Abduction in Logic Programming:
A New Definition and an Abductive Procedure Based on Rewriting

Fangzhen Lin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We propose a new definition of abduction in logic programming, and contrast it with that of Kakas and Mancarella's. We then introduce a rewriting system for answering queries and generating explanations, and show that it is both sound and complete under the partial stable model semantics and sound and complete under the answer set semantics when the underlying program is so-called odd-loop free. We discuss an application of the work to a problem in reasoning about actions and provide some experimental results.

1 Abduction in logic programming

In general, given a background theory T, and an observation q to explain, an abduction of q w.r.t. T is a theory Π such that $\Pi \cup T \models q$. Normally, we want to put some additional conditions on Π, such as that it is consistent with T and contains only those propositions called abducibles. For instance, in propositional logic, given a background theory T, a set A of assumptions or abducibles, and a proposition q, an explanation S of q is commonly defined (see [Reiter and de Kleer, 1987], [Poole, 1988], and [Konolige, 1992]) to be a minimal set of literals over A such that $T \cup S \models q$ and $T \cup S$ is consistent.

In the context of logic programming, abduction has been investigated from both proof-theoretic and model-theoretic perspectives (e.g. [Eshghi and Kowalski, 1989; Kakas and Mancarella, 1990; Satoh and Iwayama, 1992]). One of the most influential definitions of abduction in logic programming is that of Kakas and Mancarella's generalized stable model semantics [1990]. Given a logic program P, a set A of atoms standing for abducibles, and a query q, Kakas and Mancarella define an abductive explanation S to be a subset of A such that there is an answer set (also called a stable model) M of $P \cup S$ that satisfies q.

One can see the following two differences between this definition and the one that we defined above for propositional logic: In propositional logic, S is a set of literals, but in logic programming, it is just a set of atoms; In propositional logic, S must be minimal, in terms of the subset ordering relation; but there is no such requirement in the case of logic programming.

One could argue that these differences are due to the fact that under the answer set semantics, negation is considered to be "negation-as-failure." If none of the atoms in A appear in the head of a rule in the logic program P, then adding a set $S \subseteq A$ to P really means that we are adding the complete literal set, $S \cup \{\neg p \mid p \in A, p \notin S\}$, to P. This would also explain why there is no minimality condition in the definition: two complete sets of literals are never comparable in terms of the subset relation.

However, while this notion of abductive explanations makes sense in theory, it is problematic in practice. For instance, if $A = \{a, b\}$, and $P = \{q \leftarrow a.\}$, then there are two abductive explanations for q according to Kakas and Mancarella's definition: $\{a\}$ and $\{a, b\}$. In general, if A has n elements, then there are 2^{n-1} abductive explanations for q, a number that is too big to manage.

Since in this case a is the explanation that we are looking for, it is tempting here to say that we should prefer minimal abductive explanations like what we did for propositional logic. As we mentioned above, this does not make sense if we take an abductive explanation to be a complete set of literals as implied by the answer set semantics. However, one can still try to minimize the set of atoms, in this case, preferring $\{a\}$ over $\{a, b\}$.

However, this minimization strategy is problematic when a program contains negation. Consider a situation in which a boat can be used to cross a river if it is not leaking or, if it is leaking, there is a bucket available to scoop the water out of the boat. This can be axiomatized by the following logic program P:

$$canCross \leftarrow boat, \mathsf{not}\, leaking.$$
$$canCross \leftarrow boat, leaking, hasBucket.$$

Now suppose that we saw someone crossed the river, how do we explain that? Clearly, there are two possible explanations: either the boat is not leaking or the person has a bucket with her. In terms of Kakas and Mancarella's definition, there are three abductive explanations for $canCross$, $\{boat\}$, $\{boat, hasBucket\}$, and $\{boat, leaking, hasBucket\}$, assuming that $A = \{boat, leaking, hasBucket\}$ is the set of abducibles. But only one of them, $\{boat\}$, is minimal.

On a closer look, we see that in our first example, when we say that $\{a\}$ is a preferred explanation over all the others, we do not mean the complete set of literals $\{a, \neg b\}$, is preferred over all the others. While we want a to be part of the explanation, we don't necessarily want $\neg b$ because we do not want to apply negation as failure on abducibles, which are assumptions one can make one way or the other. What we want is for the set $\{a\}$ itself to be the best explanation for q.

