Disjunctive Logic Programs with Inheritance

FRANCESCO BUCCAFURRI
DIMET – Università di Reggio Calabria 89100, Reggio Calabria, Italia
(*e-mail:* bucca@ns.ing.unirc.it)

WOLFGANG FABER
Institut für Informationssysteme, Technische Universität Wien, 1040 Vienna, Austria
(*e-mail:* faber@kr.tuwien.ac.at)

NICOLA LEONE
Dipartimento di Matematica, Università degli Studi della Calabria, 87030 Rende (CS), Italia
(*e-mail:* leone@unical.it)

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 and has subsequently been integrated into **DLV**.

1 Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and negation as failure (NAF) may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning (Baral and Gelfond 1994, Lobo, Minker and Rajasekar 1992, Gelfond and Lifschitz 1991). One of the attractions of disjunctive logic programming is its ability to naturally model incomplete knowledge (Baral and Gelfond 1994, Lobo et al. 1992). The need to differentiate between atoms which are false because of the failure to prove them true (NAF, or CWA negation) and atoms the falsity of which is explicitly provable led to extend disjunctive logic programs by strong negation (Gelfond and Lifschitz 1991). Strong negation, permitted also in the heads of rules, further enhances the knowledge modeling features of the language, and its usefulness is widely acknowledged in the literature (Alferes and Pereira 1992, Baral and Gelfond 1994, Kowalski and Sadri 1990, Alferes, Pereira and Przymusinski 1996, Sakama and Inoue 1996, Alferes, Pereira and Przymusinski 1998b). 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 at least be equipped with an extra negative literal, which "blocks" inferences from r for abnormal instances (Gelfond and Son 1997). For instance, to encode the famous nonmonotonic reasoning (NMR) example stating that birds *normally* fly while penguins do not fly, one should write[1] the rule

$$\mathtt{fly(X)} \leftarrow \mathtt{bird(X)},\ \mathtt{not}\ \neg\mathtt{fly(X)}.$$

along with the fact

$$\neg\mathtt{fly(penguin)}.$$

This paper proposes an extension of disjunctive logic programming by inheritance, called DLP$^<$. The addition of inheritance enhances the knowledge modeling features of the language. Possible conflicts are solved in favor 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 contributions of the paper are the following:

- We formally define the DLP$^<$ language, providing a declarative model theoretic semantics of DLP$^<$, which is shown to generalize the Answer Set Semantics of (Gelfond and Lifschitz 1991).
- We illustrate the knowledge modeling features of the language by encoding classical nonmonotonic problems in DLP$^<$. Interestingly, DLP$^<$ also supplies a very natural representation of frame axioms.
- We analyze the computational complexity of reasoning over DLP$^<$ programs. Importantly, while inheritance enhances the knowledge modeling ability of disjunctive logic programming, it does not cause any computational overhead, as reasoning in DLP$^<$ has exactly the same complexity as reasoning in disjunctive logic programming.
- We compare DLP$^<$ to related work proposed in the literature. In particular, we stress the differences between DLP$^<$ and Disjunctive Ordered Logic ($\mathcal{DOL}$) (Buccafurri, Leone and Rullo 1998, Buccafurri, Leone and Rullo 1999); we point out the relation to the Answer Set Semantics of (Gelfond and Lifschitz 1991); we compare DLP$^<$ with prioritized disjunctive logic programs (Sakama and Inoue 1996); we analyze its relationships to inheritance networks (Touretzky 1986) and we discuss the possible application of DLP$^<$ to give a formal semantics to updates of logic programs. (Alferes, Leite, Pereira, Przymusinska and Przymusinski 1998a, Marek and Truszczyński 1994, Leone, Palopoli and Romeo 1995).
- We implement a DLP$^<$ system. To this end, we first design an efficient translation from DLP$^<$ to plain disjunctive logic programming. Then, using this translation, we implement a DLP$^<$ evaluator on top of the **DLV** system (Eiter, Leone, Mateis,

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

Pfeifer and Scarcello 1998). It is part of **DLV** and can be freely retrieved from (Faber 1999).

The sequel of the paper is organized as follows. The next two sections provide a formal definition of DLP$^<$; in particular, its syntax is given in Section 2 and its semantics is defined in Section 3. Section 4 shows the use of DLP$^<$ for knowledge representation and reasoning, providing a number of sample DLP$^<$ encodings. Section 5 analyzes the computational complexity of the main reasoning tasks arising in the framework of DLP$^<$. Section 6 discusses related work. The main issues underlying the implementation of our DLP$^<$ system are tackled in Section 7, and our conclusions are drawn in Section 8.

2 Syntax of DLP$^<$

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

Let the following disjoint sets be given: a set $\mathcal{V}$ of *variables*, a set Π of *predicates*, a set Λ 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: (1) irreflexive – $c \not< c \;\; \forall c \in \mathcal{O}$, and (2) transitive – $a < b \wedge b < c \Rightarrow a < c \;\; \forall a, b, c \in \mathcal{O}$).

A $term$ is either a constant in Λ or a variable in $\mathcal{V}$.[2]

An $atom$ is a construct of the form $a(t_1, ..., t_n)$, where a is a $predicate$ of arity n in Π 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

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are literals, `not` is the *negation as failure* symbol and $\bigotimes$ is either (1) the symbol '.' or (2) the symbol '!'. In case (1) r is a *defeasible rule*, in case (2) it is a *strict rule*.

The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of r. $b_1, ..., b_k$ is called the *positive part* of the body of r and $\texttt{not}\ b_{k+1}, ..., \texttt{not}\ b_m$ 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, and in the NAF part of the body of a rule r by $Head(r)$, $Body^+(r)$, and $Body^-(r)$, respectively.

If the body of a rule r is empty, then r is called *fact*. The symbol '$\leftarrow$' is usually omitted from facts.

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 DLP$^<$ *program for* o (*on* $\mathcal{K}$) is the set of objects $\mathcal{P} = \{(o', \Sigma(o')) \in \mathcal{K} \mid o = o' \textit{ or } o < o'\}$.

[2] Note that function symbols are not considered in this paper.

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 an 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 explicitly 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.[3] An object o is denoted as $oid(o) : o_1, \ldots, o_n\ \Sigma(o)^4$, where $(oid(o), o_1)$, …, $(oid(o), o_n)$ are exactly those pairs of the transitive reduction of $<$, in which the first object identifier is $oid(o)$. o is referred to as *sub-object* of $o_1, \ldots, o_n$.

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

$$\begin{array}{ll} \mathtt{o}_1 & \{\, \mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}. \quad \mathtt{e} \leftarrow \mathtt{b!} \,\} \\ \mathtt{o}_2 : \mathtt{o}_1 & \{\, \mathtt{b}. \quad \neg\mathtt{a} \vee \mathtt{c}. \quad \mathtt{c} \leftarrow \mathtt{b}. \,\} \end{array}$$

$\mathcal{P}$ consists of two objects $\mathtt{o}_1$ and $\mathtt{o}_2$. $\mathtt{o}_2$ is a sub-object of $\mathtt{o}_1$. 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 $\mathtt{o}_2$ (the bottom object). ∎

3 Semantics of 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 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 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 σ 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 instances 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.

[3] (a, b) is in the transitive reduction of $<$ iff $a < b$ and there is no c such that $a < c$ and $c < b$.
[4] The set $\Sigma(o)$ is denoted without commas as separators.

A subset of ground literals in $B_{\mathcal{P}}$ is said to be *consistent* if it does not contain a pair of complementary literals. 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: (1) every literal in $Body^+(r)$ is true w.r.t. I, and (2) every literal in $Body^-(r)$ is false w.r.t. I. A 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.

Next we introduce the concept of a *model* for a DLP$^<$-program. Different from 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.

Given two ground rules r_1 and r_2 we say that r_1 *threatens* r_2 *on* a literal L if (1) $\neg.L \in Head(r_1)$ and $L \in Head(r_2)$, (2) $obj\_of(r_1) < obj\_of(r_2)$ and (3) r_2 is defeasible,

Definition 1
Given an interpretation I and two ground rules r_1 and r_2 such that r_1 threatens r_2 on L we say that r_1 overrides *r_2* on *L* in *I if: (1) $\neg.L \in I$, and (2) the body of r_2 is true in I.*

A (defeasible) 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 and r' is not strict, then r' is overridden, meaning that a should be preferred to $\neg a$ because it is derivable from a more trustable rule.

Observe that, by definition of overriding, strict rules cannot be overridden, since they are never threatened.

