User Preferences VS Minimality in PPDL*

Elisa Bertino[1], Alessandra Mileo[1] and Alessandro Provetti[2]

[1] Dipartimento d'Informatica e Comunicazione
Università degli studi di Milano. Milan, I-20135 Italy
{bertino, mileo}@dico.unimi.it.
[2] Dipartimento di Fisica
Università degli studi di Messina. Messina, I-98166 Italy
ale@unime.it

Abstract In the context of Network management, Chomicki et al. defined the specification language PDL (Policy Description Language) and later extended it by introducing monitors: constraints on the actions that the network manager cannot execute simultaneously. We have furtherly extended PDL by permitting specifying user preferences on how to enforce constraints; that extension is called PPDL and it is based on Brewka's ordered disjunction connective. In this article we speculate on how the minimality requirement, stating that constraints on actions should affect action execution as little as possible, can be specified and implemented in PPDL theories. Minimal interference and maximal satisfaction of user preferences are not always achievable and tend to interact in complex ways.

1 Introduction

Chomicki, Lobo, Naqvi, and others have addressed network services management by defining a high-level specification language, called Policy Description Language (PDL). In that context, a policy is a description of how events received over a network (e.g., queries to data, connection requests etc.) are served by some given network terminal, often identified as *data server.* PDL also allows the specification of *monitors*: descriptions of sets of actions that cannot be executed simultaneously to prevent illegal, hazardous or physically impossible situations. PDL allows managers to specify policies and monitors independently from the details of the particular device executing it. We refer the reader to works by [Chomicki et al., 2000,Chomicki et al., 2003] for a complete introduction and motivation for PDL in network management.

Our long-term research develop (and deploy) PDL to take advantage of most recent Knowledge representation concepts and results. We investigate PDL *in the*

* This work has been supported by i) MIUR COFIN project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps.* and ii) the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP and IST-2001-33058 PANDA projects.

abstract, i.e., as a specification language and investigates the requirements and the complexity of evaluating and executing PDL policies that include monitors (sometimes referred to as *consistency monitors*). The framework we use for our investigation of PDL is Answer Set Programming paradigm (ASP). ASP is a form of Logic Programming based on Gelfond-Lifschitz stable models semantics [Gelfond and Lifschitz, 1991].

ASP is a suitable language for expressing complicated or under-defined problems in a very concise form. Nowadays, there are rather efficient solvers [Systems] that can compute the answer sets of programs defining thousands of atoms within few seconds. In particular, the solver DLV computes answer sets of programs with disjunctions; such programs have been used by [Chomicki et al., 2003] for implementing monitors. Also, [Brewka, 2002] introduced logic programs with ordered disjunctions, which will be described later, that can be executed by an extension of the Smodels solver (called PSMODELS) described in [Brewka et al., 2002].

In [Bertino et al., 2003b] we took a different perspective, i.e., we *chose* the output based on degrees of satisfaction of a preference rule, as shown in [Brewka, 2002]. In the past we have worked on PDL in several directions. For example we have extended it to allow users to express preferences on how to enforce policies, reconstructing Brewka's ordered disjunction connective [Brewka, 2002] into PDL. The resulting language is called *PPDL: PDL with Preferences.* PPDL monitors are computed by translating them into Brewka's LPOD programs, computing the relative answer sets and then taking the maxima w.r.t. a preference relation over answer sets themselves. Ordered disjunctions are a relatively recent development in reasoning about preferences with Logic Programming and are subject of current work by [Brewka, 2002,Brewka et al., 2002,Brewka, Benferhat, Le Berre, 2002], [Buccafurri et al., 1998], [Schaub, Wang, 2001] and others. One important aspect of Brewka's work is that preferred answer sets need not be minimal. This is a sharp departure from traditional ASP.

Adding preferences to PDL implies an assessment of the trade-off between user-preferences and minimality of the solutions. The interaction between user-preference and minimality is illustrated in the following example.

