DBAI

TECHNICAL REPORT

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

# Disjunctive Logic Programs with Inheritance

**DBAI-TR-99-30**

**Francesco Buccafurri**[1] **Wolfgang Faber**[2] **Nicola Leone**[3]

DBAI TECHNICAL REPORT

MAY 1999

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Paniglgasse 16
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

TU

TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-99-30, MAY 1999

# Disjunctive Logic Programs with Inheritance

**Abstract.** The paper proposes a new knowledge representation language, called $\mathrm{DLP}^{<}$, which extends disjunctive logic programming (with strong negation) by inheritance. The addition of inheritance enhances the knowledge modeling features of the language providing a natural representation of default reasoning with exceptions.
A declarative model-theoretic semantics of $\mathrm{DLP}^{<}$ is provided, which is shown to generalize the answer set semantics of disjunctive logic programs.
The knowledge modeling features of the language are illustrated by encoding classical nonmonotonic problems in $\mathrm{DLP}^{<}$.
The complexity of $\mathrm{DLP}^{<}$ is analyzed, proving that inheritance does not cause any computational overhead, as reasoning in $\mathrm{DLP}^{<}$ has exactly the same complexity as reasoning in disjunctive logic programming. This is confirmed by the existence of an efficient translation from $\mathrm{DLP}^{<}$ to plain disjunctive logic programming. Using this translation, an advanced KR system supporting the $\mathrm{DLP}^{<}$ language has been implemented on top of the `dlv` system.
**Keywords.** Nonmonotonic Reasoning, Knowledge Representation, Disjunctive Logic Programming, Inheritance, Exceptions.

[1]DIMET, Universitá di Reggio Calabria, 89100, Reggio Calabria, Italia. E-mail: bucca@ns.ing.unirc.it
[2]Institut für Informationssysteme, Technische Universität Wien, Paniglgasse 16, A-1040 Wien, Austria. E-mail: faber@dbai.tuwien.ac.at
[3]Institut für Informationssysteme, Technische Universität Wien, Paniglgasse 16, A-1040 Wien, Austria. E-mail: leone@dbai.tuwien.ac.at

**Acknowledgements**: Supported in part by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

Copyright © 1999 by the authors

# 1 Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and (NAF) negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [5, 23, 15]. One of the attractions of disjunctive logic programming is its ability to naturally model incomplete knowledge [5, 23]. The need to differentiate atoms, which are false because of the failure to prove them true (NAF, or CWA negation), from atoms the falsity of which is explicitly provable, led to extend disjunctive logic programs by strong negation [15]. Strong negation, permitted also in the rules' heads, further enhances the knowledge modeling features of the language, and its usefulness is widely acknowledged in the literature [1, 5, 20, 2, 31, 3]. However, it does not allow to represent default reasoning with exceptions in a direct and natural way. Indeed, to render a default rule $r$ *defeasible*, $r$ must be equipped by an extra negative literal, which "blocks" inferences from $r$ for abnormal instances [17]. For instance, to encode the famous NMR example stating that birds *normally* fly while penguins do not fly, one should write the rule $fly(X) \leftarrow bird(X),\ not\ \neg fly(X)$ along with the fact $\neg fly(penguin)$.[1]

This paper proposes an extension of disjunctive logic programming by inheritance, called $\mathrm{DLP}^<$. The addition of inheritance enhances the knowledge modeling features of the language. Possible conflicts are solved in favour of the rules which are "more specific" according to the inheritance hierarchy. This way, a direct and natural representation of default reasoning with exceptions is achieved (e.g., defeasible rules do not need to be equipped with extra literals as above – see section 4).

The main contribution of the paper are the following.

- We formally define the $\mathrm{DLP}^<$ language, providing a declarative model theoretic semantics of $\mathrm{DLP}^<$, which is shown to generalize the answer set semantics of [15].
- We illustrate the knowledge modeling features of the language by encoding classical nonmonotonic problems in $\mathrm{DLP}^<$. Interestingly, $\mathrm{DLP}^<$ supplies a natural representation also of frame axioms.
- We analyze the computational complexity of reasoning over $\mathrm{DLP}^<$ programs. Importantly, while inheritance enhances the knowledge modeling ability of disjunctive logic programming, it does not cause any computational overhead, as reasoning in $\mathrm{DLP}^<$ has exactly the same complexity as reasoning in disjunctive logic programming.
- We compare $\mathrm{DLP}^<$ to related works proposed in the literature. In particular, we evidentiate the differences between $\mathrm{DLP}^<$ and Disjunctive Ordered Logic ($\mathcal{DOL}$) [7]; we point out the relation to the answer set semantics of [15]; we compare $\mathrm{DLP}^<$ with prioritized disjunctive logic programs [31], and analyze its relationships to inheritance networks [33] and to updates of logic programs.
- We implement a $\mathrm{DLP}^<$ system. To this end, we first design an efficient translation from $\mathrm{DLP}^<$ to plain disjunctive logic programming. Then, using this translation, we implement a $\mathrm{DLP}^<$ evaluator on top of the `dlv` system [12]. The system prototype can be freely retrieved from [14]. To the best of our knowledge, this is the first fully operational system supporting a prioritized disjunctive logic programming language.

[1] *not* and $\neg$ denote the weak negation symbol and the strong negation symbol, respectively.

## 2 Syntax of $\mathrm{DLP}^<$

This section provides a formal description of syntactical constructs of the language.

Let the following disjoint sets be given: a set $\mathcal{V}$ of *variables*, a set $\Pi$ of *predicates*, a set $\Lambda$ of *constants*, and a finite partially ordered set of symbols $(\mathcal{O}, <)$, where $\mathcal{O}$ is a set of strings, called *object identifiers*, and $<$ is a strict partial order (i.e., the relation $<$ is: (i) irreflexive – $c \not< c \ \ \forall c \in \mathcal{O}$, and (ii) transitive – $a < b \wedge b < c \Rightarrow a < c \ \ \forall a, b, c \in \mathcal{O}$).

A $term$ is either a constant in $\Lambda$ or a variable in $\mathcal{V}$ (note that function symbols are not considered in this paper).

An $atom$ is a construct of the form $a(t_1, ..., t_n)$, where $a$ is a $predicate$ of arity $n$ in $\Pi$ and $t_1, ..., t_n$ are terms.

A $literal$ is either a $positive\ literal$ $p$ or a $negative\ literal$ $\neg\ p$, where $p$ is an atom ($\neg$ is the $strong\ negation$ symbol). Two literals are $complementary$ if they are of the form $p$ and $\neg p$, for some atom $p$.

Given a literal $L$, $\neg.L$ denotes its complementary literal. Accordingly, given a set $A$ of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$.

A *rule* $r$ is an expression of the form:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, not\ b_{k+1}, \cdots, not\ b_m, \qquad n \geq 1,\ m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are literals, and $not$ is the *negation as failure* symbol. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of $r$, while the conjunction $b_1, ..., b_k, not\ b_{k+1}, ..., not\ b_m$ is the *body* of $r$. $b_1, ..., b_k$ is called the *positive part* of the body of $r$ and $not\ b_{k+1}, ..., not\ b_m$ is called the *NAF (negation as failure) part* of the body of $r$. We often denote the sets of literals appearing in the head, in the positive part of the body, and in the NAF part of the body of a rule $r$ by $Head(r)$, $Body^+(r)$, and $Body^-(r)$, respectively.

An *object* $o$ is a pair $\langle oid(o), \Sigma(o)\rangle$, where $oid(o)$ is an object identifier in $\mathcal{O}$ and $\Sigma(o)$ is a (possibly empty) set of rules.

A *knowledge base* on $\mathcal{O}$ is a set of objects, one for each element of $\mathcal{O}$.

Given a knowledge base $\mathcal{K}$ and an object identifier $o \in \mathcal{O}$, the $\mathrm{DLP}^<$ *program for* $o$ (*on* $\mathcal{K}$) is the set of objects $\mathcal{P} = \{(o', \Sigma(o')) \in \mathcal{K} \mid o = o' \text{ or } o < o'\}$.

The relation $<$ induces a partial order on $\mathcal{P}$ in the obvious way, that is, given $o_i = (oid(o_i), \Sigma(o_i))$ and $o_j = (oid(o_j), \Sigma(o_j))$, $o_i < o_j$ iff $oid(o_i) < oid(o_j)$ (read "$o_i$ is more specific than $o_j$").

A term, an atom, a literal, a rule, or program is *ground* if no variable appears in it.