Example 2
Consider the program $\mathcal{P}$ of Example 1. Let $\mathtt{I} = \{\neg\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$ be an interpretation. Rule $\neg\mathtt{a} \vee \mathtt{c} \leftarrow .$ in the object $\mathtt{o}_2$ overrides rule $\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}, \mathtt{not}\ \mathtt{d}.$ in $\mathtt{o}_1$ on the literal $\mathtt{a}$ in $\mathtt{I}$. Moreover, rule $\mathtt{b} \leftarrow .$ in $\mathtt{o}_2$ overrides rule $\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}, \mathtt{not}\ \mathtt{d}.$ in $\mathtt{o}_1$ on the literal $\neg\mathtt{b}$ in $\mathtt{I}$. Thus, the rule $\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}, \mathtt{not}\ \mathtt{d}.$ in $\mathtt{o}_1$ is overridden in $\mathtt{I}$. ∎

Example 3
Consider the following program $\mathcal{P}$:

$$\begin{array}{ll} \mathtt{o}_1 & \{\ \neg\mathtt{a!} \quad \neg\mathtt{b.}\ \} \\ \mathtt{o}_2 : \mathtt{o}_1 & \{\ \mathtt{a} \leftarrow \mathtt{not}\ \mathtt{b.} \quad \mathtt{b} \leftarrow \mathtt{not}\ \mathtt{a.}\ \} \end{array}$$

Consider now the interpretations $\mathtt{M}_1 = \{\mathtt{a}, \neg\mathtt{b}\}$ and $\mathtt{M}_2 = \{\mathtt{b}, \neg\mathtt{a}\}$. While the rule $\neg\mathtt{b}.$ is overridden in $\mathtt{M}_2$, the rule $\neg\mathtt{a}!$ cannot be overridden since it is a strict rule. Due to overriding, strict rules and defeasible rules are quite different from the semantic point of view. In our example, the overriding mechanism allows us to invalidate the defeasible rule $\neg\mathtt{b}.$ in favor of the more trustable one $\mathtt{b} \leftarrow \mathtt{not}\ \mathtt{a}.$ (w.r.t. the interpretation $\mathtt{M}_2$). In words, the defeasible rule is invalidated in $\mathtt{M}_2$ because of a more specific contradictory rule and no inconsistency is generated. In other words, it is possible to find an interpretation containing the literal $\mathtt{b}$ (i.e., stating an exception for the rule $\neg\mathtt{b}.$) such that all the rules of $\mathcal{P}$ are either satisfied or overridden (i.e., invalidated) in it. Such an interpretation is just $\mathtt{M}_2$. This cannot happen for the strict rule $\neg\mathtt{a}!$. Indeed, no interpretation containing the literal $\mathtt{a}$ (i.e., stating an exception for the strict rule) can be found which satisfies all non-overridden rules of $\mathcal{P}$. ∎

In the example above we have implicitly used the notion of *model* for a program $\mathcal{P}$ that we next formally provide. A model for a program is an interpretation satisfying all its non overridden rules.

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 $\mathcal{P}$.* ∎

Note that strict rules must be satisfied in every model, since they cannot be overridden.

Example 4
It is easy to see that $\mathtt{M}_1 = \{\mathtt{a}, \neg\mathtt{b}\}$ is not a model for the program $\mathcal{P}$ of Example 3 since the rule $\neg\mathtt{a}!$ is neither overridden nor satisfied. On the contrary, $\mathtt{M}_2 = \{\mathtt{b}, \neg\mathtt{a}\}$ is a model for $\mathcal{P}$, since $\neg\mathtt{b}.$ is overridden by the rule $\mathtt{b} \leftarrow \mathtt{not}\ \mathtt{a}$. The latter rule is satisfied since $\mathtt{b} \in \mathtt{M}_2$. The rule $\neg\mathtt{a}!$ is satisfied as $\neg\mathtt{a} \in \mathtt{M}_2$ and the rule $\mathtt{a} \leftarrow \mathtt{not}\ \mathtt{b}.$ is satisfied since both body and head are false w.r.t. $\mathtt{M}_2$.

Next we define the transformation G_I based on which our semantics is defined. This transformation applied to a program $\mathcal{P}$ w.r.t. an interpretation I output a set of rules $G_I(\mathcal{P})$ with no negation by failure in the body. Intuitively, such rules are those remaining from $ground(\mathcal{P})$ by (1) eliminating the rules overridden in the interpretation I, (2) deleting rules whose NAF part is not "true" in I (i.e., some literal negated by negation as failure occurs in I) and (3) deleting the NAF part of all the remainder rules. Since the transformation encodes the overriding mechanism, the distinction between strict rules and defeasible rules in $G_I(\mathcal{P})$ is meaningless (indeed, there is no difference between strict and defeasible rules except for the overriding mechanism where the upper rule is required to be defeasible). For this reason the syntax of rules in $G_I(\mathcal{P})$ can be simplified by dropping the symbol . from defeasible rules and the symbol ! from strict rules.

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 5
Consider the program $\mathcal{P}$ of Example 1. Let $\mathtt{I}$ be the interpretation $\{\neg\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$. As shown in Example 2, rule $\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}, \mathtt{not}\ \mathtt{d}.$ is overridden in $\mathtt{I}$. Thus, $G_I(\mathcal{P})$ is the set of rules $\{\neg\mathtt{a} \vee \mathtt{c}.\quad \mathtt{e} \leftarrow \mathtt{b}.\quad \mathtt{b}.\quad \mathtt{c} \leftarrow \mathtt{b}.\ \}$. Consider now the interpretation $\mathtt{M} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$. It is easy to see that $G_M(\mathcal{P}) = \{\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}.\quad \neg\mathtt{a} \vee \mathtt{c}.\quad \mathtt{e} \leftarrow \mathtt{b}.\quad \mathtt{b}.\quad \mathtt{c} \leftarrow \mathtt{b}.\}$. ∎

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 (DLP$^<$-)answer set *for $\mathcal{P}$ if M is a minimal model of the positive version $pos(G_M(\mathcal{P}))$ of $G_M(\mathcal{P})$.* ∎

Note that interpretations must be consistent by definition, so considering $pos(G_M(\mathcal{P}))$ instead of $G_M(\mathcal{P})$ does not lose information in this respect.

Note that the notion of minimal model of Definition 2 cannot be used in Definition 4, as G_M is a set of rules and not a DLP$^<$ program.

Example 6
Consider the program $\mathcal{P}$ of Example 1:

It is easy to see that the interpretation $\mathtt{I}$ of Example 5 is not an answer set for $\mathcal{P}$. Indeed, although $\mathtt{I}$ is a model for $pos(G_I(\mathcal{P}))$ it is not minimal, since the interpretation $\{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$ is a model for $pos(G_I(\mathcal{P}))$, too. Note that the interpretation $\mathtt{I}' = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$ is not an answer set for $\mathcal{P}$. Indeed, $G_{I'}(\mathcal{P}) = \{\ \mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}.\quad \neg\mathtt{a} \vee \mathtt{c}.\quad \mathtt{e} \leftarrow \mathtt{b}.\quad \mathtt{b}.\quad \mathtt{c} \leftarrow \mathtt{b}.\ \}$ and $\mathtt{I}'$ is not a model for $pos(G_{I'}(\mathcal{P}))$, since the rule $\mathtt{a} \vee \neg\mathtt{b} \leftarrow \mathtt{c}.$ is not satisfied in $\mathtt{I}'$.

On the other hand, the interpretation $\mathtt{M}$ of Example 5 is an answer set for $\mathcal{P}$, since $\mathtt{M}$ is a minimal model for $pos(G_M(\mathcal{P}))$. Moreover, it can be easily realized that $\mathtt{M}$ is the only answer set for $\mathcal{P}$.

Finally, the program $\mathcal{P}$ of Example 3 admits one consistent answer set $\mathtt{M}_2 = \{\mathtt{b}, \neg\mathtt{a}\}$. Note that if we replace the strict rule $\neg\mathtt{a}!$ by a defeasible rule $\neg\mathtt{a}.$, $\mathcal{P}$ admits two answer sets, namely $\mathtt{M}_1 = \{\mathtt{a}, \neg\mathtt{b}\}$ and $\mathtt{M}_2 = \{\mathtt{b}, \neg\mathtt{a}\}$. Asserting $\neg\mathtt{a}$ by a strict rule, prunes the answer set $\mathtt{M}_1$ stating the exception (truth of the literal $\mathtt{a}$) to this rule. ∎

It is worthwhile noting that if a rule $\mathtt{r}$ is not satisfied in a model $\mathtt{M}$, then *all* literals in the head of $\mathtt{r}$ must be overridden in $\mathtt{M}$.