Example 1. Suppose we want to control the operations of a call center, or a customer service line. An appropriate policy should describe which calls have to be preferencially accepted according to geographical location, time of the call, time waiting etc. We build our policy specifying preferences according to that strategy. Surely we want the control system to respect our preferences, but if two calls are in conflict, we also want to minimize cancelled calls, in order to serve a maximal (resp. a maximum) number of calls within a few minutes. Therefore, the policy application should return a set of accepted calls which is both preferred w.r.t. the policy and maximal (resp. maximum).

In this article we would like to discuss the more abstract issue of the trade off between capturing user preferences and a general criterion of minimal intervention. In fact, since each action is intended as satisfying a user's request, to maximize user satisfaction we would like monitor to cancel out (prevent from being executed) only a *minimal* (preferably *minimum*) number of actions.

This article is organized as follows. After giving an overview of PDL and of its implementation in Answer Set Programming in Section 2, in Section 3 an overview of LPODs is presented , and in section 4 we show how to introduce preferences in PDL monitors. In Section 5, we present a transformation that ensures minimality of the computed monitors (actions to be blocked). Finally, in Section 6 we will briefly describe the lines of development of our research on PPDL.

2 Introduction to PDL

PDL can be described as an evolution of the typical Event-Condition-Action (ECA) schema of active databases. In fact, a PDL policy is defined as a set of rules of the form

$$e_1, \ldots e_m \textbf{ causes } a \textbf{ if } C. \tag{1}$$

where C is a Boolean condition and $e_1, \ldots e_m$ are events , which can be seen as input requests[1]. Finally, a is an action, which is understood to be a configuration command that can be executed by the network manager.

PDL assumes events and actions to be syntactically disjoint and rules to be evaluated and applied in parallel. One may notice the lack of any explicit reference to time. In fact, PDL rules are interpreted in a discrete-time framework as follows. If at a given time t the condition is evaluated true and all the events are received from the network, then at time $t + 1$ action a is executed. As a result, we can see PDL policies as describing a transducer.

[Chomicki et al., 2003] gives a precise formal semantics of PDL policies by showing a translation into function-free [disjunctive]logic programs, which have Gelfond-Lifschitz Answer Set semantics [Gelfond and Lifschitz, 1991]. So the semantics of a policy written in PDL (and its extensions) is given in terms of the Answer sets of the translated policy. This article adopts the same methodology and discusses semantics always in terms of Answer Sets.

2.1 Consistency Monitors

Chomicki and his co-authors have extended the syntax of PDL to allow describing constraints on sets of actions executing together. This is the syntax of the new rules:

$$\textbf{never } a_1 \ldots a_n \textbf{ if } C. \tag{2}$$

where C is a Boolean condition[2] and $a_1 \ldots a_n$ are actions, prohibits the *simultaneous* execution of $a_1 \ldots a_n$. A set of such rules is called a consistency

[1] Also, non-occurrence of an event may be in the premise of the rule. To allow for that, for each event e a dual event $\overline{e}$ is introduced, representing the fact that e has not been recorded.

[2] We will not consider here the boolean condition C as it is not determinant for our argumentation.

monitor. Consistency monitors are instrumental to specify hazardous, insecure or physically impossible situations that should be avoided. At this point, clearly, the question becomes what to do when applying the policy yields a set of actions that actually violates one of the rules in the monitor. Applying a monitor means filtering actions that follow from a policy application, *canceling* some of them.

There is no universally agreed procedure performing cancellation. Chomicki et al. propose two alternatives, which give to monitors an operational semantics. The first semantics, called Action-monitor, consists in dropping some of the actions to be executed so that to *respect* the monitor.

The second semantics, called Event-monitor, consists in dropping some of the events from the input, then re-applying the original policy, which, having lesser conditions violated, this time will yield a set of to-be-executed actions that does not satisfy any of the constraints in the monitor.

This article deals with Action Cancellation only. In both cases, however, we notice that choice of which action to drop is non-deterministic.