Informally, a knowledge base can be viewed as a set of *objects* embedding the definition of their properties specified through disjunctive logic rules, organized in a IS-A (inheritance) hierarchy (induced by the relation $<$). A program $\mathcal{P}$ for an object $o$ on a knowledge base $\mathcal{K}$ consists of the portion of $\mathcal{K}$ "seen"from $o$ looking up in the IS-A hierarchy. Thanks to the inheritance mechanism, $\mathcal{P}$ incorporates the knowledge explicitely defined for $o$ plus the knowledge inherited from the higher objects.

If a knowledge base admits a *bottom* element (i.e., an object less than all the other objects, by the relation $<$), we usually refer to the knowledge base as "program", since it is equal to the program for the bottom element.

Moreover, we represent the transitive-reduction of the relation $<$ on the objects.[2] In order to represent the membership of a pair of objects (resp., object identifiers) $(o_2, o_1)$ to the transitive-reduction of $<$ we use the notation $o_2 : o_1$, and say that $o_2$ is a *sub-object* of $o_1$.

**Example 1** Consider the following program $\mathcal{P}$:

[2] $(a, b)$ is in the transitive-reduction of $<$ iff $a < b$ and there is no $c$ such that $a < c$ and $c < b$.

$o_1 \quad \{\ a \vee \neg b \leftarrow c, not\ d\ \}$
$o_2 : o_1\ \{\ b \leftarrow \quad \neg a \vee c \leftarrow \quad c \leftarrow b\ \}$

$\mathcal{P}$ consists of two objects $o_1$ and $o_2$. $o_2$ is a sub-object of $o_1$. According to the convention illustrated above, the knowledge base on which $\mathcal{P}$ is defined coincides with $\mathcal{P}$, and the object for which $\mathcal{P}$ is defined is $o_2$ (the bottom object). ∎

# 3 Semantics of $\mathrm{DLP}^<$

In this section we assume that a knowledge base $\mathcal{K}$ is given and an object $o$ has been fixed. Let $\mathcal{P}$ be the $\mathrm{DLP}^<$ program for $o$ on $\mathcal{K}$. The *Universe* $U_\mathcal{P}$ of $\mathcal{P}$ is the set of all constants appearing in the rules. The *Base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of all possible ground literals constructible from the predicates appearing in the rules of $\mathcal{P}$ and the constants occurring in $U_\mathcal{P}$. Note that, unlike in traditional logic programming the Base of a $\mathrm{DLP}^<$ program contains both positive and negative literals. Given a rule $r$ occurring in $\mathcal{P}$, a *ground instance* of $r$ is a rule obtained from $r$ by replacing every variable $X$ in $r$ by $\sigma(r)$, where $\sigma$ is a mapping from the variables occurring in $r$ to the constants in $U_\mathcal{P}$. We denote by $ground(\mathcal{P})$ the (finite) multiset of all instances of the rules occurring in $\mathcal{P}$. The reason why $ground(\mathcal{P})$ is a multiset is that a rule may appear in several different objects of $\mathcal{P}$, and we require the respective ground instances be distinct. Hence, we can define a function $obj_of$ from ground instance of rules in $ground(\mathcal{P})$ onto the set $\mathcal{O}$ of the object identifiers, associating with a ground instance $\overline{r}$ of $r$ the (unique) object of $r$.

A subset of ground literals in $B_\mathcal{P}$ is said to be *consistent* if no pair of complementary literals is in it. An *interpretation* $I$ is a consistent subset of $B_\mathcal{P}$. Given an interpretation $I \subseteq B_\mathcal{P}$, a ground literal (either positive or negative) $L$ is *true* w.r.t. $I$ if $L \in I$ holds. $L$ is *false* w.r.t. $I$ otherwise.

Given a rule $r \in ground(\mathcal{P})$, the head of $r$ is *true* in $I$ if at least one literal of the head is true w.r.t $I$. The body of $r$ is *true* in $I$ if: (i) every literal in $Body^+(r)$ is true w.r.t. $I$, and (ii) every literal in $Body^-(r)$ is false w.r.t. $I$. Rule $r$ is *satisfied* in $I$ if either the head of $r$ is true in $I$ or the body of $r$ is not true in $I$.

Two ground rules $r_1$ and $r_2$ are *conflicting on* $L$ if $L \in Head(r_1)$ and $\neg.L \in Head(r_2)$.

Next we introduce the concept of a *model* for a $\mathrm{DLP}^<$-program. Unlike in traditional logic programming, the notion of satisfiability of rules is not sufficient for this goal, as it does not take into account the presence of explicit contradictions. Hence, we first present some preliminary definitions.

**Definition 1** For an interpretation $I$, two conflicting ground rules $r_1, r_2 \in ground(\mathcal{P})$, such that $L \in Head(r_2)$, we say that $r_1$ *overrides* $r_2$ *on* $L$ *in* $I$ if: 1) $obj_of(r_1) < obj_of(r_2)$, 2) $\neg.L \in I$, and 3) the body of $r_2$ is true in $I$.

A rule $r \in ground(\mathcal{P})$ is *overridden in* $I$ if for each $L \in Head(r)$ there exists $r_1 \in ground(\mathcal{P})$ such that $r_1$ overrides $r$ on $L$ in $I$. ∎

Intuitively, the notion of overriding allows us to solve conflicts arising between rules with complementary heads. For instance, suppose that both $a$ and $\neg a$ are derivable in $I$ from rules $r$ and $r'$, respectively. If $r$ is more specific than $r'$ in the inheritance hierarchy, then $r'$ is overruled, meaning that $a$ should be preferred to $\neg a$ because it is derivable from a more trustable rule.

**Example 2** Consider the program $\mathcal{P}$ of Example 1. Let $I = \{\neg a, b, c\}$ be an interpretation. Rule $\neg a \vee c \leftarrow$ in the object $o_2$ overrides rule $a \vee \neg b \leftarrow c, not\ d$ in $o_1$ on the literal $a$ in $I$. Moreover, rule $b \leftarrow$ in $o_2$ overrides rule $a \vee \neg b \leftarrow c, not\ d$ in $o_1$ on the literal $\neg b$ in $I$. Thus, the rule $a \vee \neg b \leftarrow c, not\ d$ in $o_1$ is overridden in $I$. ∎

**Definition 2** Let $I$ be an interpretation for $\mathcal{P}$. $I$ is a *model for* $\mathcal{P}$ if every rule in $ground(\mathcal{P})$ is satisfied or overridden in $I$. $I$ is a *minimal model for* $\mathcal{P}$ if no (proper) subset of $I$ is a model for $\mathcal{P}$. ∎

**Definition 3** Given an interpretation $I$ for $\mathcal{P}$, the *reduction of* $\mathcal{P}$ *w.r.t.* $I$, denoted by $G_I(\mathcal{P})$, is the set of rules obtained from $ground(\mathcal{P})$ by 1) removing every rule overridden in $I$, 2) removing every rule $r$ such that $Body^-(r) \cap I \neq \emptyset$, 3) removing the NAF part from the bodies of the remaining rules. ∎

**Example 3** Consider the program $\mathcal{P}$ of Example 1. Let $I$ be the interpretation $\{\neg a, b, c\}$. As shown in Example 2, rule $a \vee \neg b \leftarrow c, not\ d$ is overridden in $I$. Thus, $G_I(\mathcal{P})$ is the set of rules $\{\neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$. Consider now the interpretation $M = \{a, b, c\}$. It is easy to see that $G_M(\mathcal{P}) = \{a \vee \neg b \leftarrow c, \neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$. ∎

We observe that the reduction of a program is simply a set of ground rules. Given a set $S$ of ground rules, we denote by $pos(S)$ the positive disjunctive program (called the *positive version of* $S$), obtained from $S$ by considering each negative literal $\neg p(\bar{X})$ as a positive one with predicate symbol $\neg p$.

**Definition 4** Let $M$ be a model for $\mathcal{P}$. We say that $M$ is a *(*$\mathrm{DLP}^<$*-)answer set* for $\mathcal{P}$ if $M$ is a minimal model of the positive version $pos(G_M(\mathcal{P}))$ of $G_M(\mathcal{P})$. ∎

**Example 4** Consider the program $\mathcal{P}$ of Example 1:

It is easy to see that the interpretation $I$ of Example 3 is not an answer set for $\mathcal{P}$. Indeed, although $I$ is a model for $pos(G_I(\mathcal{P}))$ it is not minimal, since the interpretation $\{b, c\}$ is a model for $pos(G_I(\mathcal{P}))$, too. Note that the interpretation $I' = \{b, c\}$ is not an answer set for $\mathcal{P}$ either, because $G_{I'}(\mathcal{P}) = \{a \vee \neg b \leftarrow c, \neg a \vee c \leftarrow, b \leftarrow, c \leftarrow b\}$, and $I'$ is not a model for $pos(G_{I'}(\mathcal{P}))$, since the rule $a \vee \neg b \leftarrow c$ is not satisfied in $I'$.