Let $\mathcal{P}_1$ be the program

$$\begin{array}{ll} \mathtt{o}_3 & \{\ \mathtt{a} \vee \mathtt{b}.\quad \leftarrow \mathtt{b}.\ \} \\ \mathtt{o}_2 : \mathtt{o}_3 & \{\ \neg\mathtt{a}.\ \} \end{array}$$

and $\mathcal{P}_2$

$$\begin{array}{ll} \mathtt{o}_1 & \{\ \mathtt{a}.\quad \leftarrow \mathtt{b}.\ \} \\ \mathtt{o}_2 : \mathtt{o}_1 & \{\ \neg\mathtt{a}.\ \} \end{array}$$

Then, $\{\neg\mathtt{a}\}$ is not a model for program $\mathcal{P}_1$, because the head literal $\mathtt{b}$ in the head of $\mathtt{a} \vee \mathtt{b}.$ is not overridden in $\mathtt{M}$. If we drop $\mathtt{b}$ from rule $\mathtt{a} \vee \mathtt{b}.$, then $\{\neg\mathtt{a}\}$ is a model of the resulting program $\mathcal{P}_2$.

Observe also that two programs having the same answer sets, as $\mathtt{o}_1$ and $\mathtt{o}_3$ (both have the single answer set $\{\neg\mathtt{a}\}$), may get different answer sets even if we add the same object to both of them. Indeed, program $\mathcal{P}_1$ has no answer set, while program $\mathcal{P}_2$ has the answer set $\{\neg\mathtt{a}\}$.

This is not surprising, as a similar phenomenon also arises in normal logic programming where $\mathtt{P_1} = \{\mathtt{a.}\}$ and $\mathtt{P_2} = \{\mathtt{a} \leftarrow \mathtt{not b.}\}$ have the same answer set $\{\mathtt{a}\}$, while $\mathtt{P_1} \cup \{\mathtt{b.}\}$ and $\mathtt{P_2} \cup \{\mathtt{b.}\}$ have different answer sets ($\{\mathtt{a}, \mathtt{b}\}$ and $\{\mathtt{b}\}$, respectively).

Finally we show that each answer set of a program $\mathcal{P}$ is also a minimal model of $\mathcal{P}$:

Proposition 1
If M is an answer set for $\mathcal{P}$, then M is a minimal model of $\mathcal{P}$.

Proof
By contradiction suppose M' is a model for $\mathcal{P}$ such that $M' \subset M$.

First we show that M' is a model for $pos(G_M(\mathcal{P}))$ too, i.e., every rule in $pos(G_M(\mathcal{P}))$ is satisfied in M'. Recall that $pos(G_M(\mathcal{P}))$ is the positive version of the program obtained by applying the transformation G_M to the program $ground(\mathcal{P})$. Consider a generic rule r of $ground(\mathcal{P})$. Since M' is a model for $\mathcal{P}$ either (i) r is overridden in M' or (ii) is satisfied in M'.

In case (i), since $M' \subset M$, from Definition 1 immediately follows that r is overridden in M too. Thus, r does not occur in $G_M(\mathcal{P})$ since the transformation G_M removes all rules overridden in M.

In case (ii) (i.e., r is satisfied in M'), if r is such that $B(r) \cap M \neq \emptyset$, then the rule is removed by G_M. Otherwise, r is transformed by G_M into a rule r' obtained from r by dropping the NAF part from the body. Since r is satisfied in M', also r' is satisfied in M'. As a consequence, all the rules of $pos(G_M(\mathcal{P}))$ are satisfied in M', that is $M' \subset M$ is a model for $pos(G_M(\mathcal{P}))$. Thus, by Definition 4, M is not an answer set for $\mathcal{P}$ since it is not a minimal model of $pos(G_M(\mathcal{P}))$. The proof is hence concluded.

4 Knowledge Representation with DLP$^<$

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

Example 7
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. }
penguin : bird   { ¬flies! }
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 explicitly 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`. 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`. is preferred to the default rule `flies`. as the hierarchy explicitly states the specificity of the former. Intuitively, there is no doubt that $\mathtt{M} = \{\neg\mathtt{flies}\}$ is the only reasonable conclusion. ∎

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

Example 8
Consider the following knowledge base representing a set of security specification about a simple *part-of* hierarchy of objects.

$$
\begin{array}{ll}
\mathtt{o_1} & \\
\quad \{ & \\
\qquad \mathtt{authorize(bob)} \leftarrow \mathtt{not\ authorize(ann)}. & (1)\\
\qquad \mathtt{authorize(ann)} \vee \mathtt{authorize(tom)} \leftarrow \mathtt{not}\ \neg\mathtt{authorize(alice)}. & (2)\\
\qquad \mathtt{authorize(amy)!} & (3)\\
\quad \} & \\
\mathtt{o_2} : \mathtt{o_1} & \\
\quad \{ & \\
\qquad \neg\mathtt{authorize(alice)!} & (4)\\
\quad \} & \\
\mathtt{o_3} : \mathtt{o_1} & \\
\quad \{ & \\
\qquad \neg\mathtt{authorize(bob)!} & (5)\\
\quad \} &
\end{array}
$$

Object $\mathtt{o_2}$ is part-of the object $\mathtt{o_1}$ as well as $\mathtt{o_3}$ is part-of $\mathtt{o_1}$. 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. Possible loss of control due to overriding mechanism can be avoided by using strict rules: strict authorizations cannot be overridden.

Consider the program $\mathcal{P}_{o_2} = \{(\mathtt{o_1}, \{(1),(2),(3)\}), (\mathtt{o_2}, \{(4)\})\}$ for the object $\mathtt{o_2}$ on the above knowledge base. This program defines the access control for the object $\mathtt{o_2}$. Thanks to the inheritance mechanism, authorizations specified for the object $\mathtt{o}_1$, to which $\mathtt{o}_2$ belongs, are propagated also to $\mathtt{o}_2$. It consists of rules (1), (2) and (3) (inherited from $\mathtt{o}_1$) and (4). Rule (1) states that bob is authorized to access object $\mathtt{o}_2$ provided that no authorization for ann to access $\mathtt{o}_2$ exists. Rule (2) authorizes either ann or tom to access $\mathtt{o}_2$ provided that no denial for alice to access $\mathtt{o}_2$ is derived. The strict rule (3) grants to amy the authorization to access object $\mathtt{o}_1$. Such authorization can be considered "strong", since no exceptions can be stated to it without producing inconsistency. As a consequence, all the answer sets of the program contain the authorization for amy. Finally, rule (4) defines a denial for alice to access object $\mathtt{o}_2$. Due to the absence of the authorization for ann, the authorization to bob of accessing the object $\mathtt{o}_2$ is derived (by rule (1)). Further, the explicit denial to access the object $\mathtt{o}_2$ for alice (rule (4)) allows to derive neither authorization for ann nor for tom (by rule (2)). Hence, the only answer set of this program is $\{\mathtt{authorize(bob)}, \neg\mathtt{authorize(alice)}, \mathtt{authorize(amy)}\}$.

Consider now the program $\mathcal{P}_{o_3} = \{(\mathtt{o}_1, \{(1), (2)\}), (\mathtt{o}_3, \{(5)\})\}$ for the object $\mathtt{o}_3$. Rule (5) defines a denial for bob to access object $\mathtt{o}_3$. The authorization for bob (defined by rule (1)) is no longer derived. Indeed, even if rule (1) allows to derive such an authorization due to the absence of authorizations for ann, it is overridden by the explicit denial (rule (5)) defined in the object $\mathtt{o}_3$ (i.e., at a more specific level). The body of rule (2) inherited from $\mathtt{o}_1$ is true for this program since no denial for alice can be derived, and it entails a mutual exclusive access to object $\mathtt{o}_3$ for ann and tom (note that no other head contains authorize(ann) or authorize(bob)). The program $\mathcal{P}_{o_3}$ admits two answer sets, namely $\{\mathtt{authorize(ann)}, \neg\mathtt{authorize(bob)}, \mathtt{authorize(amy)}\}$ and $\{\mathtt{authorize(tom)}, \neg\mathtt{authorize(bob)}, \mathtt{authorize(amy)}\}$ representing two alternative authorization sets to grant the access to the object $\mathtt{o}_3$. ∎

Solving the Frame Problem

The frame problem has first been addressed by McCarthy and Hayes (McCarthy and Hayes 1969), and in the meantime a lot of research has been conducted to overcome it (see e.g. (Shanahan 1997) 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 would be needed.