One way to justify this is that all possible ways of completing this set into a complete set of literals, $\{a, \neg b\}$ and $\{a, b\}$, turn out to correspond to all the abductive explanations of q according to Kakas and Mancarella's definition. The same kind of justification turns out to work for our second example as well: the reason that $\{boat\}$ is not a preferred explanation is that while its completion according to negation-as-failure, $\{boat, \neg leaking, \neg hasBucket\}$ is an explanation, some of its other completions, for example $\{boat, leaking, \neg hasBucket\}$ is not an explanation. This motivates our following definitions.

2 Abduction in logic programming revisited

In this paper, we consider *(normal) logic programs* which are sets of rules of the form

$$a \leftarrow b_1, ..., b_m, \mathsf{not}\, c_1, ..., \mathsf{not}\, c_n$$

where a, b_i and c_i are atoms of the underlying propositional language L. Here "not" is the so-called *default negation*, and defined according to the answer set semantics [Gelfond and Lifschitz, 1988].

Let P be a logic program, A a set of propositions standing for abducibles, and q a proposition. In the following, without loss of generality, we shall assume that none of the abducibles in A occur in the head of a rule in P.[1]

In the following, by a *hypothesis* α we mean a consistent set of literals over A, i.e. it is not the case that p and $\neg p$ are both in α for some $p \in A$. We say that a hypothesis is *complete* if for each atom $p \in A$, either p or $\neg p$ is in α, but not both. Notice that a complete hypothesis is really a truth-value assignment over the language A. We say that a hypothesis α is an *extension* of another one β if $\beta \subseteq \alpha$, and a complete extension is an extension that is complete.

Definition 2.1 *A complete hypothesis α is said to be an explanation of q w.r.t. P and A iff there is an answer set M of $P \cup \alpha^+$ such that M contains q and for any $\neg p \in \alpha$, $p \notin M$, where α^+ is the set of atoms in α.*

Definition 2.2 *A hypothesis is said to be an explanation of q iff every complete extension of it is an explanation. A hypothesis α is said to be a minimal explanation if it is an explanation, and there is no other explanation α' such that $\alpha' \subset \alpha$.*

Consider the logic program P in the previous section about $canCross$. The following are the complete hypotheses that explain $canCross$:

$\{boat, \neg leaking, \neg hasBucket\}$,

$\{boat, \neg leaking, hasBucket\}$, $\{boat, leaking, hasBucket\}$.

Now consider $\{boat, hasBucket\}$. Clearly every complete extension of this set is an explanation, so it is an explanation as well. Furthermore, it is a minimal explanation as none of its element can be deleted for it continue to be an explanation. Similarly, $\{boat, \neg leaking\}$ is also a minimal explanation.

If we take a hypothesis to be the conjunction of its elements, then we have that in the propositional logic,

$$\bigvee_{\alpha \in \mathcal{S}_1} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_3} \alpha$$

where $\mathcal{S}_1$ is the set of all complete hypotheses that are explanations of q, $\mathcal{S}_2$ the set of all explanations of q, and $\mathcal{S}_3$ the set of all minimal explanations of q. Therefore the set of minimal explanations is a succinct representation of the set of all explanations.

It is clear from our definition that a complete hypothesis α is an explanation of q iff α^+ is an abductive explanation of q according to Kakas and Mancarella's definition. This implies that if none of the abducibles occur in the head of any clauses in P, then

$$\bigvee_{S \in \mathcal{S}_1} cl(S) \equiv \bigvee_{\alpha \in \mathcal{S}_2} \alpha,$$

where $\mathcal{S}_1$ is the set of all abductive explanation of q according to Kakas and Mancarella's definition, $cl(S) = S \cup \{\neg p \mid p \in A, p \notin S\}$, and $\mathcal{S}_2$ is the set of all minimal explanations of q.

So in a sense, the set of Kakas and Mancarella's abductive explanations and that of our minimal explanations are equivalent. However, as we have seen above, the number of abductive explanations can be very large. Enumerating them all is impossible even in simple, small domains. In contrast, the number of minimal explanations are much smaller. More importantly, just like explanations in propositional logic, they only include "relevant propositions."

But computationally, it may be hard to compute minimal explanations from scratch. It is often easier to compute first a small "cover" of all explanations.

Definition 2.3 *A set $\mathcal{S}$ of hypotheses is said to be a* cover *of q w.r.t. P and A iff*

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \equiv \bigvee_{\alpha \in \mathcal{S}_0} \alpha,$$

where $\mathcal{S}_0$ is the set of minimal explanations of q.