On the other hand, the interpretation $M$ of Example 3 is an answer set for $\mathcal{P}$, since $M$ is a minimal model for $pos(G_M(\mathcal{P}))$. Moreover, it can be easily realized that $M$ is the only answer set for $\mathcal{P}$. ∎

# 4 Knowledge Representation with $\mathrm{DLP}^<$

In this section, we present a number of examples which illustrate how knowledge can be represented using $\mathrm{DLP}^<$. To start, we show the $\mathrm{DLP}^<$ encoding of a classical example of nonmonotonic reasoning.

**Example 5** Consider the following program $\mathcal{P}$ with $\mathcal{O}(\mathcal{P})$ consisting of three objects $bird$, $penguin$ and $tweety$, such that $penguin$ is a sub-object of $bird$ and $tweety$ is a sub-object of $penguin$:

$bird$ $\{ flies \leftarrow \}$
$penguin : bird$ $\{ \neg flies \leftarrow \}$
$tweety : penguin$ $\{ \}$

Unlike in traditional logic programming, our language supports two types of negation, that is *strong negation* and *negation as failure*. Strong negation is useful to express negative pieces of information under the complete information assumption. Hence, a negative fact (by strong negation) is true only if it is explicitely derived from the rules of the program. As a consequence, the head of rules may contain also such negative literals and rules can be conflicting on some literals. According to the inheritance principles, the ordering relationship between objects can help us to assign different levels of reliability to the rules, allowing us to

solve possible conflicts. For instance, in our example, the contradicting conclusion *tweety both flies and does not fly* seems to be entailed from the program (as $tweety$ is a $penguin$ and $penguins$ are $birds$, both $flies$ and $\neg flies$ can be derived from the rules of the program). However, this is not the case. Indeed, the "lower"rule $\neg flies \leftarrow$ specified in the object $penguin$ is considered as a sort of refinement to the first general rule, and thus the meaning of the program is rather clear: *tweety does not fly*, as tweety is a penguin. That is, $\neg flies \leftarrow$ is preferred to the default rule $flies \leftarrow$ as the hierarchy explicitely states the specificity of the former. Intuitively, there is no doubt that $M = \{\neg flies\}$ is the only reasonable conclusion. ∎

The next example, from the field of database authorizations, combines the use of both weak and strong negation.

**Example 6** Consider the following knowledge base representing a set of security specification about a simple *part-of* hierarchy of objects.

$o_1$ $\{ r_1 : authorize(bob) \leftarrow not\ authorize(ann)$
$r_2 : authorize(ann) \vee authorize(tom) \leftarrow not\ \neg authorize(alice)\ \}$
$o_2 : o_1\ \{ r_3 : \neg authorize(alice) \leftarrow \}$
$o_3 : o_1\ \{ r_4 : \neg authorize(bob) \leftarrow \}$

Objects $o_2$ is part-of the object $o_1$ as well as $o_3$ is part-of $o_1$. Access authorizations to objects are specified by rules with head predicate $authorize$ and subjects to which authorizations are granted appear as arguments. Strong negation is utilized to encode negative authorizations that represent explicit denials. Negation as failure is used to specify the absence of authorization (either positive or negative). Inheritance implements the automatic propagation of authorizations from an object to all its sub-objects. The overriding mechanism allows us to represent exceptions: for instance, if an object $o$ inherits a positive authorization but a denial for the same subject is specified in $o$, then the negative authorization prevails on the positive one.

Consider the program $\mathcal{P}_{o_2} = \{(o_1, \{r_1, r_2\}), (o_2, \{r_3\})\}$ for the object $o_2$ on the above knowledge base. This program defines the access control for the object $o_2$. Thanks to the inheritance mechanism, authorizations specified for the object $o_1$, to which $o_2$ belongs, are propagated also to $o_2$. It consists of rules $r_1$, $r_2$ (inherited from $o_1$) and $r_3$. Rule $r_1$ states that $bob$ is authorized to access object $o_2$ provided that no authorization for $ann$ to access $o_2$ exists. Rule $r_2$ authorizes either $ann$ or $tom$ to access $o_2$ provided that no denial for $alice$ to access $o_2$ is derived. Finally, rule $r_3$ defines a denial for $alice$ to access object $o_2$. Due to the absence of the authorization for $ann$, the authorization to $bob$ of accessing the object $o_2$ is derived (by rule $r_1$). Further, the explicit denial to access the object $o_2$ for $alice$ (rule $r_3$) allows to derive neither authorization for $ann$ nor for $tom$ (by rule $r_2$). Hence, the only answer set of this program is $\{authorize(bob), \neg authorize(alice)\}$.

Consider now the program $\mathcal{P}_{o_3} = \{(o_1, \{r_1, r_2\}), (o_3, \{r_4\})\}$ for the object $o_3$. Rule $r_4$ defines a denial for $bob$ to access object $o_3$. The authorization for $bob$ (defined by rule $r_1$) is no longer derived. Indeed, even if rule $r_1$ allows to derive such an authorization due to the absence of authorizations for $ann$, it is overridden by the explicit denial (rule $r_4$) defined in the object $o_3$ (i.e., at a more specific level). The body of rule $r_2$ inherited from $o_1$ is true for this program since no denial for alice can be derived, and it entails a mutual exclusive access to object $o_3$ for $ann$ and $tom$ (note that no other head contains $authorize(ann)$ or $authorize(bob)$). The program $\mathcal{P}_{o_3}$ has two answer sets, namely $\{authorize(ann), \neg authorize(bob)\}$ and $\{authorize(tom), \neg authorize(bob)\}$ representing two alternative authorization sets utilizable to grant the access to the object $o_3$. ∎

## Solving the Frame Problem

The frame problem has first been addressed by McCarthy and Hayes [27], and in the meantime a lot of research has been conducted to overcome it (see e.g. [32] for a survey).

In short, the frame problem arises in planning, when actions and fluents are specified: An action affects some of the fluents, but all unrelated fluents should remain as they are. In most formulations using classical logic, one must specify for every pair of actions and unrelated fluents that the fluent remains unchanged. Clearly this is an undesirable overhead, since with $n$ actions and $m$ fluents, $n \times m$ clauses might have to be specified.

Instead, it would be nice to be able to specify for each fluent that it *"normally remains valid"* and that only actions which explicitly entail the contrary can change them.

Indeed, this goal can be achieved in a very elegant way using $\mathrm{DLP}^<$: One object contains the rules which specify *inertia* (the fact that fluents normally do not change). Another object inherits from it and specifies the actions and the effects of actions — in this way a very natural, straightforward and effective representation is achieved, which avoids the frame problem.

**Example 7** As an example we show how the famous Yale Shooting Problem, which is due to Hanks and McDermott [19], can be represented and solved with $\mathrm{DLP}^<$:

The scenario involves an individual (or in a less violent version a turkey), who can be shot with a gun. There are two fluents, $alive$ and $loaded$, which intuitively mean that the individual is alive and that the gun is loaded, respectively. There are three actions, $load$, $wait$ and $shoot$. Loading has the effect that the gun is loaded afterwards, shooting with the loaded gun has the effect that the individual is no longer alive afterwards (and also that the gun is unloaded, but this not really important), and waiting has no effects. It is also known that initially the individual is alive, and that first the gun is loaded, and after waiting, the gun is shot with. The question is: Which fluents hold after these actions and between them?

In our encoding, the $inertia$ object contains the defaults for the fluents, the $domain$ object additionally specifies the effects of actions, while the $yale$ object encodes the problem instance including the time framework.[3]

$$
\begin{array}{rl}
inertia\,\{ & alive(T1) \leftarrow alive(T), next(T,T1) \\
 & \neg alive(T1) \leftarrow \neg alive(T), next(T,T1) \\
 & loaded(T1) \leftarrow loaded(T), next(T,T1) \\
 & \neg loaded(T1) \leftarrow \neg loaded(T), next(T,T1)\} \\
domain : inertia\,\{ & loaded(T1) \leftarrow load(T), next(T,T1) \\
 & \neg loaded(T1) \leftarrow shoot(T), loaded(T), next(T,T1) \\
 & \neg alive(T1) \leftarrow shoot(T), loaded(T), next(T,T1)\} \\
yale : domain\,\{ & load(0) \leftarrow \quad wait(1) \leftarrow \quad shoot(2) \leftarrow \quad alive(0) \leftarrow \\
 & next(0,1) \leftarrow \qquad next(1,2) \leftarrow \qquad next(2,3) \leftarrow\}
\end{array}
$$

The only answer set for this program contains, besides the facts of the $yale$ object, $loaded(1)$, $loaded(2)$, $alive(0)$, $alive(1)$, $alive(2)$ and $\neg loaded(3)$, $\neg alive(3)$. That is, the individual is alive until the shoot action is taken, and no longer alive afterwards, and the gun is loaded between loading and shooting. ∎

We want to point out that this formalism is equally suited for solving problems which involve finding a plan (i.e. a sequence of actions), rather than determining the effects of a given plan as in the Yale Shooting

[3] With our prototype the more convenient builtins of `dlv` can be used instead.

