mathcal{H}_E$ is called a* solution *of $G(P, \mathcal{H}_A)$. If $\mathcal{H}_A$ is a single action, we denote the problem by $G_1(P, \mathcal{H}_A)$ and called it the* single action generation problem.

We have shown that the generation problem is at least PSPACE-hard by proving that the following simplified version of the problem is PSPACE-complete.

Definition 4 *A policy description P is said to be* simple *if its underlying language $\mathcal{L}$ is propositional (i.e. event and action symbols are of arity zero) and in each of its rules, the complex event is a conjunction of event literals and the condition is empty. The generation problem for simple policy descriptions is denoted by $G^s(P, \mathcal{H}_A)$ and the single action generation problem is denoted by $G^s_1(P, \mathcal{H}_A)$.*

Theorem 1 *$G^s_1(P, \mathcal{H}_A)$ is PSPACE-complete.*

The hardness is proved by simulating the divergence of a deterministic Turing machine problem in $\mathcal{PDL}$.

LPs and the generation problem

We will solve $G(P, \mathcal{H}_A)$ by finding a logic program whose stable models correspond to solutions of $G(P, \mathcal{H}_A)$. The first step is to translate P into Π_P. This has been done in the previous section, i.e., Π_P consists of rules (10)-(11), (12), (13), and (14). To complete the translation, we will define a predicate $sat(\mathcal{H}_A, T)$ as follows.

$$\begin{array}{l} sat(a, T) \leftarrow occ(a, T). \\ sat(a_1 \& \ldots \& a_n, T) \leftarrow occ(a_1, T), \ldots, occ(a_n, T). \\ sat([A_1, \ldots, A_n], T) \leftarrow n > 1, sat(A_1, T), \\ \qquad\qquad sat([A_2, \ldots, A_n], T+1). \\ sat([A_1, \ldots, A_n], T) \leftarrow n = 1, sat(A_1, T). \end{array} \tag{15}$$

where a stands for an action, $a_1 \& \ldots \& a_n$ is a conjunction of action literals, and $[A_1, \ldots, A_n]$ represents an action history. Intuitively, these rules specify when an action history $\mathcal{H}_A$ occurs in the action history defined by the program.

Unlike the evaluation problem, the generation problem searches for an event history. Thus, there are no facts in the program. Rather, we need a way to generate external events. Following (Lifschitz 1999), we add to Π_P the following rules

$$\begin{array}{l} pick(Ev, T) \leftarrow external(Ev), not\ no\_pick(Ev, T). \\ no\_pick(Ev, T) \leftarrow external(Ev), not\ pick(Ev, T). \\ occ(Ev, T) \leftarrow pick(Ev, T). \end{array} \tag{16}$$

These rules specify that at any moment of time, any instance of the external event Ev can be picked to appear in the epoch T. We have that

Theorem 2 *For a finite action history $\mathcal{H}_A$,*

- *for each solution $\mathcal{H}_E$ of length n to the problem $G(P, \mathcal{H}_A)$ there exists an integer T such that $\Pi_P \setminus R \cup fact(\mathcal{H}_E, n) \models sat(\mathcal{H}_A, T)$ where R is the set of rules in (12); and*
- *for each T such that $\Pi_P \cup sat(\mathcal{H}_A, T)$ has a stable model there exists a solution $\mathcal{H}_E$ to the problem $G(P, \mathcal{H}_A)$ such that $\mathcal{H}_A \prec P(\mathcal{H}_E)$.*[6]

When is $G_1^s(P, \mathcal{H}_A)$ polynomial solvable?

It follows from Theorem 1 that there is no efficient algorithm to solve the problem $G_1^s(P, \mathcal{H}_A)$ in general. We will show next that for a restricted class of instances, the problem can be solved in polynomial time.

In this section, by P we denote an arbitrary simple policy description whose sets of external and internal events are X and Y respectively. We will discuss the problem $G_1^s(P, \mathcal{H}_A)$ with respect to the class of instances in which X and Y are fixed, denoted by $G_1^s(P, X, Y, \mathcal{H}_A)$. The size of the description is now n_1 - the number of rules in P.