Proposition 2.4 *If $\mathcal{S}$ is a cover of q, then each $\alpha \in \mathcal{S}$ must be an explanation of q.*

So a cover is a set of explanations such that any complete explanation must be an extension of one of the explanations in the cover. Once we have a cover, then we can find all minimal explanations by propositional reasoning alone. Recall that a conjunction of literals α is a *prime implicant* of a formula φ if $\alpha \models \varphi$, and there is no other β such that $\beta \models \varphi$ and β is a subset of α, i.e. α is a minimal conjunction of literals that entails φ.

Proposition 2.5 *Let $\mathcal{S}$ be a cover of q. Then a hypothesis is a minimal explanation of q iff it is a prime implicant of $\bigvee_{\alpha \in \mathcal{S}} \alpha$.*

In the rest of this paper, we shall propose a rewriting system for generating explanations of a proposition in a logic program. We shall first define it for logic programs without abducibles. We will then extend the system to logic programs

[1] If $p \in A$ occurs in the head of a rule, then we can always introduce a new proposition, say p', add the rule $p \leftarrow p'$ to P, add p' to A and delete p from A.

with abducibles, and show that for any query, the rewriting system generates an approximation of a cover set in the general case, and exactly a cover set when the given logic program has no so-called “odd loops.” We will then discuss an application of our system to reasoning about actions and present some experimental results.

3 Goal Rewrite Systems

Given a (ground) program P, the *Clark completion* of P, denoted $Comp(P)$, is the following set of equivalences: for each atom $\phi \in L$,

- if ϕ does not appear as the head of any rule in P, $\phi \leftrightarrow F \in Comp(P)$.
- otherwise, $\phi \leftrightarrow B_1 \vee ... \vee B_n \in Comp(P)$ (with default negations replaced by negative literals), if there are exactly n rules $\phi \leftarrow B_i \in P$ with ϕ as the head. We write T for B_i if B_i is empty.

The idea of goal rewriting is simple. A completed definition $\phi \leftrightarrow B_1 \vee \ldots \vee B_n \in Comp(P)$ can be used as a rewrite rule from left to right: ϕ is rewritten to $B_1 \vee \ldots \vee B_n$, and $\neg\phi$ to $\neg B_1 \wedge \ldots \wedge \neg B_n$. We call these *literal rewriting*, and the completed definitions *program (rewrite) rules.*

A goal is a formula which may involve $\neg$, $\vee$ and $\wedge$. A goal is also referred to as a *goal formula*. A goal resulted from a literal rewriting from another goal is called a *derived goal*. A goal with negation appearing only in front of atoms is said to be *signed*, a term introduced in [Kunen, 1989] for a similar purpose. For convenience, we assume that *all goals are signed*, which can be achieved easily by simple transformations using the following rules: for any formulas Φ and Ψ,

$$\begin{array}{l}\neg\neg\Phi \rightarrow \Phi \\ \neg(\Phi \vee \Psi) \rightarrow \neg\Phi \wedge \neg\Psi \\ \neg(\Phi \wedge \Psi) \rightarrow \neg\Phi \vee \neg\Psi\end{array}$$

Like a formula, a goal may be further transformed to a suitable form for literal rewriting without changing its semantics. With a mechanism of loop handling, rewriting of a goal Q terminates at either T meaning that Q is proved, or F meaning that Q is not proved. Hence, a goal rewrite system consists of three types of rewrite rules: (i) program rules from $Comp(P)$ for literal rewriting, (ii) simplification rules to transform and simplify goals, and (iii) loop rules for handling loops.

3.1 Simplification rules

The simplification subsystem is formulated with a mechanism of loop handling in mind, which requires keeping track of literal sequences $g_0, ..., g_n$ where each g_i, $0 < i \leq n$, is in the goal formula resulted from rewriting g_{i-1}. Two central mechanisms in formalizing goal rewrite systems are *rewrite chains* and *contexts*.

- *Rewrite Chain:* Suppose a literal l is written by its definition $\phi \leftrightarrow \Phi$ where $l = \phi$ or $l = \neg\phi$. Then, each literal l' in the derived goal is generated in order to prove l. This ancestor-descendant relation is denoted $l \prec l'$. A sequence $l_1 \prec \ldots \prec l_n$ is then called a *rewrite chain*, abbreviated as $l_1 \prec^+ l_n$. Notice that it is essential here that any goal G be in the form of a signed goal, and that when $\neg p$ is in G, we have that $l \prec \neg p$ but not $l \prec p$.
- *Context:* A rewrite chain $g = g_0 \prec g_1 \prec \ldots \prec g_n = T$ records a set of literals $C = \{g_0, ..., g_{n-1}\}$ for proving g. We will write $T(\{g_0, ..., g_{n-1}\})$ and call C a *context*.