2.2 Encoding PDL in Answer Set Programming

Let us describe the formal setting used in the rest of the article. First of all, following Chomicki et al., we slightly simplify the notation by restricting to policies and monitors with empty condition C. Second, we recall that the event ($\mathcal{E}$) and action ($\mathcal{A}$) alphabets are assumed to be disjoint. Also, $\mathcal{E}$ is implicitly extended by introducing for each event e its opposite $\overline{e}$. The intended meaning of $\overline{e}$: e did not happen, is captured by the next definition. A set E of events observed on the network is *completed* by $E^c = E \cup \{\overline{e_i} \ : \ e_i \notin E\}$.

The next set of definitions concerns the encoding of PDL policies as Answer Set Programs, basically following those of Chomicki et al.. The set of observed events, E, is represented by a set $occ(E)$ of *occurs* facts. Policies are encoded as follows.

Definition 1. *A Policy P over $\mathcal{E}$ and $\mathcal{A}$ ($\mathcal{E} \cap \mathcal{A} = \emptyset$) is a set of rules as in equation (1) that can be translated in ASP as follows:*

$$\begin{aligned} exec(a) \ :- \ & occ(e_1), \ \ldots, \ occ(e_l), \\ & not\ occ(e_{l+1}), \ \ldots, \ not\ occ(e_m). \end{aligned} \tag{3}$$

where $e_i \in \mathcal{E}, i = 1 \ldots n$ and $a \in \mathcal{A}$; for simplicity we will ignore the boolean condition C mentioned in equation (1).

Clearly, a set of *occurs* facts together with rules describing *exec* makes up a very simple ASP program, for which there is a unique answer set.

Definition 2. *Let $occ(E)$ be a set of occurs facts and P a policy. The consequence set of E w.r.t. a policy P, denoted $P(E)$, is the set of all actions implied by program $occ(E) \cup P$ such that*

$a \in P(E) \leftrightarrow occ(E) \cup P \models_{asp} exec(a)$.

and such set is unique[3]. Let us now introduce consistency monitors as ASP programs.

2.3 Encoding Action-Cancellation Monitors in ASP

Chomicki et al. define Action-Cancellation monitors in ASP as a set of rules as follows:
each constraint of the form "**never** a_1,...,a_n" is captured as a *conflict rule*

$$block(a_1) \ \vee \ \ldots \ \vee \ block(a_n) \ :- \ exec(a_1), \ldots, \ exec(a_n). \tag{4}$$

and for each action a occurring in a policy rule, there is an *accepting* rule:

$$accept(a) \ :- \ exec(a), \ not \ block(a). \tag{5}$$

respectively[4]. Given a PDL policy P and a set of input events E, we define the ASP program $\pi_P(E)$as containing:

1. $occ(E)$, i.e., a set of *occurs* facts representing E;
2. a set of rules encoding the pure policy, for which the encoding schema is given in formula 1, and
3. a set of rules encoding the consistency monitor, as mentioned above.

The Answer sets of $\pi_P(E)$ can be obtained by feeding it to the ASP solver DLV. The key point is that the resulting answer sets contains *accept* atoms that represent a set of actions compatible w.r.t. monitor application. ¿From the point of view of consistency monitoring, all answers are equivalent. Hence, we may reduce to computing just one. Extra preferences/constraints on the solution can be added by adding constraints to $\pi_P(E)$. This is what we will do in the next Section.

3 Logic Programs with Ordered Disjunctions

LPODs have been introduced by [Brewka, 2002] in his work on combining Qualitative Choice Logic and Answer set programming. A new connective called ordered disjunction, denoted with "×," is introduced. An LPOD consists of rules of the form

$$C_1 \ \times \ \ldots \ \times \ C_n \ :- \ A_1, \ \ldots, \ A_m, \ not \ B_1 \ \ldots, \ not \ B_k. \tag{6}$$

where the C_i, A_j and B_l are ground literals. The intuitive reading [Brewka, 2002] of the rule head is:

[3] The ASP declarative semantics, with the definition of $\models_{asp}$, can be found in [Gelfond and Lifschitz, 1991]

[4] Remember that rules defining *exec* showed in equation (1) are still part of the encoding.

if possible C_1*, but if* C_1 *is not possible, then* C_2*,*
...
if all of $C_1, \ldots, C_{n-1}$ *are not possible, then* C_n*.*

The $\times$ connective is allowed to appear in the head of rules only, and is used to define a preference relation to *select* some of the answer sets of a program by using ranking of literals in the head of rules, on the basis of a given strategy or a context. The answer sets of a LPODs program are defined by Brewka as sets of atoms that maximizes a preference relation induced by the "$\times$-rules" of the program.