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 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 9

As an example we show how the famous Yale Shooting Problem, which is due to Hanks and McDermott (Hanks and McDermott 1987), can be represented and solved with 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.

The problem involves *temporal projection*: It is 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.

For the time framework we use the **DLV** bounded integer built-ins: The upper bound $\mathtt{n}$ of positive integers is specified by either adding the fact $\#\mathtt{maxint} = \mathtt{n}.$ to the program or by passing the option $-\mathtt{N} = \mathtt{n}$ on the commandline (this overrides any $\#\mathtt{maxint} = \mathtt{n}.$ statement). It is then possible to use the built-in constant $\#\mathtt{maxint}$, which evaluates to the specified upper bound, and several built-in predicates, of which in this paper we just use $\#\mathtt{succ}(\mathtt{N}, \mathtt{N1})$, which holds if $\mathtt{N1}$ is the successor of $\mathtt{N}$ and $\mathtt{N1} \leq \#\mathtt{maxint}$. For additional **DLV** built-in predicates, consult the **DLV** homepage (Faber and Pfeifer since 1996).

$$
\begin{array}{ll}
\mathtt{inertia} & \\
\quad \{ & \\
\qquad \mathtt{alive(T1)} \leftarrow \mathtt{alive(T)}, \#\mathtt{succ(T,T1)}. & (6)\\
\qquad \neg\mathtt{alive(T1)} \leftarrow \neg\mathtt{alive(T)}, \#\mathtt{succ(T,T1)}. & (7)\\
\qquad \mathtt{loaded(T1)} \leftarrow \mathtt{loaded(T)}, \#\mathtt{succ(T,T1)}. & (8)\\
\qquad \neg\mathtt{loaded(T1)} \leftarrow \neg\mathtt{loaded(T)}, \#\mathtt{succ(T,T1)}. & (9)\\
\quad \} & \\
\mathtt{domain} : \mathtt{inertia} & \\
\quad \{ & (10)\\
\qquad \mathtt{loaded(T1)} \leftarrow \mathtt{load(T)}, \#\mathtt{succ(T,T1)}! & (11)\\
\qquad \neg\mathtt{loaded(T1)} \leftarrow \mathtt{shoot(T)}, \mathtt{loaded(T)}, \#\mathtt{succ(T,T1)}! & (12)\\
\qquad \neg\mathtt{alive(T1)} \leftarrow \mathtt{shoot(T)}, \mathtt{loaded(T)}, \#\mathtt{succ(T,T1)}! & (13)\\
\quad \} & \\
\mathtt{yale} : \mathtt{domain} & \\
\quad \{ & (14)\\
\qquad \mathtt{load(0)}! \quad \mathtt{wait(1)}! \quad \mathtt{shoot(2)}! \quad \mathtt{alive(0)}! & (15)\\
\quad \} &
\end{array}
$$

The only answer set for this program (and $\#\mathtt{maxint} = 3$) contains, besides the facts of the `yale` object, $\mathtt{loaded(1)}$, $\mathtt{loaded(2)}$, $\mathtt{alive(0)}$, $\mathtt{alive(1)}$, $\mathtt{alive(2)}$ and $\neg\mathtt{loaded(3)}$,

$\neg\mathtt{alive}(3)$. 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 doing temporal projection (determining the effects of a given plan) as in the Yale Shooting Problem: You have to add a rule $\mathtt{action(T)} \vee \neg\mathtt{action(T)} \leftarrow \#\mathtt{succ(T,T1)}$. for every action, and you have to specify the goal state by a query, e.g. $\neg\mathtt{alive}(3), \neg\mathtt{loaded}(3)$? A query is a **DLV** language feature which (for this example) is equivalent to the rules $\mathtt{h} \leftarrow \neg\mathtt{alive}(3), \neg\mathtt{loaded}(3)$. and $\mathtt{i} \leftarrow \mathtt{not}\ \mathtt{h}, \mathtt{not}\ \mathtt{i}$., meaning that only answer sets containing $\neg\mathtt{alive}(3)$ and $\neg\mathtt{loaded}(3)$ should be considered.

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

Example 10
In (Erdem 1999), 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. (Gelfond and Lifschitz 1993, Dung 1993, Giunchiglia and Lifschitz 1998, Lifschitz 1999)). Then, a translation from these languages to another language (in our case 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 (Erdem 1999).

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

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

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

$$
\begin{array}{l}
\mathtt{bw\_inertia} \\
\quad \{ \\
\qquad \mathtt{on(B,L,T1)} \leftarrow \mathtt{on(B,L,T)}, \#\mathtt{succ(T,T1)}. \qquad (16) \\
\quad \}
\end{array}
$$

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

$$
\begin{array}{ll}
\mathtt{bw\_domain} : \mathtt{bw\_inertia} & \\
\quad \{ & \\
\qquad \mathtt{move(B,L,T)} \vee \neg\mathtt{move(B,L,T)} \leftarrow \mathtt{block(B)}, \mathtt{loc(L)}, \#\mathtt{succ(T,T1)}! & (17) \\
\qquad \mathtt{on(B,L,T1)} \leftarrow \mathtt{move(B,L,T)}, \#\mathtt{succ(T,T1)}! & (18) \\
\qquad \neg\mathtt{on(B,L,T1)} \leftarrow \mathtt{move(B,L1,T)}, \mathtt{on(B,L,T)}, \#\mathtt{succ(T,T1)}! & (19) \\
\qquad \leftarrow \mathtt{move(B,L,T)}, \mathtt{on(B1,B,T)}. & (20) \\
\qquad \leftarrow \mathtt{move(B,B1,T)}, \mathtt{on(B2,B1,T)}, \mathtt{block(B1)}. & (21)
\end{array}
$$

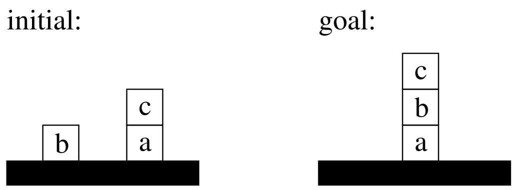

Figure 1. The Sussman Anomaly

$$\leftarrow \mathtt{move(B,B,T).} \tag{22}$$

$$\leftarrow \mathtt{move(B,L,T), move(B1,L1,T), B <> B1.} \tag{23}$$

$$\leftarrow \mathtt{move(B,L,T), move(B1,L1,T), L <> L1.} \tag{24}$$

$$\mathtt{loc(table)!} \tag{25}$$

$$\mathtt{loc(B)} \leftarrow \mathtt{block(B)!} \tag{26}$$

}

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. Rule (17) expresses that at any time $\mathtt{T}$, the action of moving a block $\mathtt{B}$ to location $\mathtt{L}$ may be initiated ($\mathtt{move(B,L,T)}$) or not ($\neg\mathtt{move(B,L,T)}$).

Rules (18) and (19) specify the effects of the move action: The moved block is at the target location at the next time, and no longer on the source location.

(20) – (24) 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: (20): A moved block must be clear. (21): The target of a move must be clear if it is a block (the table may hold an arbitrary number of blocks). (22) A block may not be on itself. (23) and (24): No two move actions may be performed at the same time.

The timesteps are again represented by **DLV**'s integer built-in predicates and constants.

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

$\mathtt{sussman} : \mathtt{bw\_domain}$

{

$$\mathtt{block(a)!} \quad \mathtt{block(b)!} \quad \mathtt{block(c)!} \tag{27}$$

$$\mathtt{on(b,table,0)!} \quad \mathtt{on(c,a,0)!} \quad \mathtt{on(a,table,0)!} \tag{28}$$

}

$$\mathtt{on(c,b,\#maxint), on(b,a,\#maxint), on(a,table,\#maxint)?} \tag{29}$$

Since different problem instances may involve different numbers of blocks, the blocks are defined as facts (27) together with the problem instance.[6]

We give the initial situation by facts (28), while the goal situation is specified by query

[5] We use constraints for clarity, but they can be eliminated by rewriting $\leftarrow \mathtt{B.}$ to $\mathtt{p} \leftarrow \mathtt{B, not\ p.}$, where $\mathtt{p}$ is a new symbol which does not appear anywhere else in the program.

[6] Note that usually the instance will be separated from the domain definition.

(29). 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 (Marek and Truszczyński 1991, Marek and Truszczyński 1990, Reiter 1980), two important decision problems, corresponding to two different reasoning tasks, arise in DLP$^<$:
(Brave Reasoning) Given a 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 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 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 DLP$^<$ programs.

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

Theorem 1
Brave Reasoning on DLP$^<$ programs is Σ_2^P-complete.

Proof
Given a ground 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 coNP, by virtue of Lemma 1. The problem therefore lies in Σ_2^P.