  For simplicity, we assume that whenever $\neg F$ is generated, it is automatically replaced by $T(C)$, where C is the set of literals on the corresponding rewrite chain, and $\neg T$ is automatically replaced by F.

Note that for every literal in any derived goal, the rewrite chain leading to it from a literal in the given goal is uniquely determined. As an example, suppose the completion of a program is: $\{a \leftrightarrow \neg b \wedge \neg c, b \leftrightarrow q \vee \neg p\}$. We then have a rewrite sequence $a \rightarrow \neg b \wedge \neg c \rightarrow \neg q \wedge p \wedge \neg c$. For the three literals in the last goal, we have the following rewrite chains for them from a: $a \prec \neg b \prec \neg q$, $a \prec \neg b \prec p$, and $a \prec \neg c$.

Simplification Rules:

Let Φ_i's be any goal formulas, C a context, and l a literal.

SR1. $F \vee \Phi \rightarrow \Phi$

SR1'. $\Phi \vee F \rightarrow \Phi$

SR2. $F \wedge \Phi \rightarrow F$

SR2'. $\Phi \wedge F \rightarrow F$

SR3. $T(C_1) \wedge T(C_2) \rightarrow T(C_1 \cup C_2)$ if $C_1 \cup C_2$ is consistent

SR4. $T(C_1) \wedge T(C_2) \rightarrow F$ if $C_1 \cup C_2$ is inconsistent

SR5. $T(C) \wedge l \rightarrow F$ if $\neg l \in C$

SR5'. $l \wedge T(C) \rightarrow F$ if $\neg l \in C$

SR6. $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \rightarrow (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$

SR6'. $(\Phi_1 \vee \Phi_2) \wedge \Phi_3 \rightarrow (\Phi_1 \wedge \Phi_3) \vee (\Phi_2 \wedge \Phi_3)$ □

The simplification system is a nondeterministic transformation system. The primed version of a rule is its symmetric case. Most of the rules are about the logical equivalence between the two sides of a rule. SR3 merges two contexts if they are consistent, otherwise SR4 makes it a failure to prove. SR5 and SR5' prevent generating an inconsistent context before literal l is even proved.

Note that the proof-theoretic meaning of a goal formula may not be the same as the logical meaning of the formula. E.g., the goal formula $a \vee \neg a$ (a tautology in classic logic) could well lead to an F if neither a nor $\neg a$ can be proved.

For goal rewriting that does not involve loops, the system described so far is sufficient.

Example 3.1 *Let P be*

$$\begin{array}{ll} g \leftarrow \mathsf{not}\, a & \\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c; & a \leftarrow b, \mathsf{not}\, d \\ b \leftarrow q; & b \leftarrow \mathsf{not}\, p \end{array}$$

Then $Comp(P)$ is:

$$\begin{array}{l} g \leftrightarrow \neg a; \quad a \leftrightarrow (\neg b \wedge \neg c) \vee (b \wedge \neg d); \quad b \leftrightarrow q \vee \neg p \\ q \leftrightarrow F; \quad p \leftrightarrow F; \quad d \leftrightarrow F; \quad c \leftrightarrow F \end{array}$$

The rewrite sequence below is generated by focusing on the left part of a goal.

$$
\begin{array}{l}
g \to \neg a \\
\to (b \vee c) \wedge (\neg b \vee d) \\
\to (q \vee \neg p \vee c) \wedge (\neg b \vee d) \\
\to (F \vee \neg p \vee c) \wedge (\neg b \vee d) \\
\to (\neg p \vee c) \wedge (\neg b \vee d) \\
\to (T(C) \vee c) \wedge (\neg b \vee d) \quad \textit{where } C = \{g, \neg a, b, \neg p\} \\
\to (T(C) \wedge (\neg b \vee d)) \vee (c \wedge (\neg b \vee d)) \\
\to (T(C) \wedge \neg b) \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \ \texttt{\%}\text{apply SR5} \\
\to F \vee (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \\
\to (T(C) \wedge d) \vee (c \wedge (\neg b \vee d)) \\
\to (T(C) \wedge F) \vee (c \wedge (\neg b \vee d)) \\
\to F \vee (c \wedge (\neg b \vee d)) \\
\to c \wedge (\neg b \vee d) \to F \wedge (\neg b \vee d) \to F
\end{array}
$$

3.2 Loop rules

After a literal l is rewritten, it is possible that at some later stage either l or $\neg l$ appears again in a goal on the same rewrite chain. Thus, a loop is a rewrite chain $\{l_1, ..., l_n\}$ such that $l_1 = l_n$, or $l_1 = \neg l_n$. A loop analysis involves classifying all the cases of loops, and for each one, determining the outcome of a rewrite according the underlying semantics. For the problem at hand, there are only four cases.