We now describe a back-tracking algorithm of complexity $O(n_1^{n_3})$ for finding a solution to the problem $G_1^s(P, X, Y, \mathcal{H}_A)$. It is based on the following observations: if $G_1^s(P, X, Y, \mathcal{H}_A)$ has a solution then (i) it has a solution $\mathcal{H}_E = (E_1, \ldots, E_n)$ with $n \leq 2^{n_3}$ and $int(E_i) \neq int(E_j)$, for $1 \leq i \neq j \leq n$, where $int(E)$ denotes the set of internal events in epoch E and (ii) E_n is an epoch satisfying a rule generating $\mathcal{H}_A$.[7]

For an epoch S, let $pde(S)$ be the set of internal events which are triggered by S, i.e., for each $e \in S$ there exists a rule "E **triggers** e" in P such that E is satisfied by S. For an action $\mathcal{H}_A$, $g(P, \mathcal{H}_A)$ is the set of epochs which satisfies a rule generating $\mathcal{H}_A$ and for an event history $\mathcal{H}_E = (E_1, \ldots, E_n)$, $\mathcal{E}(P, \mathcal{H}_E)$ is the set of epochs E such that $int(E) \neq int(E_j)$ $(1 \leq j \leq n)$ and $pde(E) = int(E_1)$. The algorithm relies on the computations of $g(P, \mathcal{H}_A)$ and $\mathcal{E}(P, \mathcal{H}_E)$. The pseudo-code for the two algorithms used in solving $G_1^s(P, X, Y, \mathcal{H}_A)$, $gen(P, \mathcal{H}_A)$ and $genHistory(P, \mathcal{H}_E)$, are given in the next table. $gen(P, \mathcal{H}_A)$ receives a simple policy description and an action $\mathcal{H}_A$ as input, and returns a solution to $G_1^s(P, X, Y, \mathcal{H}_A)$ (or *nil* if no solution exists.) $genHistory(P, \mathcal{H}_E)$ receives a simple policy description and an event history $\mathcal{H}_E$ as input, and returns a plausible history which has $\mathcal{H}_E$ as its suffix. In the table, **EH** stands for "event history". The next theorem shows that $G_1^s(P, X, Y, \mathcal{H}_A)$ can be solved in polynomial time.

```
EH gen(P, H_A)
compute g(P, H_A)
for each E ∈ g(P, H_A)
 set Ans := (E)
 compute H_E = genHistory(P, Ans)
 if H_E ≠ nil then return H_E
endfor
return nil
```

```
EH genHistory(P, H_E)
compute E(P, H_E)
if E(P, H_E) = ∅ return nil
for each E ∈ E(P, H_E),
set H'_E:=(E, H_E)
 if H'_E is plausible then return H'_E
 else compute H''_E=genHistory(P, H'_E)
   if H''_E ≠ nil then return H''_E
endfor
return nil
```

Theorem 3 *The complexity of $gen(P, \mathcal{H}_A)$ is $O(n_1^{n_3})$.*

Corollary 1 *$G_1^s(P, X, Y, \mathcal{H}_A)$ is polynomial solvable.*

Discussion and Conclusions

We presented a logic programming-based approach to represent and reason about policies by translating policies written in $\mathcal{PDL}$ into logic programs. We used the logic programming representation to solve the generation problem. We proved that finding an event history that triggers an action history is generally intractable. To cope with the high complexity of the generation problem, we identified a class of policy descriptions in which the problem can be solved in polynomial time.