Σ_2^P-hardness follows from Theorem 3 and the results in (Eiter and Gottlob 1995, Eiter, Gottlob and Mannila 1997b). □

Theorem 2
Cautious Reasoning on DLP$^<$ programs is Π_2^P-complete.

Proof
Given a ground 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 coNP, by virtue of Lemma 1. Therefore, the complement of cautious reasoning is in Σ_2^P, and cautious reasoning is in Π_2^P.

Π_2^P-hardness follows from Theorem 3 and the results in (Eiter and Gottlob 1995, Eiter et al. 1997b). □

6 Related Work

6.1 Answer Set Semantics

Answer Set Semantics, proposed by Gelfond and Lifschitz in (Gelfond and Lifschitz 1991), 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 DLP$^<$ program consisting of a single object $o = \langle oid(o), \Sigma(o)\rangle$.[7] Then, M is an answer set of $\mathcal{P}$ if and only if it is a consistent answer set of $\Sigma(o)$ (as defined in (Gelfond and Lifschitz 1991)).

Proof
First we show that $G_M(\mathcal{P})$ is equal to $\Sigma(o)^M$ (as defined in (Gelfond and Lifschitz 1991)):

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 definition of Π^S in §7 in (Gelfond and Lifschitz 1991), 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 the criteria for a consistent set M of literals being an answer set of a positive (i.e. NAF free) program (as in (Gelfond and Lifschitz 1991)) is equal to the notion of satisfaction:

Since the set is consistent, condition (ii) in §7 of (Gelfond and Lifschitz 1991) does not apply. Condition (i) says: $L_{k+1}, \ldots, L_m \in M$ (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 (Gelfond and Lifschitz 1991) 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

[7] On inheritance-free programs, there is no difference between strict and defeasible rules. Therefore, without loss of generality we assume that rules are of only one type here. This allows us to drop the symbol ('.' or '!') at the end of the rules of single object programs.

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. □

Theorem 3 shows that the set of rules contained in a single object of a DLP$^<$ program has precisely the same answer sets (according to the definition in (Gelfond and Lifschitz 1991)) as the single object program (according to Definition 4).

For a DLP$^<$ program $\mathcal{P}$ consisting of more than one object, the answer sets (as defined in (Gelfond and Lifschitz 1991)) of the collection of all rules in $\mathcal{P}$ in general do not coincide with the answer sets of $\mathcal{P}$.

For instance the program

```
o       { p. }
o1 : o  { ¬p. }
```

has the answer set $\{\neg\mathtt{p}\}$, while the disjunctive logic program $\{\mathtt{p}.\ \neg\mathtt{p}.\}$ does not have a consistent answer set.

Nevertheless, in Section 7.1 we will show that each DLP$^<$ program $\mathcal{P}$ can be translated into a disjunctive logic program $\mathcal{P}'$, the semantics of which is equivalent to the semantics of $\mathcal{P}$. However, this translation requires the addition of a number of extra predicates.

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 (Buccafurri et al. 1998, Buccafurri et al. 1999). The DLP$^<$ language incorporates some ideas taken from $\mathcal{DOL}$. However, the two languages are very different in several respects. Most importantly, unlike with DLP$^<$, even if a program belongs to the common fragment of $\mathcal{DOL}$ and of the language of (Gelfond and Lifschitz 1991) (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.[8] In short, we observe the following differences between $\mathcal{DOL}$ and DLP$^<$:

- $\mathcal{DOL}$ does not include default negation `not`, while DLP$^<$ does.
- $\mathcal{DOL}$ and 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 $\Sigma(o) = \{\mathtt{p}.\quad \neg\mathtt{p}.\}$. Then, according to $\mathcal{DOL}$, the semantics of $\mathcal{P}$ is given by two models, namely, $\{\mathtt{p}\}$ and $\{\neg\mathtt{p}\}$. On the contrary, $\mathcal{P}$ has no answer set according to DLP$^<$ semantics.
- DLP$^<$ generalizes (consistent) Answer Set Semantics to disjunctive logic programs with inheritance, while $\mathcal{DOL}$ does not.

6.3 Prioritized Logic Programs

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 (Delgrande, Schaub

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

and Tompits 2000, Brewka and Eiter 1998, Gelfond and Son 1997, Nute 1994, Kowalski and Sadri 1990, Pradhan and Minker 1996, Sakama and Inoue 1996). However, we are aware of only one previous work on priorities in **disjunctive** programs, namely, the paper by Sakama and Inoue (Sakama and Inoue 1996). This interesting work can be seen as an extension of Answer Set Semantics to deal with priorities. Comparing the two approaches under the perspective 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 (Sakama and Inoue 1996), priorities are defined among **literals**, while priorities concern program **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 reasoning tasks. Indeed, from the simulation of abductive reasoning in the language of (Sakama and Inoue 1996), and the complexity results on abduction reported in (Eiter, Gottlob and Leone 1997a), it follows that brave reasoning is Σ_3^P-complete for the language of (Sakama and Inoue 1996). On the contrary, brave reasoning is "only" Σ_2^P-complete in $\mathrm{DLP}^{<}$[9].

(Delgrande et al. 2000) deals with nondisjunctive programs, but the authors note that their semantics-defining transformation "is also applicable to disjunctive logic programs". In this formalism, the preference relation is defined by regular atoms (with a set of constants representing the rules), allowing the definition of dynamic preferences. However, the semantics of the preferences is based on the order of rule application (or defeating) and thus seems to be quite different from our approach.

A comparative analysis of the various approaches to the treatment of preferences in ($\vee$-free) logic programming has been carried out in (Brewka and Eiter 1998).

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 (Touretzky 1986).

(Touretzky 1986) 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

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

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 being a royal elephant than to being an 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 DLP$^<$ fully agrees with the intuition underlying the *inferential distance ordering*.

Example 11
Let us represent the Royal Elephant example in our framework:

$$\begin{array}{ll} \texttt{elephant} & \{\texttt{gray}.\} \\ \texttt{royal\_elephant} : \texttt{elephant} & \{\neg\texttt{gray}.\} \\ \texttt{clyde} : \texttt{elephant},\ \texttt{royal\_elephant} & \{\ \} \end{array}$$

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

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

$$\begin{array}{ll} \texttt{republican} & \{\neg\texttt{pacifist}.\} \\ \texttt{quaker} & \{\texttt{pacifist}.\} \\ \texttt{nixon} : \texttt{republican}, \texttt{quaker} & \{\ \} \end{array}$$

This 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 DLP$^<$ could potentially be applied. Roughly, a simple formulation of the problem is the following: 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 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 $t_n < \cdots < t_0$.[10] The answer sets of the program for t_k can be taken as the semantics of the execution of

[10] In this context, $<$ should be interpreted as “more recent”.

$U_1, \cdots, U_k$ on P. For instance, given the logic program $\mathtt{P} = \{\mathtt{a} \leftarrow \mathtt{b}, \mathtt{not\ c}\}$ and the updates $\mathtt{U_1} = \{\mathtt{b.}\}$, $\mathtt{U_2} = \{\mathtt{c.}\}$, $\mathtt{U_3} = \{\neg\mathtt{b.}\}$, we build the DLP$^<$ program

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

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

The semantics of updates obtained in this way is very similar to the approach adopted for the ULL language in (Leone et al. 1995). Further investigations are needed on this topic to see whether DLP$^<$ can represent update problems in more general settings like those treated in (Marek and Truszczyński 1994) and in (Alferes et al. 1998a). A comparative analysis of various approaches to updating logic programs is being carried out in (Eiter, Fink, Sabbatini and Tompits 2000). Preliminaries results of this work show that, under suitable syntactic conditions, DLP$^<$ supports a nice "iterative" property for updates, which is missed in other formalisms.

7 Implementation Issues

7.1 *From* DLP$^<$ *to Plain DLP*

In this section we show how a 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 (DLP) systems for the implementation of DLP$^<$.

Notation.

1. Let $\mathcal{P}$ be the input DLP$^<$ program.
2. We denote a literal by $\phi(\bar{X})$, where $\bar{X}$ is the tuple of the literal's arguments, and ϕ 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 ϕ.
3. An adorned predicate ϕ is *conflicting* if both $\phi(\bar{X})$ and $\neg.\phi(\bar{Y})$ occur in the heads of rules in $\mathcal{P}$.
4. Given an object o in $\mathcal{P}$, and a head literal $\phi(\bar{X})$ of a defeasible rule in $\Sigma(o)$, we say that ϕ is *threatened in o* if a literal $\neg.\phi(\bar{Y})$ occurs in the head of a rule in $\Sigma(o')$ where $o' < o$. A defeasible rule r in $\Sigma(o)$ is *threatened in o* if all its head literals are threatened in o.