Definition 3.2 *Let $S = l_1 \prec^+ l_n$ be a rewrite chain.*

- *If $\neg l_1 = l_n$ or $l_1 = \neg l_n$, then S is called an* odd loop.
- *If $l_1 = l_n$, then*
  - *S is called* positive loop *if l_1 and l_n are both atoms and each literal on $l_1 \prec^+ l_n$ is also an atom;*
  - *S is called* negative loop *if l_1 and l_n are both negative literals and each literal on $l_1 \prec^+ l_n$ is also negative;*
  - *Otherwise, S is called an* even loop.

In all the cases above, l_n is called a loop literal.

Note that when $l_1 = l_n$, and all of l_i, $1 \leq i \leq n$, have the same sign, this sign is either positive or negative, though both types of loops are caused by "positive loops" in the given program. These two types of loops must be treated differently according to the semantics.

It turns out that we only need two rewrite rules to handle all four cases.

<u>**Loop Rules:**</u> Let $g_1 \prec^+ g_n$ be a rewrite chain.

LR1. $g_n \to F$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a positive loop or an odd loop.

LR2. $g_n \to T(\{g_1, ..., g_n\})$
if $g_i \prec^+ g_n$, for some $1 \leq i < n$, is a negative loop or an even loop. □

Apparently, a loop literal should always be rewritten by a loop rule.

Example 3.3 $P_1 = \{b \leftarrow \mathsf{not}\, c;\ c \leftarrow c\}$. *Below, b is proved and $\neg b$ is not.*

$$b \to \neg c \to \neg c \to T(\{b, \neg c\});\ \ \neg b \to c \to c \to F$$

$P_2 = \{d \leftarrow \mathsf{not}\, a;\ a \leftarrow \mathsf{not}\, b;\ b \leftarrow \mathsf{not}\, a\}$. *Both d and $\neg d$ can be proved.*

$$
\begin{array}{l}
d \to \neg a \to b \to \neg a \to T(\{d, \neg a, b\}) \\
\neg d \to a \to \neg b \to a \to T(\{\neg d, a, \neg b\})
\end{array}
$$

$P_3 = \{a \leftarrow \mathsf{not}\, b;\ b \leftarrow \mathsf{not}\, b\}$. *Neither a nor $\neg a$ is proved:*

$$a \to \neg b \to b \to F;\ \ \neg a \to b \to \neg b \to F$$

3.3 Goal rewrite systems and their properties

To summarize, a *rewrite sequence* is a sequence of zero or more rewrite steps $Q_0 \to \ldots \to Q_k$ (denoted $Q_0 \to^* Q_k$) such that Q_0 is an *initial goal* (one involving no context), and for each $0 \leq i < k$, Q_{i+1} is obtained from Q_i by

- literal rewriting at a non-loop literal in Q_i; or
- applying a simplification rule to a subformula in Q_i; or
- applying a loop rule to a loop literal in Q_i.

We may call a subsequence $Q_i \to^* Q_k$ a rewrite sequence in the understanding that it is part of some rewrite sequence $Q_0 \to^* Q_i \to^* Q_k$ from an initial goal Q_0.

Definition 3.4 *A* goal rewrite system *is a triple $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$, where $\mathcal{Q}_L$ is the set of all goals, $\mathcal{R}_P$ is a set of rewrite rules which consists of program rules from $Comp(P)$, the simplification rules, and the loop rules; and $\to$ is the set of all rewrite sequences.*

Goal rewrite systems are like term rewrite systems (see, e.g., [Dershowitz and Jouannaud, 1990]) everywhere except at terminating steps: a terminating step at a subgoal may depend on the history of rewriting.

Two desirable properties of rewrite systems are the properties of *termination* and *confluence*. Rewrite systems that possess both of these properties are called *canonical* systems. A canonical system guarantees that the final result of rewriting from any given goal is unique, independent of any order of rewriting. It therefore allows an implementation to be based on any particular order of rewriting.

Since the simplification system is terminating and literal rewriting only generates non-repeated rewrite chains, it is clear that a goal rewrite system is terminating when the given program is finite. A goal is called a *normal form* if it cannot be rewritten by any rule.