Problem: You have to add a rule $action(T) \vee \neg action(T) \leftarrow next(T, T1)$ for every action, and you have to specify the goal state by a query, e.g. $\neg alive(3), \neg loaded(3)$?[4].

Below you find a classical example: The blocksworld domain and the Sussman anomaly as a concrete problem.

**Example 8** In [13], several planning problems, including the blocksworld problems, are encoded using disjunctive datalog.

In general, planning problems can be effectively specified using action languages (e.g. [16, 8, 18, 22]). Then, a translation from these languages to another language (in our case $\mathrm{DLP}^{<}$) is applied.

We omit the step of describing an action language and the associated translation, and directly show the encoding of an example planning domain in disjunctive datalog. This encoding is rather different from the one presented in [13].

The objects in the blocksworld are one $table$ and an arbitrary number of labeled cubic $block$s. Together, they are referred to as $locations$.

The state of the blocksworld at a particular time can be fully specified by the fluent $on(B, L, T)$, which specifies that block $B$ resides on location $L$ at time $T$.

So, first we state in the object $bw_inertia$ that the fluent $on$ is inertial.

$$bw_inertia\ \{\quad on(B, L, T1) \leftarrow on(B, L, T), next(T, T1)\}$$

We continue to define the blocksworld domain in the object $bw_domain$, which inherits from the inertia object:

$$\begin{aligned} bw_domain : bw_inertia\ \{\quad & move(B, L, T) \vee \neg move(B, L, T) \leftarrow block(B), location(L), actiontime(T) \\ & on(B, L, T1) \leftarrow move(B, L, T), next(T, T1) \\ & \neg on(B, L, T1) \leftarrow move(B, L1, T), on(B, L, T), next(T, T1) \\ & \leftarrow move(B, L, T), on(B1, B, T) \\ & \leftarrow move(B, B1, T), on(B2, B1, T), block(B1) \\ & \leftarrow move(B, B, T) \\ & \leftarrow move(B, L, T), move(B1, L1, T), B <> B1 \\ & \leftarrow move(B, L, T), move(B1, L1, T), L <> L1 \\ & actiontime(T) \leftarrow T < \#maxint, \#int(T) \\ & next(T, T1) \leftarrow \#succ(T, T1) \\ & location(table) \leftarrow \\ & location(B) \leftarrow block(B)\} \end{aligned}$$

There is one action, which is moving a block from one location to another location. A move is started at one point in time, and it is completed before the next time. At any time $T$, the action of moving a block $B$ to location $L$ may be initiated ($move(B, L, T)$) or not ($\neg move(B, L, T)$).

The following two rules specify the results of the move action: The moved block is at the target location at the next time, and no longer on the source location.

What follows are constraints, and their semantics is that in any answer set the conjunction of their literals must not be true.[5] Their respective meanings are:

[4]This is a dlv language feature which (for this example) is equivalent to the rules $h \leftarrow \neg alive(3), \neg loaded(3)$ and $i \leftarrow not\ h, not\ i$, meaning that only answer sets containing $\neg alive(3)$ and $\neg loaded(3)$ should be considered.

[5]We use constraints for clarity, but they can be eliminated by rewriting $\leftarrow B$ to $p \leftarrow B, not\ p$.

Figure 1: The Sussman Anomaly

- A moved block must be clear.
- The target of a move must be clear if it is a block (the table may hold an arbitrary number of blocks).
- A block may not be on itself.
- No two move actions may be performed at the same time (2 constraints).

Finally, the time is described by `dlv`'s integer built-in predicates and constants. The first time is always 0, and the last time $\#maxint$ is specified when invoking `dlv` via the command-line option `-N=k`. $\#int(T)$ represents all numbers in $[0..\#maxint]$, and $\#succ(T, T1)$ defines the successor relation in the same interval. $actiontime$ defines the times at which an action may be performed, that are all times except the last one.

What is left is the concrete problem instance, in our case the so-called Sussman Anomaly (see Figure 1):

$$sussman : bw_domain\ \{\ block(a) \leftarrow \quad block(b) \leftarrow \quad block(c) \leftarrow$$
$$on(b, table, 0) \leftarrow \quad on(c, a, 0) \leftarrow \quad on(a, table, 0) \leftarrow\}$$

$on(c, b, \#maxint), on(b, a, \#maxint), on(a, table, \#maxint)$?

Since different problem instances may involve different numbers of blocks, the blocks are defined here (note that usually the instance will be separated from the domain definition).

We give the initial situation by facts, while the goal situation is specified by a query. This query enforces that only those answer sets are computed, in which the conjunction of the query literals is true.

# 5 Computational Complexity

As for the classical nonmonotonic formalisms [25, 24, 30], two important decision problems, corresponding to two different reasoning tasks, arise in $\mathrm{DLP}^<$:
*(Brave Reasoning)* Given a $\mathrm{DLP}^<$ program $\mathcal{P}$ and a ground literal $L$, decide whether there exists an answer set $M$ for $\mathcal{P}$ such that $L$ is true w.r.t. $M$.
*(Cautious Reasoning)* Given a $\mathrm{DLP}^<$ program $\mathcal{P}$ and a ground literal $L$, decide whether $L$ is true in all answer sets for $\mathcal{P}$.

We next prove that the complexity of reasoning in $\mathrm{DLP}^<$ is exactly the same as in traditional disjunctive logic programming. That is, inheritance comes for free, as the addition of inheritance does not cause any computational overhead. We consider the propositional case, i.e., we consider ground $\mathrm{DLP}^<$ programs.

**Lemma 1** *Given a ground* $\mathrm{DLP}^<$ *program* $\mathcal{P}$ *and an interpretation* $M$ *for* $\mathcal{P}$*, deciding whether* $M$ *is an answer set for* $\mathcal{P}$ *is in* $\mathrm{coNP}$.

**Proof.** We check in NP that $M$ is not an answer set of $\mathcal{P}$ as follows. Guess a subset $I$ of $M$, and verify that: (1) $M$ is not a model for $pos(G_M(\mathcal{P}))$, or (2) $I$ is a model for $pos(G_M(\mathcal{P}))$ and $I \subset M$. The construction of $pos(G_M(\mathcal{P}))$ (see Definition 3) is feasible in polynomial time, and the tasks (1) and (2) are clearly tractable. Thus, deciding whether $M$ is not an answer set for $\mathcal{P}$ is in NP, and, consequently, deciding whether $M$ is an answer set for $\mathcal{P}$ is in $\mathrm{coNP}$. ∎

**Theorem 1** *Brave Reasoning on* $\mathrm{DLP}^<$ *programs is* $\Sigma_2^P$*-complete.*

**Proof.** Given a ground $\mathrm{DLP}^<$ program $\mathcal{P}$ and a ground literal $L$, we verify that $L$ is a brave consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) $M$ is an answer set for $\mathcal{P}$, and (2) $L$ is true w.r.t. $M$. Task (2) is clearly polynomial; while (1) is in $\mathrm{coNP}$, by virtue of Lemma 1. The problem therefore lies in $\Sigma_2^P$.

$\Sigma_2^P$-hardness follows from Theorem 3 and the results in [9, 10]. ∎

**Theorem 2** *Cautious Reasoning on* $\mathrm{DLP}^<$ *programs is* $\Pi_2^P$*-complete.*

**Proof.** Given a ground $\mathrm{DLP}^<$ program $\mathcal{P}$ and a ground literal $L$, we verify that $L$ is not a cautious consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) $M$ is an answer set for $\mathcal{P}$, and (2) $L$ is not true w.r.t. $M$. Task (2) is clearly polynomial; while (1) is in $\mathrm{coNP}$, by virtue of Lemma 1. Therefore, the complement of cautious reasoning is in $\Sigma_2^P$, and cautious reasoning is in $\Pi_2^P$.

$\Pi_2^P$-hardness follows from Theorem 3 and the results in [9, 10]. ∎

# 6 Related Work

## 6.1 Answer Set Semantics

Answer set semantics, proposed by Gelfond and Lifschitz in [15], is the most widely acknowledged semantics for disjunctive logic programs with strong negation. For this reason, while defining the semantics of our language, we took care of ensuring full agreement with answer set semantics (on inheritance-free programs).