The rewriting algorithm translating DLP$^<$ programs in plain disjunctive logic programs with constraints[11] is shown in Figure 2.

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

[11] Again we use constraints for clarity, see footnote 5 on page 13

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

1: $DLP(\mathcal{P}) \Leftarrow \{prec'(o, o_1) \leftarrow \ | \ o < o_1\}$
2: **for** each object $o \in \mathcal{O}(\mathcal{P})$ **do**
3: **for** each threatened adorned predicate ϕ 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 ϕ and $X, X_1, \cdots, X_n$ are distinct variables.
7: **end for**
8: **for** each rule r in $\Sigma(o)$, say $\phi_1(\bar{X}_1) \vee \cdots \vee \phi_n(\bar{X}_n) \leftarrow BODY$, **do**
9: **if** r is threatened **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 ϕ 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 ϕ and $X_0, \cdots, X_n$ are distinct variables.
23: **end for**
24: **for** each conflicting adorned predicate ϕ 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 ϕ 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 threatened literal $\phi(\bar{X})$ appearing in o, rules defining when the literal is overridden are added (statements 3–7).
  - — For each rule r belonging to o:
    1. If r is threatened, 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 satisfies the rule if it is overridden. In addition, a rule is added which encodes when the rule is overridden (statements 9–12).
    2. Otherwise (i.e., if r is not threatened) just the rule head is rewritten as described above, since these rules cannot be overridden (statements 13–15).
- 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

19–23). The information in which object an atom has been derived is only needed for determination of overriding.

- Finally, statements 24–28 add a constraint for each adorned predicate, which prevents the generation of inconsistent sets of literals.

$DLP(\mathcal{P})$ is referred to as the *DLP version* of the program $\mathcal{P}$.[12]

We now give an example to show how the translation works:

Example 12

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

(1) **rules expliciting partial order among objects** :
$\mathtt{prec'(o_2, o_1).}$

(2) **rules for threatened adorned predicates in** o_1 :
$\mathtt{ovr'(a, o_1) \leftarrow \neg a'(X), prec'(X, o_1).}$
$\mathtt{ovr'(\neg b, o_1) \leftarrow b'(X), prec'(X, o_1).}$

(3) **rewriting of rules in** o_1 :
$\mathtt{a'(o_1) \lor \neg b'(o_1) \leftarrow c, not\ d, not\ ovr'(r_1, o_1).}$
$\mathtt{ovr'(r_1, o_1) \leftarrow ovr'(a, o_1), ovr'(\neg b, o_1).}$
$\mathtt{e'(o_1) \leftarrow b.}$

(4) **rewriting of rules in** o_2 :
$\mathtt{\neg a'(o_2) \lor c'(o_2).}$
$\mathtt{b'(o_2).}$
$\mathtt{c'(o_2) \leftarrow b.}$

(5) **projection rules** :
$\mathtt{a \leftarrow a'(X).} \qquad \mathtt{\neg a \leftarrow \neg a'(X).}$
$\mathtt{b \leftarrow b'(X).} \qquad \mathtt{\neg b \leftarrow \neg b'(X).}$
$\mathtt{c \leftarrow c'(X).} \qquad \mathtt{d \leftarrow d'(X).}$
$\mathtt{e \leftarrow e'(X).}$

(6) **constraints** :
$\mathtt{\leftarrow a, \neg a.}$
$\mathtt{\leftarrow b, \neg b.}$

∎

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 $\{prec', ovr'\} \cup \{\phi' \mid \exists\ an\ adorned\ literal\ \phi(\bar{X})\ appearing\ in\ \mathcal{P}\}$. $\pi(M)$ 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

Let $\mathcal{P}$ be a DLP$^<$-program. Then, for each answer set M for $\mathcal{P}$ there exists a consistent answer set M' for $DLP(\mathcal{P})$ such that $\pi(M') = M$. Moreover, for each consistent answer set M' for $DLP(\mathcal{P})$ there exists an answer set M for $\mathcal{P}$ such that $\pi(M') = M$.

[12] $DLP(\mathcal{P})$ is a function-free disjunctive logic program. Allowing functions could make the algorithm notation more compact, but would not give any computational benefit.

Proof

First we show that given an answer set M for $\mathcal{P}$ there exists a consistent answer set M' for $DLP(\mathcal{P})$ such that $\pi(M') = M$. 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 (defeasible) 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 $L \in Head(r)$, r is defeasible 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 corresponding literal $\phi'(obj\_of(r), \bar{X})$ occurs in K,
3. $K \subseteq \{\phi'(o, \bar{X}) \mid \phi(\bar{X})$ is an adorned predicate appearing in $\mathcal{P} \wedge o \in \mathcal{O}\}$,
4. K is a consistent set of literals.

First observe that the family $\mathcal{K}$ is not empty (i.e., there is at least a set of consistent sets of literals satisfying items (1), (2) and (3) above). This immediately follows from the fact that M is an answer set of the program $\mathcal{P}$.

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

Now we examine which rules the program $pos(G_{M_K}(DLP(\mathcal{P})))$ contains (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 statement 21 of Figure 2) appear unchanged in $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, these rules do not contain a NAF part (recall that the GL transformation can modify only rules in which a NAF part occurs). Each constraint of $DLP(\mathcal{P})$ (added by statement 26 of the algorithm), that is a rule of the form $b \leftarrow \phi(\bar{X}), \neg.\phi(\bar{X}), \texttt{not}\ b$ (where b is a literal not occurring in M_K) is translated into the rule $b \leftarrow \phi(\bar{X}), \neg.\phi(\bar{X})$.

The other rules in $pos(G_{M_K}(DLP(\mathcal{P})))$ originate from rules of $ground(DLP(\mathcal{P}))$ obtained by rewriting rules of $ground(\mathcal{P})$ (see statements 11 and 15 of the algorithm).

Thus, consider a rule r of $ground(\mathcal{P})$.

If r is defeasible and overridden in M then the corresponding rule in $ground(DLP(\mathcal{P}))$ (generated by statement 11 of the algorithm) contains a NAF part not satisfied in M_K, by construction of K_2 and K_3. Hence, such a rule appears neither in $pos(G_M(\mathcal{P}))$ nor in $pos(G_{M_K}(DLP(\mathcal{P})))$.

The other case we have to consider is that the rule r is either a strict rule or a defeasible rule not overridden in M.

First suppose that r is a strict rule or is a defeasible rule not threatened in M (recall that a rule not threatened in M is certainly not overridden in M). In this case, the corresponding rule, say r' in $ground(DLP(\mathcal{P}))$ (generated by statement 15 of the algorithm) has the same body of r and the head modified by renaming predicates (from ϕ to ϕ') and by adding the object o (from which the rule r comes) as first argument in each head literal. Since the body of r' does not contain literals from K_1, K_2, K_3 and K, and further $M \subseteq M_K$

(for each $K \in \mathcal{K}$), r' is eliminated by the GL transformation w.r.t. M_K if and only if r is eliminated by the GL transformation w.r.t. M. Moreover, in case r' is not eliminated by the GL transformation w.r.t M_K, since the body of r' does not contain literals from K_1, K_2, K_3 and K, the GL transformation w.r.t. M_K modifies the body of r' in the same way the GL transformation w.r.t. M modifies the body of r. Thus, each rule r in $pos(G_M(\mathcal{P}))$ has a corresponding rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ with the same body and a rewritten head.

Now suppose that r is a threatened defeasible rule that is not overridden in M. In this case, the corresponding rule, say r' in $ground(DLP(\mathcal{P}))$ (generated by statement 11 of the algorithm) has the head modified by renaming predicates (from ϕ to ϕ') and by adding the object o as first argument in each head literal and a body obtained by adding to the body of r a literal of the form $\texttt{not}\ ovr'(r, o, \bar{X})$, where o is the object from which r comes, and $\bar{X}$ represents the tuple of terms appearing in the head literals of r. Since the rule r is not overridden in M, the literal $ovr'(r, o, \bar{X})$ cannot belong to K_2 and hence cannot belong to M_K. Thus, the GL transformation w.r.t M_K eliminates the NAF part of the rule r'. As a consequence, also in this case, each rule in $pos(G_M(\mathcal{P}))$ has a corresponding rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ with

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. Rules of $pos(G_{M_K}(DLP(\mathcal{P})))$ of the form $b \leftarrow \phi(\bar{X}), \texttt{not}\ \phi(\bar{X})$, originated by the translation of the constraints, are satisfied since M_K is a consistent set of literals.