Proposition 3.5 *Let $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ be a goal rewrite system. If P is finite then every rewrite sequence in $\to$ is finite. Further, for any rewrite sequence $Q_0 \to^* Q_k$, if Q_k is a normal form, then either $Q_k = F$ or $Q_k = T(C_1) \vee \ldots \vee T(C_m)$ for some $m \geq 1$.*

Definition 3.6 *A goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ is confluent iff for any rewrite sequences $t_1 \to^* t_2$ and $t_1 \to^* t_3$, there exist $t_4 \in \mathcal{Q}_L$ and rewrite sequences $t_2 \to^* t_4$ and $t_3 \to^* t_4$.*

Theorem 3.7 *Any goal rewrite system $\langle \mathcal{Q}_L, \mathcal{R}_P, \to \rangle$ with a finite P is confluent.*

The goal rewrite systems described here are sound and complete w.r.t. *partial stable models*. These results are special cases of those for the rewrite systems for abduction given in the next section.

4 Rewrite systems for abduction

Let P be a program and A a set of abducibles. An *abducible literals* is either an abducible ϕ or its negative counterpart $\neg\phi$.

The rewriting framework that we defined earlier can be extended for abduction in a straightforward way: the only difference in the extended framework is that we do not apply the Clark completion to abducibles. That is, once an abducible appears in a goal, it will remain there unless it is eliminated by the simplification rule $SR2$ or $SR2'$.

Just like a rewrite to T is written as $T(C)$, where C is the underlying rewrite chain (cf. Section 4.1), a rewrite to an abducible literal l will be written as $l(C)$ for rewrite chain C.

In the following we shall denote by $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the rewrite system obtained by the logic program P and the set A of abducibles. We shall show that it is both sound and complete w.r.t. the partial stable models semantics [Przymusinski, 1990], which is a three-valued generalization of answer set semantics. An answer set of P is also its partial stable model. But sometimes P may not have any answer sets. For instance, if $P = \{p \leftarrow \mathsf{not}\, p;\ q \leftarrow\}$, then P has no answer set, because there is no way to assign a truth value to p. However, P has a partial stable model in which q is true and p is undefined. The following definition is adopted from [You and Yuan, 1995].

Let P be a (ground) program. For any set S of default negations, let $F_P(S) = \{\mathsf{not}\, a \mid P \cup S \not\vdash a\}$, where $\vdash$ is the standard propositional derivation relation with each default negation $\mathsf{not}\, \phi$ being treated as a named atom not_ϕ. A *partial stable model* M of P is defined by a maximal fixpoint S of the function that applies F_P twice, $F_P^2(S) = S$, while satisfying $S \subseteq F_P(S)$, in the following way: for any atom ξ, $\neg\xi \in M$ if $\mathsf{not}\, \xi \in S$, $\xi \in M$ if $P \cup S \vdash \xi$, and ξ is *undefined* otherwise. Notationally, any ξ such that $\xi \notin M$ and $\neg\xi \notin M$ represents that ξ is undefined in M. An *answer set* E (also called a *stable model*) can then be defined as a special case by a fixpoint W such that $F_P(W) = W$ and $E = \{\xi \mid P \cup W \vdash \xi\}$.

Theorem 4.1 *Let P be a finite program, A a set of propositions, and $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ the goal rewrite system w.r.t. P and A.*

Soundness: *For any literal g and any rewrite sequence*

$$g \rightarrow^* [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G,$$

where each l_i is either an abducible literal or T, if $C_1 \cup \cdots \cup C_k$ is consistent, then there exists a partial stable model M of $P \cup \{l_1, ..., l_k\}^+$ such that $g \in \bigcup_{1 \leq j \leq k} C_i \subseteq M$.

Completeness: *For any set of atoms $S \subseteq A$, and any literal g in a partial stable model M of $P \cup S$, there exists a rewrite sequence*

$$g \rightarrow^* [l_1(C_1) \wedge \cdots \wedge l_k(C_k)] \vee G,$$

such that $g \in \bigcup_{1 \leq j \leq k} C_j \subseteq M$, and $S \subseteq \{l_1, ..., l_k\}$.

We say that a program P has no odd loops if there is no odd loop starting with any literals (programs that have no odd loops are also called *call-consistent* [Dung, 1992]). It is well-known that if P has no odd loops, then the partial stable models of P and the answer sets of P coincide. We now show that for any goal, the underlying rewrite system will generate an approximation of a cover for any program P, and exactly a cover when P has no odd loops.