**Theorem 3** *Let* $\mathcal{P}$ *be a* $\mathrm{DLP}^<$ *program consisting of a single object* $o = \langle oid(o), \Sigma(o)\rangle$*. Then,* $M$ *is an answer set of* $\mathcal{P}$ *if and only if it is a consistent answer set of* $\Sigma(o)$ *(according with [15]).*

**Proof.** First we show that $G_M(\mathcal{P})$ is equal to $\Sigma(o)^M$ (as defined in [15]):

Deletion rule 1) of Definition 3 never applies, since for every literal L and any two rules $r_1$, $r_2 \in ground(\mathcal{P})$, $obj_of(r_1) \not< obj_of(r_2)$ holds, thus violating condition 1) in Definition 1 and therefore no rule can be overridden. It is evident that the deletion rules 2) and 3) of Definition 3 are equal to deletion rules (i) and (ii) of the definiton of $\Pi^S$ in §7 in [15], respectively. The first ones delete rules, where some NAF literal is contained in $M$, while the second ones delete all NAF literals of the remaining rules.

Next, we show that that the criteria for a consistent set $M$ of literals being an answer set of a positive (i.e. $NAF$ free) program (as in [15]) is equal to the notion of satisfaction:

Since the set is consistent, condition (ii) in §7 of [15] does not apply. Condition (i) says: $L_{k+1}, \ldots, Lm \in M$ (the body is true) implies that the head is true. This is logically equivalent to "The body is not true or the head is true", which is the definition of rule satisfaction.

In total we have that the minimal models of $pos(G_M(\mathcal{P}))$ are equal to the consistent answer sets of $\Sigma(o)^M$, since answer sets are minimal by definition.

Additionally, we require in Definition 4 that $M$ is also a model of $\mathcal{P}$, while in [15] there is no such requirement. However, all minimal models of $pos(G_M(\mathcal{P}))$ are also models of $\mathcal{P}$: All rules in $G_M(\mathcal{P})$ are satisfied, and only the deletion rules 2) and 3) of Definition 3 have been applied (as shown above). So for any rule $r$, which has been deleted by 2), some literal in $Body^-(r)$ is in $M$, so $r$'s body is not true, and thus $r$ is satisfied in $M$. If a rule $r$, which has been transformed by 3), is satisfied without $Body^-(r)$, then either $Head(r)$ is true or $Body^+(r)$ is not true, so adding any $NAF$ part to it does not change its satisfaction status. ∎

## 6.2 Disjunctive Ordered Logic

Disjunctive Ordered Logic ($\mathcal{DOL}$) is an extension of Disjunctive Logic Programming with strong negation and inheritance (without default negation) proposed in [7]. The $\mathrm{DLP}^<$ language incorporates some ideas taken from $\mathcal{DOL}$. However, the two languages are very different in several respects. Most importantly, unlike with $\mathrm{DLP}^<$, even if a program belongs to the common fragment of $\mathcal{DOL}$ and of the language of [15] (i.e., it contains neither inheritance nor default negation), $\mathcal{DOL}$ semantics is completely different from answer set semantics, because of a different way of handling contradictions.[6] In short, we observe the following differences between $\mathcal{DOL}$ and $\mathrm{DLP}^<$:

- $\mathcal{DOL}$ does not include default negation $not$, while $\mathrm{DLP}^<$ includes it.
- $\mathcal{DOL}$ and $\mathrm{DLP}^<$ have different semantics on the common fragment. Consider a program $\mathcal{P}$ consisting of a single object $o = \langle oid(o), \Sigma(o)\rangle$, where $\Sigma(o) = \{p \leftarrow;\ \ \neg p \leftarrow\}$. Then, according with $\mathcal{DOL}$, the semantics of $\mathcal{P}$ is given by two models, namely, $\{p\}$ and $\{\neg p\}$. On the contrary, $\mathcal{P}$ has no answer set according with $\mathrm{DLP}^<$ semantics.
- $\mathrm{DLP}^<$ generalizes (consistent) answer set semantics to disjunctive logic programs with inheritance; while $\mathcal{DOL}$ does not generalize it.

## 6.3 Prioritized Logic Programs

$\mathrm{DLP}^<$ can be also seen as an attempt to handle priorities in disjunctive logic programs (the lower the object in the inheritance hierarchy, the higher the priority of its rules).

There are several works on preference handling in logic programming [6, 17, 28, 20, 29, 31]. However, we are aware of only one previous work on priorities in disjunctive programs, namely, the paper by Sakama and Inoue [31]. This interesting work can be seen as an extension of answer set semantics to deal with priorities. Comparing the two works, under the viewpoint of priority handling, we observe the following:

- On priority free programs, the two languages yield essentially the same semantics, as they generalize answer set semantics and consistent answer set semantics, respectively.
- In [31], priorities are defined among literals, while priorities concern program's rules in $\mathrm{DLP}^<$.
- The different kind of priorities (on rules vs. literals) and the way how they are dealt with in the two approaches, imply different complexity in the respective reasonings. Indeed, from the simulation of abductive reasoning in the language of [31], and the complexity results on abduction derived in [11], it follows that brave reasoning is $\Sigma_3^P$-complete for the language of [31]. On the contrary, brave reasoning is "only" $\Sigma_2^P$-complete in $\mathrm{DLP}^<$[7].

[6] Actually, this was a main motivation for the authors to look for a different language.

[7] We refer to the complexity in the propositional case here.

A comparative analysis of the various approaches to the treatment of preferences in ($\vee$-free) logic programming has been carried out in [6].

### 6.4 Inheritance Networks

From a different perspective, the objects of a $\mathrm{DLP}^<$ program can also be seen as the nodes of an inheritance network.

We next show that $\mathrm{DLP}^<$ satisfies the basic semantic principles which are required for inheritance networks in [33].

[33] constitutes a fundamental attempt to present a formal mathematical theory of multiple inheritance with exceptions. The starting point of this work is the consideration that an intuitively acceptable semantics for inheritance must satisfy two basic requirements:

1. Being able to reason with redundant statements, and
2. not making unjustified choices in ambiguous situations.

Touretzky illustrates this intuition by means of two basic examples.

The former requirement is presented by means of the *Royal Elephant* example, in which we have the following knowledge: "Elephants are gray.", "Royal elephants are elephants.", "Royal elephants are not gray.", "Clyde is a royal elephant.", "Clyde is an elephant."

The last statement is clearly redundant; however, since it is consistent with the others there is no reason to rule it out. Touretzky shows that an intuitive semantics should be able to recognize that Clyde is not gray, while many systems fail in this task.

Touretzky's second principle is shown by the *Nixon diamond* example, in which the following is known: "Republicans are not pacifists.", "Quakers are pacifists.", "Nixon is both a Republican and a quaker."

According to our approach, he claims that a good semantics should draw no conclusion about the question whether Nixon is a $pacifist$.

The proposed solution for the problems above is based on a topological relation, called *inferential distance ordering*, stating that an individual $A$ is "nearer"to $B$ than to $C$ iff $A$ has an inference path $through$ $B$ to $C$. If $A$ is "nearer"to $B$ than to $C$, then as far as $A$ is concerned, information coming from $B$ must be preferred w.r.t. information coming from $C$. Therefore, since Clyde is "nearer"to royal elephant than to elephant, he states that Clyde is not gray. On the contrary no conclusion is taken on Nixon, as there is not any relationship between $quaker$ and $republican$.

The semantics of $\mathrm{DLP}^<$ fully agrees with the intuition underlying the *inferential distance ordering*.

**Example 9** Let us represent the Royal Elephant example in our framework:

| | |
|---|---|
| $elephant$ | $\{gray \leftarrow\}$ |
| $royal_elephant : elephant$ | $\{\neg gray \leftarrow\}$ |
| $clyde : elephant,\ royal_elephant$ | $\{\ \}$ |

The only answer set of the above $\mathrm{DLP}^<$ program is $\{\neg gray\}$.

The Nixon Diamond example can be expressed in our language as follows:

| | |
|---|---|
| $republican$ | $\{pacifist \leftarrow\}$ |
| $quaker$ | $\{\neg pacifist \leftarrow\}$ |
| $nixon : republican, quaker$ | $\{\ \}$ |

This $\mathrm{DLP}^<$ program has no answer set, and therefore no conclusion is drawn.