Consider now the remaining rules (those corresponding to rules of $pos(G_M(\mathcal{P}))$). Let r be a rule of $pos(G_{M_K}(DLP(\mathcal{P})))$ and r' the rule of $pos(G_M(\mathcal{P}))$ corresponding to r. As shown earlier, the two rules have the same body. Thus, if the body of r is true w.r.t. M_K, the body of r' is true w.r.t. M, since no literal of $M_K \setminus M$ can appear in the body of the rule r' (and hence of the rule r). As a consequence, by property (2) of the collection $\mathcal{K}$ to which the set K belongs, the head of the rule r is true in M_K.

Thus, M_K is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$.

Now we prove the following claim:

Claim 1. Let $\bar{M}$ be a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Then, $\pi(\bar{M})$ is a model for $pos(G_M(\mathcal{P}))$.

Proof
By contradiction suppose that $\pi(\bar{M})$ is not a model for $pos(G_M(\mathcal{P}))$. Thus, there exists a rule $r \in pos(G_M(\mathcal{P}))$ with body true in $\pi(\bar{M})$ and head false in $\pi(\bar{M})$. Since, as shown earlier, the rule r has a corresponding rule r' in $pos(G_{M_K}(DLP(\mathcal{P})))$ with the same body of r and the head obtained by replacing each literal $\phi(\bar{X})$ of the head of r by the corresponding literal $\phi'(o, \bar{X})$, where o is the object from which r comes. Since $\pi(\bar{M})$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$, at least one of the "$\phi'$ literal" of the head of r' must be true in $\pi(\bar{M})$. Then, due to the presence of the rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ in $pos(G_{M_K}(DLP(\mathcal{P})))$, $\pi(\bar{M})$ must contain also the "ϕ corresponding literal" belonging to the head of r (contradiction) □

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 item (1), suppose by contradiction $\bar{M}$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ such that $M \cap \bar{M} \neq M$. Thus, $\pi(\bar{M}) \subset M$. On the other hand, by Claim 1, $\pi(\bar{M})$ is a model for $pos(G_M(\mathcal{P}))$. But since M is an answer set for $\mathcal{P}$ and then a minimal model for $pos(G_M(\mathcal{P}))$, a contradiction arises.

Now we have to prove the item (2) above. First observe that the properties 3. and 4. of the family $\mathcal{K}$ are trivially verified by the models of $pos(G_{M_K}(DLP(\mathcal{P})))$. Thus, by contradiction suppose there exists a model $\bar{M}$ of $pos(G_{M_K}(DLP(\mathcal{P})))$ such that it does not satisfy one of the properties 1. or 2. characterizing the family $\mathcal{K}$.

First suppose that property 1. is not satisfied by $\bar{M}$, that is, there is a literal $\phi(\bar{X})$ in M such that no corresponding literal $\phi'(o, \bar{X})$ occurs in $\bar{M}$, for some object identifier o. Let $\bar{M}'$ be 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 "ϕ' literals" by hypothesis. Further, no other rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ contains a literal from M in the head. On the other hand, by Claim 1, $\pi(M')$ is a model for $pos(G_M(\mathcal{P}))$. But this is a contradiction, since $\pi(M') \subset M$ and M is an answer set for $\mathcal{P}$.

Suppose now that property 2. is not satisfied by $\bar{M}$, that is, there is a rule $r \in pos(G_M(\mathcal{P}))$ such that the body of r is true in M and no "ϕ' literals" corresponding to literals of the head occur in $\bar{M}$. Since, $M \subseteq \bar{M}$ (see item (1) above), the corresponding rule of $pos(G_{M_K}(DLP(\mathcal{P})))$ is not satisfied. But this implies that $\bar{M}$ can not be a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ (contradiction).

A consequence of the fact that every model of $pos(G_{M_K}(DLP(\mathcal{P})))$ contains M is that every model of $pos(G_{M_K}(DLP(\mathcal{P})))$ must contain the sets K_1, K_2 and K_3. because of the rules added by statements 1,5 and 12 of the algorithm.

Thus, the model $M' = M_{K'}$ for $K' \in \mathcal{K}$ such that no set $\bar{K} \in \mathcal{K}$ exists such that $\bar{K} \subset K'$ is a minimal model for $pos(G_{M_K}(DLP(\mathcal{P}))$, that is, a consistent answer set for $DLP(\mathcal{P})$. Hence, the first part of the proof is concluded, since $\pi(M') = M$.

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

First we prove that a literal $\phi(\bar{X})$ belongs to M if and only if there exits a literal $\phi'(o, \bar{X})$ in M', for some object identifier o.

Indeed, $\phi(\bar{X}) \in M$ implies that $\phi(\bar{X}) \in M'$. But since M' is a minimal model for $pos(G_{M'}(DLP(\mathcal{P}))$, there must exits a rule in $pos(G_{M'}(DLP(\mathcal{P}))$ with head containing the literal $\phi(\bar{X})$ and body true w.r.t. M' (otherwise the literal $\phi(\bar{X})$ could be dropped from M' without invalidate any rule of $pos(G_{M'}(DLP(\mathcal{P}))$ and thus M' would not be minimal). Conversely, if $\phi'(o, \bar{X}) \in M'$, for some object identifier o, the literal $\phi(\bar{X})$ belongs to M', since M' is a model for $pos(G_{M'}(DLP(\mathcal{P}))$ and the rule $\phi(\bar{X}) \leftarrow \phi'(o, \bar{X})$ belongs to $pos(G_{M'}(DLP(\mathcal{P}))$. Thus, $\phi(\bar{X}) \in M$.

Moreover, we prove that every rule of $pos(G_{M'}(DLP(\mathcal{P}))$ has a corresponding rule in $pos(G_M(\mathcal{P}))$ with same body and a head obtained by replacing the ϕ' literals with the ϕ corresponding ones and by eliminating the object argument from these literals. Indeed, from the above result, the GL transformation deletes a rule r from $ground(DLP(\mathcal{P}))$ if either the corresponding rule belonging to $ground(\mathcal{P})$ is overridden in M (due the the

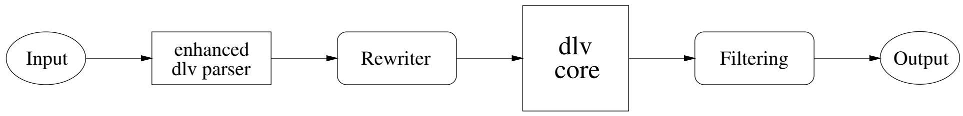

Figure 3. Flow diagram of the system.

literal not $ovr'(r, o, \bar{X})$ occurring in the body of r) or some negated (by negation not) literal is false in M'. But this literal is false in M' if and only if it is false in M. On the other hand, in case the rule is not deleted by the GL transformation, its body is rewritten in the same way of the corresponding rule appearing in $pos(G_M(\mathcal{P}))$.

As a consequence, M is a model for $pos(G_M(\mathcal{P}))$. Indeed, if the body of a rule r of $pos(G_M(\mathcal{P}))$ is true w.r.t. M, the corresponding rule r' of $pos(G_{M'}(DLP(\mathcal{P}))$ has the body true w.r.t. M'. Hence, at least one of the head literals of r' must be true in M'. Let $\phi'(o, \bar{X})$ such a literal. As shown earlier, this implies that $\phi(\bar{X})$ belongs to M. But $\phi(\bar{X})$ appears in the head of r and hence r is satisfied in M.

Now we prove that M is minimal. By contradiction, suppose that $\bar{M} \subset M$ is a model for $pos(G_M(\mathcal{P}))$. Consider the literals belonging to the set $M \setminus \bar{M}$. Because of the correspondence between the rules of $pos(G_M(\mathcal{P}))$ and the rules of $pos(G_{M'}(DLP(\mathcal{P}))$, the set of literals obtained from M' by eliminating all the literals $\phi(\bar{X})$ belonging to the set $M \setminus \bar{M}$ as well as the corresponding ϕ' literals is still a model for $pos(G_{M'}(DLP(\mathcal{P}))$. But this is a contradiction, since M' is a consistent answer set for $DLP(\mathcal{P})$.