Theorem 4.2 *Let $\langle \mathcal{Q}_L, \mathcal{R}_P, A, \rightarrow \rangle$ be a goal rewrite system. Suppose q is a proposition and*

$$q \rightarrow^* [l_{11}(C_{11}) \wedge \cdots \wedge l_{1k_1}(C_{1k_1})] \vee \ldots \vee [l_{m1}(C_{m1}) \wedge \cdots \wedge l_{mk_m}(C_{mk_m})]$$

is a rewrite sequence such that each l_{ij} is either T or an abducible literal, and $C_{i1} \cup \cdots \cup C_{ik_i}$ is consistent for each i. Then, if P has no odd loops then

$$\{\{l_{11}, \cdots, l_{1k_1}\}, \cdots, \{l_{m1}, \cdots, l_{mk_m}\}\}$$

is a cover of q. In general, for arbitrary P we have

$$\bigvee_{\alpha \in \mathcal{S}} \alpha \supset [l_{11} \wedge \cdots \wedge l_{1k_1}] \vee \cdots \vee [l_{m1} \wedge \cdots \wedge l_{mk_m}]$$

where $\mathcal{S}$ is any cover of q.

Consider again the boat example in Section 1. The Clark completion of $canCross$ is:

$$canCross \equiv (boat \wedge \neg leak) \vee (boat \wedge leak \wedge hasBucket).$$

Since $boat$, $leak$ and $hasBucket$ are abducibles, so rewriting for $canCross$ terminates in one step, and produces the following cover:

$$\{boat \wedge \neg leak,\ boat \wedge leak \wedge hasBucket\}.$$

Notice that the second explanation is not minimal. To get minimal ones, we have to compute prime implicants of the disjunction of explanations in the cover, which are $boat \wedge \neg leak$ and $boat \wedge hasBucket$.

5 Related work

Traditionally, logic programming proof procedures have been defined abstractly in terms of derivation and refutation. Termination has been considered a separate, implementation issue. On the one hand, this separation is possible since the underlying semantics allows the completeness to be stated without resorting to termination. But completeness is rarely guaranteed in an implementation. On the other hand, the separation is also necessary since these procedures deal with non-ground programs for which the problem of loop-checking is undecidable (even for function-free programs [Bol *et al.*, 1991]). For answer set programming, however, loop handling has become a semantic issue: a sound and complete procedure cannot be defined without it. Thus, a distinct feature of our work is a mechanism of loop handling both for termination and for the implementation of the underlying semantics.

Completed programs have been used in query answering in abstract, abductive procedures in [Console *et al.*, 1991; Denecker and Schreye, 1998; Fung and Kowalski, 1997] for non-ground programs with constraints. These procedures are defined for the three-valued completion semantics under the *certainty* mode of reasoning – computing bindings for which an (existential) goal is true in all indented models. In our case the reasoning mode is *brave* – establishing whether a query is true in one of the intended models.

Our work is closely related to another abstract procedure, the Eshghi-Kowalski procedure (EKP) [Eshghi and Kowalski, 1989], which is sound and complete for ground programs under finite-failure three-valued stable models [Giordano *et al.*, 1996]. Besides loop handling and termination, to some extent, one can say that our goal rewriting system (GRS) simulates EKP in a nontrivial way.

1. GRS incurs no backtracking! Backtracking is simulated by rewriting disjunctions, e.g., $F \vee \Phi \rightarrow \Phi$.
2. Loops that go through negation are handled in EKP by nested structures while in GRS by a *flat* structure using rewrite chains.

These features plus loop handling made it possible to formalize our system as a rewriting system benefiting from the known properties of term rewriting in the literature. This also distinguishes our use of rewrite systems from that by [Fung and Kowalski, 1997]. It seems remarkable that a form of non-monotonic reasoning is just rewriting, two areas of research that had little connection previously.

To illustrate these feature, consider the following program

$$\begin{array}{ll} r1.\ g \leftarrow \mathsf{not}\, a & r3.\ b \leftarrow a \\ r2.\ a \leftarrow \mathsf{not}\, b, \mathsf{not}\, c & r4.\ c \leftarrow a \end{array}$$

and the question whether we can prove g. We may answer this question by the following reasoning: To have g we must have $\mathsf{not}\ a$ (r1); To have $\mathsf{not}\ a$ we must have either b or c (r2) which requires having a (r3 and r4). This results in a contradiction. Note that in this reasoning we need to remember what was required previously ($\mathsf{not}\ a$ in this case). This is exactly how the proof is done by GRS

$$g \rightarrow \neg a \rightarrow b \vee c \rightarrow a \vee c \rightarrow F \vee c \rightarrow c \rightarrow a \rightarrow F$$

However, EKP will go through *six* nested levels, and do it twice through backtracking, before the same conclusion can be reached.