LPOD programs can be interpreted using a special version of the solver *Smodels,* called *Psmodels,* which is presented in [Brewka et al., 2002]. Basically, LPOD programs are translated into equivalent (but longer) ASP programs and then send to *smodels*. To do so, we need to send the source LPOD program to (a recent version of) *Lparse,* the front-end to *Smodels,* with the *–priorities* option. The code showed in this article is intended for such type of interpretation.

3.1 Overview of LPOD semantics

The semantics of LPOD programs is given in terms of a model preference criterion over answer sets. [Brewka, 2002] shows how Inoue and Sakama's split program technique can be used to generate programs whose answer sets characterize the LPOD preference models. In short, a LPOD program is rewritten into several split programs, where only one head appears in the conclusion. Split programs are created by iterating the substitution of each LPOD rule 6 with a rule of the form:

$$C_i \text{ :-} A_1, \ldots, A_m, \mathit{not}\ B_1, \ldots, \mathit{not}\ B_k, \mathit{not}\ C_1, \ldots, \mathit{not}\ C_{i-1} \tag{7}$$

Consequently, Brewka defines answer sets for the LPOD program Π as the answer sets of any of the split programs generated from Π.

There is one very important difference between Gelfond and Lifschitz's answer sets and LPOD semantics: in the latter (set theoretic) minimality of models is not always wanted, and therefore not guaranteed. This can be better explained by the following example.

Example 2. Consider these two facts:

1. $A \times B \times C$.
2. $B \times D$.

To best satisfying both ordered disjunctions, we would expect $\{A, B\}$ to be the single preferred answer set of this LPOD, even if this is not even an answer set of the corresponding disjunctive logic program (where "$\times$" is replaced by "$\vee$") according to the semantics of [Gelfond and Lifschitz, 1991] as B is sufficient to satisfy both disjunctions and is *minimal*.

The example above shows that the built in minimality precludes preferred answer sets to be considered. Hence it is necessary to use non-minimal semantics. In the following, we will discuss the minimality of solution issue in depth. Please refer to [Brewka, 2002] and the survey in [Schaub, Wang, 2001] for further details about LPODs semantics and formalizations of preference criteria.

4 PPDL: PDL with Preference monitors

To describe a preference relation on action to be blocked when a constraint violation occur, we extend PDL allowing a new[5] kind of constraint of the form:

$$\textbf{never } a_1 \times \ldots \times a_n \textbf{ if } C. \tag{8}$$

which means that actions $a_1, \ldots, a_n$ cannot be executed together, *and* in case of constraint violation, a_1 should be preferably blocked; if that is not possible (i.e. a_1 must be performed), then block a_2 , then a_3, ...; if all of $a_1, \ldots, a_{n-1}$ must be performed, then block a_n.

Starting from equation (8), we can now show how to encode *PPDL policies* with preference cancellation rules into ASP programs.

Remember that, for pure Action-Cancellation monitor, [Chomicki et al., 2000] propose an encoding where for each constraint of the form "**never** a_1,...,a_n" we put in $\pi_P(E)$ a blocking rule as in equation (4), and for each action a_i we put in $\pi_P(E)$ an accepting rule as in equation (5).