### 6.5 Updates in Logic Programs

The definition of the semantics of updates in logic programs is another topic where $\mathrm{DLP}^<$ could potentially be applied. Roughly, a simple formulation of the problem is: given a ($\vee$-free) logic program $P$ and a sequence $U_1, \cdots, U_n$ of successive updates (insertion/deletion of ground atoms), determine what is or is not true in the end. Expressing the insertion (deletion) of an atom $A$ by the rule $A \leftarrow (\neg A \leftarrow)$, we can represent this problem by a $\mathrm{DLP}^<$ knowledge base $\{\langle t_0, P\rangle, \langle t_1, \{U_1\}\rangle, \cdots, \langle t_n, \{U_n\}\rangle\}$ ($t_i$ intuitively represents the instant of time when the update $U_i$ has been executed), where $t_n < \cdots < t_0$. The answer sets of the program for $t_k$ can be taken as the semantics of the execution of $U_1, \cdots, U_k$ on $P$. For instance, given the logic program $P = \{a \leftarrow b, not\ c\}$ and the updates $U_1 = \{b \leftarrow\}$, $U_2 = \{c \leftarrow\}$, $U_3 = \{\neg b \leftarrow\}$, we build the $\mathrm{DLP}^<$ program

$t_0$ $\quad \{\ a \leftarrow b, c, not\ d\ \}$
$t_1 : t_0$ $\quad \{\ b \leftarrow\ \}$
$t_2 : t_1$ $\quad \{\ c \leftarrow\ \}$
$t_3 : t_2$ $\quad \{\ \neg b \leftarrow\ \}.$

The answer set $\{a, b, c\}$ of the program for $t_2$ gives the semantics of the execution of $U_1$ and $U_2$ on $P$; while the answer set $\{c\}$ of the program for $t_3$ expresses the semantics of the execution of $U_1$, $U_2$ and $U_3$ on $P$ in the given order.

The semantics of updates obtained in this way is very similar to the approach adopted for the ULL language in [21]. Further investigations are needed on this topic to see whether $\mathrm{DLP}^<$ can represent update problems in more general settings like those treated in [26] and in [4].

## 7 Implementation Issues

### 7.1 From $\mathrm{DLP}^<$ to Plain DLP

In this section we show how a $\mathrm{DLP}^<$ program can be translated into an equivalent plain disjunctive logic program (with strong negation, but without inheritance). The translation allows us to exploit existing disjunctive logic programming systems for the implementation of $\mathrm{DLP}^<$.

**Notation.**

1. We denote a literal by $\phi(\bar{X})$, where $\bar{X}$ is the tuple of literals' arguments, and $\phi$ represents an *adorned predicate*, that is either a predicate symbol $p$ or a strongly negated predicated symbol $\neg p$. Two adorned predicates are *complementary* if one is the negation of the other (e.g., $q$ and $\neg q$ are complementary). $\neg.\phi$ denotes the complementary adorned predicate of the adorned predicate $\phi$.

2. An adorned predicate $\phi$ is *conflicting* if: (a) $\phi(\bar{X})$ occurs in the head of some rule in $\mathcal{P}$ and, (b) $\neg.\phi(\bar{Y})$ occurs in the head of some rule in $\mathcal{P}$.

3. Given an object $o$ in $\mathcal{P}$, and a head literal $\phi(\bar{X})$ of a rule in $\Sigma(o)$, we say that $\phi$ is *defeasible in $o$* if a literal $\neg.\phi(\bar{Y})$ occurs in the head of a rule in $\Sigma(o')$ where $o' < o$. A rule $r$ in $\Sigma(o)$ is *defeasible in $o$* if all its head literals are defeasible in $o$.

The rewriting algorithm translating $\mathrm{DLP}^<$ programs in plain disjunctive logic programs with constraints[8] is shown in Figure 2.

An informal description of how the algorithm proceeds is the following:

[8] We use constraints for clarity, but they can be eliminated by rewriting $\leftarrow B$ to $p \leftarrow B, not\ p$.

**ALGORITHM**
**INPUT**: a DLP$^<$-program $\mathcal{P}$
**OUTPUT**: a plain disjunctive logic program with constraint $DLP(\mathcal{P})$

1: $DLP(\mathcal{P}) \Leftarrow \{prec'(o, o_1) \leftarrow \mid o < o_1\}$
2: **for** each object $o \in \mathcal{O}(\mathcal{P})$ **do**
3: **for** each defeasible adorned predicate $\phi$ in $o$ **do**
4: Add the following rule to $DLP(\mathcal{P})$:
5: $ovr'(\phi, o, X_1, \cdots, X_n) \leftarrow \neg.\phi'(X, X_1, \cdots, X_n), prec'(X, o)$
6: where $n$ is the arity of $\phi$ and $X_1, \cdots, X_n$ are distinct variables.
7: **end for**
8: **for** each rule $r$ in to $\Sigma(o)$, say $\phi_1(\bar{X}_1) \vee \cdots \vee \phi_n(\bar{X}_n) \leftarrow BODY$, **do**
9: **if** $r$ is defeasible **then**
10: Add the following two rules to $DLP(\mathcal{P})$:
11: $\phi'_1(o, \bar{X}_1) \vee \cdots \vee \phi'_n(o, \bar{X}_n) \leftarrow BODY,\ not\ ovr'(r, o, \bar{X}_1, ..., \bar{X}_n)$
12: $ovr'(r, o, \bar{X}_1, ..., \bar{X}_n) \leftarrow ovr'(\phi_1, o, \bar{X}_1), ..., ovr'(\phi_n, o, \bar{X}_n)$
13: **else**
14: Add the following rule to $DLP(\mathcal{P})$:
15: $\phi'_1(o, \bar{X}_1) \vee \cdots \vee \phi'_n(o, \bar{X}_n) \leftarrow BODY$
16: **end if**
17: **end for**
18: **end for**
19: **for** each adorned predicate $\phi$ appearing in $\mathcal{P}$ **do**
20: Add the following rule to $DLP(\mathcal{P})$:
21: $\phi(X_1, \cdots, X_n) \leftarrow \phi'(X_0, X_1, \cdots, X_n)$
22: where $n$ is the arity of $\phi$ and $X_0, \cdots, X_n$ are distinct variables.
23: **end for**
24: **for** each conflicting adorned predicate $\phi$ appearing in $\mathcal{P}$ **do**
25: Add the following constraint to $DLP(\mathcal{P})$:
26: $\leftarrow \phi(X_1, \cdots, X_n), \neg.\phi(X_1, \cdots, X_n)$
27: where $n$ is the arity of $\phi$ and $X_1, \cdots, X_n$ are distinct variables.
28: **end for**

Figure 2: A Rewriting Algorithm

- $DLP(\mathcal{P})$ is initialized to a set of facts with head predicate $prec'$ representing the partial ordering among objects (statement 1).
- Then, for each object $o$ in $\mathcal{O}(\mathcal{P})$:
  - For each defeasible literal $\phi(\bar{X})$ appearing in $o$, rules defining when the literal is overridden are added (statements 3–5).
  - For each rule $r$ belonging to $o$:
    1. If $r$ is defeasible, then the rule is rewritten, such that the head literals include information about the object in which they have been derived, and the body includes a literal which makes the rule not applicable if it is overridden. In addition, a rule is added which encodes when the rule is overridden (statement 8).
    2. Otherwise (i.e., $r$ is not defeasible) just the rule head is rewritten as described above, since these rules cannot be overridden (statement 10).
- For all adorned predicates in the program, we add a rule which states that an atom with this predicate holds, no matter in which object it has been derived (statements 14–16). The information in which object an atom has been derived is only needed for determination of overriding.

We now give two examples to show how the translation works:

**Example 10** The datalog version $DLP(\mathcal{P})$ of the program $\mathcal{P}$ of Example 1 is:

**1)** **rules expliciting partial order among objects** :
$prec'(o_2, o_1) \leftarrow$

**2)** **rules for defeasible adorned predicates in** $o_1$ :
$ovr'(a, o_1) \leftarrow \neg a'(X), prec'(X, o_1)$
$ovr'(\neg b, o_1) \leftarrow b'(X), prec'(X, o_1)$

**3)** **rewriting of rules in** $o_1$ :
$a'(o_1) \vee \neg b'(o_1) \leftarrow c, not\ d, not\ ovr'(r_1, o_1)$
$ovr'(r_1, o_1) \leftarrow ovr'(a, o_1), ovr'(\neg b, o_1)$

**4)** **rewriting of rules in** $o_2$ :
$\neg a'(o_2) \vee c'(o_2) \leftarrow$
$b'(o_2) \leftarrow$
$c'(o_2) \leftarrow b$

**5)** **projection rules** :
$a \leftarrow a'(X) \quad \neg a \leftarrow \neg a'(X)$
$b \leftarrow b'(X) \quad \neg b \leftarrow \neg b'(X)$
$c \leftarrow c'(X) \quad d \leftarrow d'(X)$

∎

Given a model $M$ for $DLP(\mathcal{P})$, $\pi(M)$ is the set of literals obtained from $M$ by eliminating all the literals with a "primed" predicate symbol, i.e. a predicate symbol in the set $\{prec', ovr'\} \cup \{\phi' | \exists\, an\ adorned\ literal\ \phi(\bar{X})\ appearing\ in\ \mathcal{P}\}$. $\pi(M)$ is the set of literals without all atoms which were introduced by the translation algorithm.