We used SMODELS in our preliminary experiments and the results are very encouraging. Due to the portability requirements of our applications we have translated part of the logic program into a JAVA program. The program receives a simple policy description P and an action history $\mathcal{H}_A$ as input and returns as output an event history $\mathcal{H}_E$ generating $\mathcal{H}_A$ (or *nil*, if such an event history does not exist). In the future, we plan to integrate this program with the system **ExSPS** to create a tool for policies developing and testing. We are planning to use the logic programming representation described in this paper to investigate other problems of hypothetical reasoning. We are interested in problems such as *completion of a partial event history* (that generates a given action history); *generation of action histories with constraints* (finding an event history that generates an action history that satisfies some given constraints, for example, an action history that contains n occurrences of action a and no occurrences of action b; or an action history that contains no parallel occurrences of a and b; etc.); or *policy composition*. While completion is important for explaining observed behaviors, generation of action histories with constraints and

[6] To compute the stable models of Π_P using SMODELS we need to type all the rules of Π_P with predicates of the form $time(T)$, $external(E)$ etc. wherever it is needed. For example, the rule "$occ(Ev, T) \leftarrow pick(Ev, T)$" will be written as "$occ(Ev, T) \leftarrow external(Ev), time(T), pick(Ev, T)$."

[7] E is satisfied by S if for each positive literal $e \in E$, $e \in S$, and for each negative literal $\neg e \in E$, there is no instance of e in S.

policy composition are good for the policy development process. They could provide the necessary guidance for avoiding undesirable behaviors. For example, to avoid the parallel occurrence of two actions a and b is equivalent to make sure that no event history triggers a and b at the same time. Thus, the generation of the action history $a \wedge b$ must not have a solution. On the other hand, policy composition can help reducing the time in writing policies. Our hypothesis is that these problems are still at least PSPACE-hard but for certain classes of policies, they can be solved effectively.

Acknowledgment

We wish to thank Jan Chomicki for his insightful guidance in proving Theorem 1. The first author is grateful to Lucent Technologies for the support during his stay at Bell Labs.

References

Bhatia, R.; Lobo, J.; and Kohli, M. 2000. Policy evaluation for network management. In *INFOCOM 2000*.

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

Chomicki, J.; Lobo, J.; and Naqvi, S. 2000. A logic programming approach to conflict resolution in policy management. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, 121–133.

Chomicki, J.; and Lobo, J. 2001. Monitors for History-based Policies. In M. Sloman, J. Lobo, E. Lupu eds., *Proceedings of POLICY 2001: Policies for Distributed Systems and Networks*, LNCS No. 1995, 57–72.

Chomicki, J. 1993. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming* 12:1–31.

Citrigno, S.; Eiter, T.; Faber, W.; Gottlob, G.; Koch, C.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. The dlv system: Model generator and application frontends. In *Proceedings of the 12th Workshop on Logic Programming*, 128–137.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of European Workshop on Planning*.

Eiter, T., and Subrahmanian, V. S. 1999. Heterogeneous active agents, ii: Algorithms and complexity. *Artificial Intelligence* 108(1,2):257–307.

Eiter, T.; Subrahmanian, V. S.; and Pick, G. 1999. Heterogeneous active agents, i: Semantics. *Artificial Intelligence* 108(1,2):179–255.

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

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Logic Programming: Proc. of the Fifth International Conf. and Symp.*, 1070–1080.

Gelfond, M., and Lifschitz, V. 1993. Representing actions and change by logic programs. *Journal of Logic Programming* 17(2,3,4):301–323.

Giunchiglia, E., and Lifschitz, V. 98. An action language based on causal explanation: preliminary report. In *Proc. of AAAI 98*, 623–630.

IETF. 1998. Policy framework definition language.

Lifschitz, V. 1999. Answer set planning. In *Proceedings of ICLP*, 23–37.

Lobo, J.; Bhatia, R.; and Naqvi, S. 1999. A policy description language. In *Proceedings of the 16th National Conference on Artificial Intelligence*, 291–298.

Moffett, J. D., and Sloman, M. S. 1993. Policy hierarchies for distributed system management. *IEEE JSAC Special Issue on Network-Management* 11(9).

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

Niemelä, I., and Simons, P. 1997. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. ICLP & LPNMR*, 420–429.

Russell, S., and Norvig, P. 1996. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 1 edition.

Virmani, A.; Lobo, J.; and Kohli, M. 1999. NETMON, network management for the saras softswitch. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications/International Conference on Artificial Intelligence*.

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.