Each new constraint defined in formula (8) is translated in LPOD as an *ordered blocking rule* of the form:

$$block(a_1) \times \ldots \times block(a_n) :\!- exec(a_1), \ldots, exec(a_n), C. \tag{9}$$

Fact 1 *Since the PPDL-to-LPOD translation described above is not provided with a mechanism for avoiding action block, the resulting program is deterministic: using LPOD semantics we will obtain answer sets where the leftmost action of each rules of the form (9) that fires is always dropped.*

Indeed, ordered disjunctions are worth having when some actions *may not be blocked*. This is the subject of next Section.

4.1 Anti-blocking rules

We now extend PPDL further by allowing users to describe actions that *cannot be filtered* under certain conditions. To do so, let us introduce the following *anti-blocking rule*:

$$\textbf{keep } a \textbf{ if } C. \tag{10}$$

[5] The standard preference-less constraints are still part of the language

where a is an action that cannot be dropped when the boolean condition C is satisfied. This rule is applied whenever a constraint of the form (8) is violated, and a is one of the conflicting actions. In ASP, anti-blocking rules are mapped in a constraint formulated as follows:

$$:- \ block(a), \ C. \tag{11}$$

which is intended as *action a cannot be blocked if condition C holds*. Notice that if if we want to control the execution of action a, postulating that under condition C action a is executed *regardless*, then we should write, in PPDL:

$$\emptyset \ \textbf{causes} \ a \ \textbf{if} \ C.$$
$$\textbf{keep} \ a \ \textbf{if} \ C.$$

that will be translated in LPOD as follows:

$$exec(a) \ :- \ C.$$
$$:- \ block(a), \ C.$$

Unlike in traditional PDL, where actions are strictly the consequence of events, by the **causes** described above we allows *self-triggered* or *internal* actions. We should mention that, even without internal events, a PPDL policy with monitor, blocking and anti-blocking rules, may be inconsistent. Consider the following example.

Example 3. Take policy P_{diet}:

$$\begin{array}{rcl} P_{diet} & = & \{ \ hungry \ \textbf{causes} \ eat\_meat. \\ & & \ \ hungry \ \textbf{causes} \ eat\_cake. \ \} \end{array}$$

and a preference monitor M_{diet} saying that if we are on diet, we cannot eat both meat and cake in the same meal, even if he/she is hungry. In particular, it is preferable to give up meat; if this is not possible (because we are anemyc), then we will give up cake.

$$\begin{array}{rcl} M_{diet} & = & \{ \ \textbf{never} \ eat\_meat \ \times \ eat\_cake. \\ & & \ \ \textbf{keep} \ eat\_meat \ \textbf{if} \ anemic. \\ & & \ \ \textbf{keep} \ eat\_cake \ \textbf{if} \ greedy. \ \} \end{array}$$

where *anemic* and *greedy* stand for Boolean conditions. Both P_{diet} and M_{diet} are translated the following LPOD, named π_{diet}:

$$\begin{array}{l} exec(eat\_meat) \ :- \ occ(hungry). \\ exec(eat\_cake) \ :- \ occ(hungry). \\ block(eat\_meat) \times \ block(eat\_cake) :- \ exec(eat\_meat), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad exec(eat\_cake). \\ :- \ block(eat\_meat), anemic. \\ :- \ block(eat\_cake), greedy. \end{array}$$

Now, suppose that event *hungry* has occurred. If we are from Naples, and anemic and greedy, π_{diet} is inconsistent.

The simple example above shows that if we want to use prioritized semantics in extended PPDL, we have to be careful in introducing anti-blocking rules, in order to ensure that at least one action can be blocked in any case when a constraint is violated.

5 Minimal Preferential Monitors

There are contexts where minimality of the solution is strictly required. This requirement can be described in PPDL by adding *anti-blocking rules*. To yield minimality, a monitor should be defined as in the following informal (and rather ad hoc) example.

$$\begin{array}{l} \textbf{\textit{never}}\ a_1\ \times\ a_2\ \textbf{\textit{if}}\ c_1. \\ \textbf{\textit{never}}\ a_2\ \times\ a_3\ \textbf{\textit{if}}\ c_2. \\ \textbf{\textit{keep}}\ a_1\ \textbf{\textit{if}}\ c_1,\ c_2. \end{array} \tag{12}$$

The translation of such monitor into a logic program yields the following unique answer set:

$$M\ \equiv\ \{block(a_2), accept(a_1), accept(a_3)\}$$

as expected. However, specifying all the needed ***keep*** axioms in general cases is cumbersome and inevitably going to slow down computation. More research is needed to find out the range of applicability of such technique.