The DLP version of a DLP$^<$-program $\mathcal{P}$ can be used in place of $\mathcal{P}$ in order to evaluate answer sets of $\mathcal{P}$. The result supporting the above statement is the following:

**Theorem 4** *For each answer set $M$ for $\mathcal{P}$ there exists a stable model $M'$ for $DLP(\mathcal{P})$ such that $\pi(M') = M$. Moreover, for each stable model $M'$ for $DLP(\mathcal{P})$ there exists an answer set $M$ for $\mathcal{P}$ such that $\pi(M') = M$.*

**Proof.** First we show that given an answer set $M$ for $\mathcal{P}$ there exists a stable model $M'$ for $DLP(\mathcal{P})$ such that $\pi(M') = M$. According to Definition 4, $M$ is a minimal model for $pos(G_M(\mathcal{P}))$. We proceed by constructing the model $M'$. Let $K_1 = \{prec'(o, o_1) \mid o < o_1\}$. Let $K_2$ be the set of ground literals $ovr'(r, o, \bar{X})$ such that there exists a rule $r \in ground(\mathcal{P})$ with $obj_of(r) = o$ such that $r$ is overridden in $M$ and $\bar{X}$ is the tuple of arguments appearing in the head of $r$. Let $K_3$ be the set of ground literals $ovr'(L, o)$ such that there exist two rules $r, r' \in ground(\mathcal{P})$ such that $L \in Head(r)$ and $r'$ overrides $r$ in $L$. Let denote by $\mathcal{K}$ the collection of sets of ground literals such that each element $K \in \mathcal{K}$ satisfies the following properties:

1. for each literal $\phi(\bar{X}) \in M$ there is a literal $\phi'(o, \bar{X})$ in $K$, for some object identifier $o$,
2. for each $r \in G_M(\mathcal{P})$ such that the body of $r$ is true in $M$, for at least one literal $\phi(\bar{X})$ of the head of $r$ a correspondig literal $\phi'(obi_of(r), \bar{X})$ occurs in $K$.

Let $M_K = K_1 \cup K_2 \cup K_3 \cup K \cup M$. It is easy to show that $G_{M_K}(DLP(\mathcal{P}))$ is indipendent on wich $K \in \mathcal{K}$ is choosen. Indeed, no literals from $K$ appear in the NAF part of the rules in $ground(DLP(\mathcal{P}))$.

Let see now which rules the program $G_{M_K}(DLP(\mathcal{P}))$ contain (for any set $K \in \mathcal{K}$).

Both the rules with head predicate $prec'$ and $ovr'$ and the rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ (added by the statement 15 of the algorithm) and the costraints appear unchanged in $G_{M_K}(DLP(\mathcal{P}))$. Indeed, these rules and constraints do not contain $NAF$ part (recall that the GL tranformation can modify only rules where a NAF part occur). The other rules of $G_{M_K}(DLP(\mathcal{P}))$ are those originating from rules of $ground(DLP(\mathcal{P}))$ obtained by rewriting rules of $ground(\mathcal{P})$ (see statements 8 and 10 of the algorithm). Consider a rule $r$ of $ground(\mathcal{P})$. If $r$ is overridden in $M$ then the corresponding rule in $ground(DLP(\mathcal{P}))$ contains a $NAF$ part not satisfied in $M'$, by construction of $K_2$ and $K_3$. Hence, such a rule appears neither in $G_M(\mathcal{P})$ nor in $G_{M_K}(DLP(\mathcal{P}))$. Suppose now $r$ is not overridden in $M$. In case $Body^-(r) \cap M \neq \emptyset$, the corresponding rule of $DLP(\mathcal{P})$ is eliminated in the GL transformation. Indeed, $M \subseteq M_K$, for each $K \in \mathcal{K}$. Hence, $r$ appears neither in $G_M(\mathcal{P})$ nor in $G_{M_K}(DLP(\mathcal{P}))$. On the other hand, if $r$ is not overridden in $M$ and $Body^-(r) \cap M = \emptyset$, it appears in $G_{M_K}(DLP(\mathcal{P}))$ with the same body of the corresponding rule in $G_M(\mathcal{P})$ and the head modified by renaming predicates (from $\phi$ to $\phi'$) and by adding the object $o$ as first argument in each head literal. Thus, informally, every rule in $G_M(\mathcal{P})$ has a corresponding rule in $G_{M_K}(DLP(\mathcal{P}))$ with the same body and a rewritten head.

Now we prove that, for any $K \in \mathcal{K}$, $M_K$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, rules with head predicate $prec'$ are clearly satisfied. Further, rules with head predicate $ovr'$ are satisfied by construction of set $K_2$, $K_3$ and $K$. Moreover, rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ are satisfied since $M \subseteq M_K$ and by construction of $K$. Consider now the remaining rules (those corresponding to rules of $G_M(\mathcal{P})$). Let $r$ be one of these rules. As shown earlier, it corresponds to a rule $r'$ of $G_M(\mathcal{P})$ such that $Body(r) = Body(r')$. Clearly the body of $r$ is true in $M_K$ if and only if the body of $r'$ is true in $M$. Thus, the only case we have to consider is that of the body of $r'$ true in $M$. In this case, it easy to see that also the head of $r$ is satisfied in $M_K$, by construction of the set $K$ (see point 2 above characterizing the collection $\mathcal{K}$). Thus, $M_K$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$.

Moreover we prove that each model for $pos(G_{M_K}(DLP(\mathcal{P})))$

1. contains $M$, and

2. belongs to $\mathcal{K}$

To prove point 1, suppose by contradiction $\bar{M}$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ such that $M \cap \bar{M} \neq M$. Since $\bar{M}$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ also rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ must be satisfied in $\bar{M}$. Thus, for each literal $\phi(\bar{X})$ of $M$ not belonging to $\bar{M}$, no corresponding literal $\phi'(X, \bar{X})$ can occur in $\bar{M}$. Since, as shown earlier, rules of $G_{M_K}(DLP(\mathcal{P}))$ correspond to rule of $G_M(\mathcal{P})$, it is easy to see that this implies that $\pi(\bar{M})$ is also a model for $G_M(\mathcal{P})$. But, since $\pi(\bar{M}) \subset M$, a contradiction arises.

Now we have to prove the statement 2 above. By contradiction suppose $\bar{M} \notin \mathcal{K}$. Thus, $\bar{M}$ does not satisfy one of two properties characterizing the family $\mathcal{K}$. First suppose the property 1 is not satisfied by $\bar{M}$, that is, there is a literal $\phi(\bar{X})$ in $M$ such that no corresponding literal $\phi'(o, \bar{X})$ occur in $\bar{M}$, for some object identifier $o$. Let $\bar{M}'$ the set obtained by $\bar{M}$ by eliminating all such literals $\phi(\bar{X})$. It is easy to see that $\bar{M}'$ is still a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ are satisfied since literals $\phi(\bar{X})$ dropped from $\bar{M}$ do not have corresponding "$\phi'$ literals"by hypothesis. Further, no other rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ contains literal from $M$ in the head. Thus, similarly to the previuos case, due to the correspondence between rules of $G_M(\mathcal{P})$ and rules of $G_{M_K}(DLP(\mathcal{P}))$, $\pi(\bar{M}')$ is a model for $G_M(\mathcal{P})$. But this is a contradiction, since $\pi(\bar{M}') \subset M$.

Suppose now the property 2 is not satisfied by $\bar{M}$, that is, there is a rule $r \in G_M(\mathcal{P})$ such that the body of $r$ is true in $M$ and no "$\phi'$ literals"corresponding to literals of the head occur in $\bar{M}$. Since, as we have previously proven, $M \subseteq \bar{M}$, the corresponding rule of $G_{M_K}(DLP(\mathcal{P}))$ is not satisfied. This implies that $\bar{M}$ is not a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ (contradiction).

As a consequence, it is easy to realize that each model for $pos(G_{M_K}(DLP(\mathcal{P})))$ must contain both sets $K_1$, $K_2$ and $K_3$.