Since M is a model for $pos(G_M(\mathcal{P}))$ and is minimal, $M = \pi(M')$ is an answer set for the program $\mathcal{P}$. Hence the proof is concluded. □

Example 13
Consider the program $\mathcal{P}$ of Example 1. It is easy to see that $DLP(\mathcal{P})$ (see Example 12) admits one consistent answer set $M = \{prec'(o_2, o_1), a'(o_1), e'(o_1), b'(o_2), c'(o_2), ovr'(\neg b, o_1), a, e, b, c\}$. Thus, $\pi(M) = \{a, b, c, e\}$. On the other hand, $\pi(M)$ is the only answer set for $\mathcal{P}$, as shown in Example 6. ∎

7.2 System Architecture

We have used the **DLV** system (Eiter et al. 1998) to implement a system for DLP$^<$. The concept is that of a front-end to plain DLP, which has been used before for implementing various ways of reasoning modes and languages on top of the **DLV** system. The front-end implements the translation described in Section 7. A schematic visualization of its architecture is shown in Figure 3.

First of all, we have extended the **DLV** parser to incorporate the DLP$^<$ syntax. In this way, all the advanced features of **DLV** (e.g. bounded integer arithmetics, comparison built-ins, etc.) are also available with 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, π is applied to each answer set in order to strip the internal predicates from the output.

On the webpage (Faber 1999) the system is described in detail. It has been fully incorporated into the **DLV** system. To use it, just supply an input file using the syntax described in Section 2 — **DLV** automatically invokes the DLP$^<$ frontend in this case.

Note that it is currently required to specify the objects in the order of the inheritance hierarchy: Specifying that some object inherits from another object which has not been defined before will result in an error. Since cyclic dependencies are not allowed in our language, this requirement is not a restriction.

8 Conclusion

We have presented a new language, named DLP$^<$, resulting from the extension of (function-free) disjunctive logic programming with inheritance. DLP$^<$ comes with a declarative model-theoretic semantics, where possible conflicts are solved in favor of more specific rules (according to the inheritance hierarchy). DLP$^<$ is a consistent generalization of the Answer Set Semantics of disjunctive logic programs, it respects some fundamental inheritance principles, and it seems to be suitable also to give a semantics to updates in logic programs.

While inheritance enhances the knowledge representation modeling features of disjunctive logic programming, the addition of inheritance does not increase its computational complexity. Thus, inheritance "comes for free": the user can take profit of its knowledge modeling ability, without paying any extra cost in terms of computational load. It was therefore possible to implement a DLP$^<$ system on top of the disjunctive logic programming system **DLV**. The system is freely available on the web (Faber 1999) and ready-to-use for experimenting with the use of inheritance in KR applications.

Acknowledgements

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

References

Alferes, J. J. and Pereira, L. M. (1992). On Logic Program Semantics with Two Kinds of Negation, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'92)*, MIT Press, pp. 574–588.

Alferes, J. J., Leite, J. A., Pereira, L. M., Przymusinska, H. and Przymusinski, T. C. (1998a). Dynamic Logic Programming, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann, pp. 98–111.

Alferes, J. J., Pereira, L. M. and Przymusinski, T. C. (1996). Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming, *European Workshop on Logics in Artificial Intelligence (JELIA'96)*, Springer, pp. 143–163.

Alferes, J. J., Pereira, L. M. and Przymusinski, T. C. (1998b). 'Classical' Negation in Nonmonotonic Reasoning and Logic Programming, *Journal of Automated Reasoning* **20**(1/2): 107–142.

Baral, C. and Gelfond, M. (1994). Logic Programming and Knowledge Representation, *Journal of Logic Programming* **19/20**: 73–148.

Brewka, G. and Eiter, T. (1998). Preferred Answer Sets for Extended Logic Programs, *in* A. Cohn, L. Schubert and S. Shapiro (eds), *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann, pp. 86–97.

Buccafurri, F., Leone, N. and Rullo, P. (1998). Disjunctive Ordered Logic: Semantics and Expressiveness, *in* A. G. Cohn, L. Schubert and S. C. Shapiro (eds), *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann Publishers, pp. 418–429.

Buccafurri, F., Leone, N. and Rullo, P. (1999). Semantics and Expressiveness of Disjunctive Ordered Logic, *Annals of Mathematics and Artificial Intelligence* **25**(3-4): 311–337.

Delgrande, J., Schaub, T. and Tompits, H. (2000). Logic programs with compiled preferences, *in* W. Horn (ed.), *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, IOS Press, pp. 392–398.

Dung, P. M. (1993). Representing Actions in Logic Programming and Its Applications in Database Updates, *Proceedings of the Tenth International Conference on Logic Programming*, pp. 222–238.

Eiter, T. and Gottlob, G. (1995). On the Computational Cost of Disjunctive Logic Programming: Propositional Case, *Annals of Mathematics and Artificial Intelligence* **15**(3/4): 289–323.

Eiter, T., Fink, M., Sabbatini, G. and Tompits, H. (2000). Updating Logic Programs (Revisited), *Proceedings of 7th European Workshop Logic in Artificial Intelligence – JELIA 2000*. Accepted.

Eiter, T., Gottlob, G. and Leone, N. (1997a). Abduction from Logic Programs: Semantics and Complexity, *Theoretical Computer Science* **189**(1–2): 129–177.

Eiter, T., Gottlob, G. and Mannila, H. (1997b). Disjunctive Datalog, *ACM Transactions on Database Systems* **22**(3): 364–418.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. (1998). The KR System `dlv`: Progress Report, Comparisons and Benchmarks., *in* A. G. Cohn, L. Schubert and S. C. Shapiro (eds), *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann Publishers, pp. 406–417.

Erdem, E. (1999). Applications of Logic Programming to Planning: Computational Experiments, Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`.

Faber, W. (1999). `dlvi` homepage. `<URL:http://www.dbai.tuwien.ac.at/proj/dlv/inheritance/>`.

Faber, W. and Pfeifer, G. (since 1996). `dlv` homepage. `http://www.dbai.tuwien.ac.at/proj/dlv/`.

Gelfond, M. and Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing* **9**: 365–385.

Gelfond, M. and Lifschitz, V. (1993). Representing Action and Change by Logic Programs, *Journal of Logic Programming* **17**: 301–321.

Gelfond, M. and Son, T. C. (1997). Reasoning with Prioritized Defaults, *Proceedings of the Workshop of Logic Programming and Knowledge Representation (LPKR '97)*, Springer, pp. 164–223.

Giunchiglia, E. and Lifschitz, V. (1998). An Action Language Based on Causal Explanation: Preliminary Report, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pp. 623–630.

Hanks, S. and McDermott, D. (1987). Nonmonotonic logic and temporal projection, *Artificial Intelligence* **33**(3): 379–412.

Kowalski, R. A. and Sadri, F. (1990). Logic Programs with Exceptions, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, MIT Press, pp. 598–616.

Leone, N., Palopoli, L. and Romeo, M. (1995). A Language for Updating Logic Programs and its Implementation, *Journal of Logic Programming* **23**(1): 1–61.

Lifschitz, V. (1999). Action Languages, Answer Sets and Planning, *in* K. Apt, V. W. Marek, M. Truszczyński and D. S. Warren (eds), *The Logic Programming Paradigm – A 25-Year Perspective*, Springer Verlag, pp. 357–373.

Lobo, J., Minker, J. and Rajasekar, A. (1992). *Foundations of Disjunctive Logic Programming*, The MIT Press, Cambridge, Massachusetts.

Marek, V. and Truszczyński, M. (1994). Revision Specifications by Means of Programs, *European Workshop on Logics in Artificial Intelligence (JELIA'94)*, Springer, pp. 122–136.

Marek, W. and Truszczyński, M. (1990). Modal Logic for Default Reasoning, *Annals of Mathematics and Artificial Intelligence* **1**(1-4): 275–302.

Marek, W. and Truszczyński, M. (1991). Autoepistemic Logic, *Journal of the ACM* **38**(3): 588–619.

McCarthy, J. (1990). *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*, Ablex.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence, *in* B. Meltzer and D. Michie (eds), *Machine Intelligence 4*, Edinburgh University Press, pp. 463–502. reprinted in (McCarthy 1990).

Nute, D. (1994). Defeasible logic, *in* D. M. Gabbay, C. Hogger and J. Robinson (eds), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 3, Oxford University Press, pp. 353–395.

Pradhan, S. and Minker, J. (1996). Using Priorities to Combine Knowledge Bases, *International Journal of Cooperative Information Systems* **5**(2&3): 333–364.

Reiter, R. (1980). A Logic for Default Reasoning, *Artificial Intelligence* **13**(1–2): 81–132.

Sakama, C. and Inoue, K. (1996). Representing Priorities in Logic Programs, *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming (JICSLP'96)*, MIT Press, pp. 82–96.

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

Touretzky, D. S. (1986). *The Mathematics of Inheritance Systems*, Pitman, London.