We can present here some general consideration about preferences and minimality applied on *block* atoms of a preferential monitors. In formula (12) we wanted to obtain a minimal model containing only action a_2, and we got this by adding an ad-hoc *keep* clause. This situation can be generalized.

Definition 3. *Whenever we want to obtain a* minimal and preferred model*, that is* blocking a minimal number of actions by respecting user's preferences as much as we can*, we follow the procedure defined below:*

1. *generate all* ordered blocking rules *from the* never *constraints of the form as in the monitor (8);*
2. *add, for each* ordered blocking rule *(9), a constraint saying that we can block just one of the "*$\times - blocked$*" actions:*

$$:\!-\ block(a_1),\ \ldots,\ block(a_n),\ exec(a_1),\ \ldots,\ exec(a_n),\ C. \tag{13}$$

3. *call the* PSMODELS *solver applied to this new logic program. This will return models that are minimal* and *preferred.*

As a result, the first and the second rule in (12) will be translated into the following rules:

$$\begin{array}{l} block(a_1) \ \times \ block(a_2) \ if \ c_1. \\ block(a_2) \ \times \ block(a_3) \ if \ c_2. \end{array}$$

$$\begin{array}{l} :\!- \ block(a_1), \ block(a_2). \\ :\!- \ block(a_2), \ block(a_3). \end{array}$$

and the models (minimal and preferred) obtained by calling PSMODELS are:

$$\begin{array}{rcl} M_1 & \equiv & \{block(a_1), \ block(a_3), \ accept(a_2), \ \ldots\}. \\ M_2 & \equiv & \{block(a_2), \ accept(a_1), \ accept(a_3), \ \ldots\}. \end{array} \tag{14}$$

Unfortunately, there are still cases where this procedure does not return any result as showed in the example below.

Example 4. Consider the monitor containing the following constraints:

$$\begin{array}{l} \textbf{never} \ a \ \times \ b \ \times \ c. \\ \textbf{never} \ c \ \times \ d. \\ \textbf{never} \ d \ \times \ b \ \times \ a. \end{array}$$

Applying our procedure to get *minimal and preferred* monitors, we obtain:

$$\begin{array}{l} block(a) \ \times \ block(b) \ \times \ block(c). \\ block(c) \ \times \ block(d). \\ block(d) \ \times \ block(b) \ \times \ block(a). \end{array}$$

$$\begin{array}{l} \% \ new \ constraints \\ :\!- \ block(a), \ block(b), \ block(c). \\ :\!- \ block(c), \ block(d). \\ :\!- \ block(d), \ block(b), \ block(a). \end{array}$$

The above LPOD program does not have any answer set. To avoid undesirable results like that in the example above, we can formulate a sufficient condition for existence of monitors.

Definition 4. *A* priority graph G_M*, for a monitor* M*, is defined as follows:*

- *vertices corresponds to actions;*
- *there is an arc* $\langle a, b\rangle \in G_M$ *iff actions* a *and* b *appear together in a rule of* M*.*

Conjecture 1
Let M *be a monitor and let* $\tau(M)$ *be the translation of such monitor as in definition (3).*
$\tau(M)$ *has an answer set (corresponding to a* minimal and preferred *model) if the priority graph* G_M *is free from odd length cycles.*

5.1 Hybrid monitors, minimality and preferential blocking

Definition 3 describes how to generate a LPODs that yields Answer Sets representing minimal and preferred monitors. However, such transformation from PPDL to LPOD should be slightly modified when the monitor contains both prioritized constraints (8), and simple PDL constraints like:

$$\textbf{never } a_j \textbf{ if } C. \tag{15}$$

In this case we have an *hybrid* monitor and we have to treat both kinds of constraints separately. To do so, we provide an algorithm where the solver is called twice and creation of program $\tau(M)$ is divided into several steps.
Before describing the two steps of the reduction, let us define the following sets:

$S_b(M) = \{block(a_j) \ :\!- \ C_j, \ \tilde{\forall} \textbf{ never } a_j \textbf{ if } C_j \ \in \ M.\}$

$A_{true} = \{block(a_j) \ s.t. \ \pi_p \ \Rightarrow \ block(a_j)\}$

$R = \{block(a_1) \ \times \ \ldots \ \times \ block(a_j) \ where \ block(a_i) \ \in \ A_{true}, \ for \ some \ i = 1..j\}$

$C_{new} = \{:\!- \ block(a_1), \ \ldots, \ block(a_j).\tilde{\forall} \ preferential \ blocking \ rule \ \notin \ R\}$

Here follows a formal description of such procedure:

Definition 5. *Let M be a monitor, let P be a policy and let $\tau_i(M)$ be the $i-th$ step in translation of such monitor into a logic program* [6].

1. $\tau_0(M) : \ \tau(P) \ + \ S_b$
2. A_{true} *is obtained by calling* PSMODELS *on* $\tau_0(M)$
3. $\tau_1(M) : \ \tau_0(M) \ - \ S_b \ + \ A_{true}$.
4. *if* A_{true} *is a model of* $\tau_1(M)$, *then STOP, else*
5. $\tau_2(M) : \ \tau_1(M) \ - \ R$.
6. $\tau_3(M) : \ \tau_2(M) \ + \ C_{new}$.

Example 5. Consider the hybrid monitor M as follows:

$$\begin{array}{l}\textbf{never } a \ \times \ b \ \times \ c. \\ \textbf{never } c \ \times \ d. \\ \textbf{never } d \ \times \ e. \\ \textbf{never } e \ \times \ a \ \times \ b.\end{array}$$

$$\begin{array}{l}\textbf{never } a \textbf{ if} c_1. \\ \textbf{never } c \textbf{ if} c_2. \\ \textbf{never } d \textbf{ if} c_3.\end{array}$$

[6] Notice that a monitor M can contain both *simple* constraints as in (15) and preference constraints as in (8). The former type of constraints are translated as in (9), whereas the latter type of constraints are translated into the following *blocking rule*:

$$:\!- \ block(a_j), \ C. \tag{16}$$

Suppose condition c_1, c_2 are true. With the standard procedure, we should add constraints that will make the program inconsistent, even though there are minimal preferred monitors. However, applying the modified procedure described above, we get the needed answer sets. Indeed the translation gives the following program:

$$
\begin{array}{l}
block(a) \times block(b) \times block(c). \\
block(c) \times block(d). \\
block(d) \times block(e). \\
block(e) \times block(a) \times block(b). \\
\\
\% \textit{ new constraints} \\
:\!- \; block(d),\; block(e).
\end{array}
$$

We added only an additional constraint, as the other blocking rules contains at least one among the block atoms already added in the model, i.e. $block(a)$, $block(c)$. This program has two minimal *and* preferred model:

$$
\begin{array}{rcl}
M_1 & \equiv & \{block(a),\; block(c),\; block(d),\; accept(b),\; accept(e),\; \ldots\}. \\
M_2 & \equiv & \{block(a),\; block(c),\; block(e),\; accept(b),\; accept(d),\; \ldots\}.
\end{array} \tag{17}
$$

6 Final considerations and future work

We believe that flexible policy languages, by which applications can specify whether and how to enforce constraints, are required. Starting from Chomicki, Lobo et al. work on PDL, our PPDL language enables the specification of user-preferences in policy enforcement (cancellation of actions), yet in its first version it was somewhat limited.

First, PPDL syntax of preference rules required that preference relation over a set of actions which cannot be executed together needs to be a total order.

Second, as PPDL monitors are executed by translating them to LPOD programs, minimality of the solution, i.e., minimality of the set of block actions, cannot be guaranteed. As Brewka point out, and our discussion corroborates, maximizing preference implies giving up minimality of the model in the set-theoretic sense.