Thus, the model $M' = M_{K'}$ for $K' \in \mathcal{K}$ such that there not exits a set $\bar{K} \in \mathcal{K}$ such that $\bar{K} \subset K'$ is a minimal model for $pos(G_{M_K}(DLP(\mathcal{P}))$, that is, a stable model for $DLP(\mathcal{P})$. Hence, the proof is concluded, since $\pi(M') = M$.

Now, we prove that given a stable model $M'$ for $DLP(\mathcal{P})$ $\pi(M')$ is an answer set for $\mathcal{P}$.

Let $r'$ be a rule of $pos(G_{M'}(DLP(\mathcal{P}))$ containing literals $\phi'(\bar{X})$ such that $\phi$ is an (adorned) predicate appearing in $\mathcal{P}$. It easy to see that $r'$ has a corresponding rule $r$ in $pos(G_M(\mathcal{P}))$ such that both $Body(r) = Body(r')$ and the head of $r'$ can be obtained by replacing each literal $\phi'(X_0, X_1, ..., X_n)$ by the literal $\phi(X_1, ..., X_n)$. Indeed, from the precence of rules added by statement 15 of the algorithm, $\phi'(X_0, X_1, ..., X_n) \in M'$ if and only if $\phi(X_1, ..., X_n) \in M$. Further, a literal $ovr'(r, o, \bar{X})$ belongs to $M'$ if and only if the rule $r$ of $\mathcal{P}$ is overridden in $M$. In a similar way it can be shown that each rule $r$ of $pos(G_M(\mathcal{P}))$ corresponds to a rule $r'$ of $pos(G_{M'}(DLP(\mathcal{P}))$ (i.e., bodies of rules coincide and the head of $r$ can be obtained by the head of $r'$ by eliminating the first argument from each literal appearing in it and by renaming all predicates from $\phi'$ in $\phi$). From this correspondence between $pos(G_M(\mathcal{P}))$ and $pos(G_{M'}(DLP(\mathcal{P}))$, it clearly follows that $M$ is a model for $pos(G_M(\mathcal{P}))$ and it is minimal (otherwise, $M'$ would not be minimal for $pos(G_{M'}(DLP(\mathcal{P})))$. Thus, $M$ is an answer set for $\mathcal{P}$. ∎

**Example 11** Consider the program $\mathcal{P}$ of Example 1. It is easy to see that $DLP(\mathcal{P})$ (see Example 10) admit one answer set $M = \{prec'(o_2, o_1), a'(o_1), b'(o_2), c'(o_2), ovr'(\neg b, o_1), a, b, c\}$. Thus, $\pi(M) = \{a, b, c\}$. On the other hand, $\pi(M)$ is the only answer set for $\mathcal{P}$, as shown in Example 4. ∎

## 7.2 Prototype Architecture

We have used the `dlv` system [12] to implement a prototype system for DLP$^<$. A schematic visualisation of its architecture is shown in Figure 3.

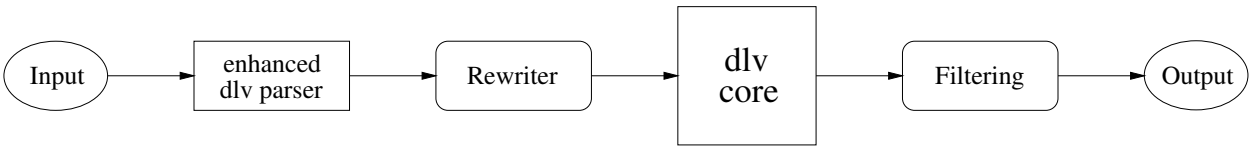


Figure 3: Flow diagram of the prototype.

First of all, we have extended the `dlv` parser to incorporate the $\mathrm{DLP}^<$ syntax. In this way, all the advanced features of `dlv` (e.g. bounded integer arithmetics, comparison builtins, etc.) are also available with $\mathrm{DLP}^<$. The *Rewriter* module implements the translation depicted in Figure 2. Once the rewritten version $\pi(\mathcal{P})$ of $\mathcal{P}$ is generated, its answer sets are then computed using the dlv core. Before the output is shown to the user, $\pi$ is applied to each answer set in order to strip the internal predicates from the output.

On the webpage [14] the prototype is described in detail, and binary versions are available for different platforms.

## Acknowledgements

The idea of representing inertia in planning problems in a higher object is due to Axel Polleres. This work was partially supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

## References

[1] Alferes, J., Pereira, L.M., On Logic Program Semantics with Two Kinds of Negation *Proc. of JICSLP 1992*, pp. 574-588

[2] Alferes, J., Pereira, L.M., Przymusinski, T.C., Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming, *Proc. of JELIA 1996*, pp. 143-163

[3] Alferes, J., Pereira, L.M., Przymusinski, T.C., 'Classical' Negation in Nonmonotonic Reasoning and Logic Programming, *Journal of Automated Reasoning* 20(1), 1998, pp. 107-142

[4] Alferes, J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C., Dynamic Logic Programming, *Proc. KR'98*, Morgan Kaufmann, 1998.

[5] Baral, C., Gelfond, M. Logic Programming and Knowledge Representation *Journal of Logic Programming*, Vol. 19/20, May/July 1994, pp. 73–148.

[6] Brewka, G., Eiter, T., Preferred Answer Sets for Extended Logic Programs, *Proc. KR'98*, Morgan Kaufmann, 1998.

[7] Buccafurri, F., Leone, N., Rullo, P., Disjunctive Ordered Logic: Semantics and Expressiveness, *Proc. KR'98*, Morgan Kaufmann, 1998.

[8] P. M. Dung. Representing Actions in Logic Programming and its Applications in Database Updates. In *Proc. $10^{th}$ ICLP*, pp. 222–238, 1993.

[9] Eiter, T., Gottlob, G., On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 1995.

[10] Eiter, T., Gottlob, G., and Mannila, H., Disjunctive Datalog, *ACM Transactions on Database Systems*, 22(3), September, 1997.

[11] Eiter, T., Gottlob, G., Leone, N., Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science* 189(1-2), pp. 129–177, 1997.

[12] Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., The Knowledge Representation System `dlv`: Progress Report, Comparisons, and Benchmarks, *Proc. KR'98*, Morgan Kaufmann, 1998.

[13] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft, 1999.

[14] Faber, W., The DLP$^<$ Homepage: `www.dbai.tuwien.ac.at/proj/dlv/inheritance/`

[15] Gelfond, M., and Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365-385, 1991.

[16] M. Gelfond and V. Lifschitz. Representing Action and Change by Logic Programs. *JLP*, 17:301–321, 1993.

[17] Gelfond, M. and Son, T.C., Reasoning with Prioritized Defaults, *Proc. of LPKR'97*, Port Jefferson, New York, October 1997.

[18] E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proc. AAAI '98*, pp. 623–630, 1998.

[19] Hanks, S., McDermott, D., Nonmonotonic Logic and Temporal Projection, *Artificial Intelligence*, 33:379–412, 1987.

[20] Kowalski, R.A., Sadri, F., Logic Programs with Exceptions, *Proc. of 7th ICLP*, Jerusalem, 1990, pp. 598-616.

[21] Leone, N., Palopoli, L., Romeo, M., A Language for Updating Logic Programs, *Journal of Logic Programming*, 23(1), 1995, pp. 1–61.

[22] V. Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczinski, and D. S. Warren, eds., *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999, to appear.

[23] Lobo, J., Minker, J., and Rajasekar, A., *Foundations of Disjunctive Logic Programming*. MIT Press, Cambridge, MA, 1992.

[24] Marek, W., Truszczyński, M., Modal logic for default reasoning, *Annals of Mathematics and Artificial Intelligence*, 1, 1990, pp. 275-302.

[25] Marek, W., Truszczyński, M., Autoepistemic Logic, *Journal of the ACM*, 38, 3, 1991, pp. 518-619.

[26] Marek, W., Truszczyński, M., Revisions Specifications by means of Programs, *Proc. JELIA'94*, Sringer, LNAI 838, 1994, pp. 122–136.

[27] McCarthy, J., Hayes, P. J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, pp. 463–502, 1969.

[28] Nute, D., Defeasible Logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume III, pages 353–395. Clarendon Press, Oxford, 1994.

[29] Pradhan, S., and Minker, J., Combining Datalog Databases Using Priorities. *International Journal of Cooperative Intelligent Systems*, 1996.

[30] Reiter, R., A Logic for Default Reasoning, *Artificial Intelligence*, vol.13, pp. 81-132, 1980.

[31] Sakama, C., Inoue, K., Representing priorities in logic programs, *Proc. JICSLP'96*, MIT Press, 1996, pp. 82–96.

[32] Shanahan, M., *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, 1997.

[33] Touretzky, D.S., *The Mathematics of Inheritance Systems*, Pitman London, 1986.