In this article we have discussed two techniques that allow us to capture user preferences in a PPDL monitor without giving up set-theoretic minimality of the answer.

The first solution was given for PPDL monitors where all action constraints are stated in terms of preferential blocking. In such cases, the minimality requirement is captured by a new translation, that introduced several new constraints. A sufficient condition was given that ensures correctness of ASP interpretation of the resulting program w.r.t. PPDL semantics.

The second, perhaps less elegant solution was given for PPDL monitors that mix preferential and traditional (using **never**) constraints. In such a case, we

showed a two-steps translation from PPDL to ASP that uses the original Lobo's PDL to ASP translation as an intermediate step.

The proof of correctness of this translation will appear in the final version of this article. Another topic for future research is a comparison with existing work on ordered disjunctions in DLV [Buccafurri et al., 1998].

References

[Buccafurri et al., 1998] Buccafurri F., Leone L. and Rullo P., 1998. Disjunctive Ordered Logic: Semantics and Expressiveness. Proc. of KR'98. MIT Press, pp. 418-431.

[Bertino et al., 2003] Bertino E., Mileo A. and Provetti A., 2003. *PDL with Maximum Consistency Monitors.* Proc. of ISMIS 2003 Conference. Springer LNAI, to appear. Available from *http://mag.usr.dsi.unimi.it/*

[Bertino et al., 2003b] Bertino E., Mileo A. and Provetti A., 2003. *Policy Monitoring with User-Preferences in PDL.* Proc. of NRAC 2003 Workshop. To appear. Available from *http://mag.usr.dsi.unimi.it/*

[Brewka, 1996] Brewka G., 1996. *Well-founded semantics for extended logic programs with dynamic preferences.* Journal of AI Research 4:19-36.

[Brewka, Benferhat, Le Berre, 2002] Brewka G., Benferhat S., and Le Berre D., 2002. *Qualitative choice logic.* Proc. of Principles of Knowledge Representation and Reasoning, KR-02.

[Brewka, 2002] Brewka, G., 2002. *Logic Programming with Ordered Disjunction.* Proc. of AAAI-02. Extended version presented at NMR-02.

[Brewka et al., 2002] Brewka, G., Niemelä I and Syrjänen T., 2002. *Implementing Ordered Disjunction Using Answer Set Solvers for Normal Programs.* Proc. of JELIA'02. Springer Verlag LNAI.

[Chomicki et al., 2000] Chomicki J., Lobo J. and Naqvi S., 2000. *A logic programming approach to conflict resolution in policy management.* Proc. of KR2000, Morgan Kaufmann, pp 121–132.

[Chomicki et al., 2001] Chomicki J. and Lobo J., 2001. *Monitors for History-Based Policies.* Proc. of Int'l Workshop on Policies for Distributed Systems and Networks. Springer-Verlag, LNCS 1995, pp. 57–72.

[Chomicki et al., 2003] Chomicki J., Lobo J. and Naqvi S., 2003. *Conflict Resolution using Logic Programming.* To appear on IEEE Transactions on Knowledge and Data Engineering 15:2.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V., 1991. Classical negation in logic programs and disjunctive databases. New Generation Computing: 365–387.

[Lobo et al., 1999] Lobo J., Bhatia R. and Naqvi S., 1999. A Policy Description Language, in Proc. of *AAAI/IAAI, 1999*, pp. 291–298.

[Schaub, Wang, 2001] Schaub T., and Wang K., 2001. *A comparative study of logic programs with preference.* Proc. of Int'l. Joint Conference on AI, IJCAI-01.

[Systems] Web location of the most known ASP solvers.
Cmodels: *http://www.cs.utexas.edu/users/yuliya/*
aspps: *http://www.cs.uky.edu/ai/aspps/*
DLV: *http://www.dbai.tuwien.ac.at/proj/dlv/*
Smodels: *http://www.tcs.hut.fi/Software/smodels/*