Rewriting can be applied to function-free programs for proving ground goals. The idea is that if every derived goal is ground, then all the mechanisms given in this paper apply directly. Obviously, if for every rule in the given program a variable that appears in the body also appears in the head, then a ground goal will be rewritten to another ground goal. *Domain restricted* programs [Niemelä, 1999] can be instantiated only on domain predicates over variables that do not appear in the head so that the resulting non-ground programs also satisfy this requirement. For example, the program given in the next section is domain restricted.

6 Applications and experimental results

We have implemented the writing framework in SWI-Prolog. In the following, we discuss the performance of our implementation on one particular application of abduction in logic programming, which is the problem of computing successor state axioms from a causal action theory [Lin, 2000].

Consider a logistics domain in which we have a truck and a package. We know that the truck and the package can each be at only one location at any given time, and that if the package is in the truck, then when the truck moves to a new location, so is the package. Suppose that we have the following propositions: $\mathrm{ta}(x)$ – the truck is at location x initially; $\mathrm{pa}(x)$ – the package is at location x initially; in – the package is in the truck initially; $\mathrm{ta}(x, y, z)$ – the truck is at location x after the action of moving it from y to z is performed; $\mathrm{pa}(x, y, z)$ – the package is at location x after the action of moving the truck from y to z is performed; and $\mathrm{in}(y, z)$ – the package is in the truck after the action of moving the truck from y to z is performed. We then have the following logic program:

$$\begin{array}{l} \mathrm{ta}(X, X1, X). \\ \mathrm{pa}(X, X1, X2) \leftarrow \mathrm{ta}(X, X1, X2), \mathrm{in}(X1, X2). \\ \mathrm{ta}(X, X1, X2) \leftarrow X \neq X2, \mathrm{ta}(X), \mathsf{not}\ \mathrm{taol}(X, X1, X2). \\ \mathrm{taol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{ta}(Y, X1, X2). \\ \mathrm{pa}(X, X1, X2) \leftarrow \mathrm{pa}(X), \mathsf{not}\ \mathrm{paol}(X, X1, X2). \\ \mathrm{paol}(X, X1, X2) \leftarrow Y \neq X, \mathrm{pa}(Y, X1, X2). \\ \mathrm{in}(X, Y) \leftarrow \mathrm{in}. \end{array}$$

The first rule is the effect axiom. The second rule is a causal rule which says that if a package is in the truck, then the package should be where the truck is. The rest are frame axioms. For instance, the third one is the frame axiom about ta, with the help of a new predicate taol: if the truck is initially at X, and if one cannot prove that it will be elsewhere after the action is performed, then it should still be at X.

As one can see, the above program, when fully instantiated over any given finite set D of locations, has no odd loops. So our rewrite system will generate a cover for any query. Note that in the program we have omitted domain predicate $loc(Y)$ for each variable Y in the body of a rule (all the variables in the program refer to locations). Thus, the program is domain restricted and needs only to be instantiated for the variable Y in the fourth and sixth rules over the domain of locations.

Now let the set A of abducibles be the following set:

$$\{\mathrm{in}\} \cup \{\mathrm{pa}(x), \mathrm{ta}(x) \mid x \in D\}.$$

The following table shows some of the results for $D = \{1, 2, 3, 4\}$:[2]

Query	Result	Time
$\mathrm{ta}(1, 2, 3)$	false	0.0
$\mathrm{ta}(3, 2, 3)$	true	0.0
$\mathrm{pa}(1, 2, 3)$	$\mathrm{pa}(1) \wedge \neg\mathrm{in}$	0.05
$\neg\mathrm{pa}(1, 2, 3)$	$\neg\mathrm{pa}(1) \vee \mathrm{in} \vee \mathrm{pa}(2) \vee \mathrm{pa}(3) \vee \mathrm{pa}(4)$	0.2
$\mathrm{pa}(2, 2, 3)$	$\mathrm{pa}(2) \wedge \neg\mathrm{in}$	0.08
$\neg\mathrm{pa}(2, 2, 3)$	$\neg\mathrm{pa}(2) \vee \mathrm{in} \vee \mathrm{pa}(1) \vee \mathrm{pa}(3) \vee \mathrm{pa}(4)$	0.1
$\mathrm{pa}(3, 2, 3)$	$\mathrm{pa}(3) \vee \mathrm{in}$	0.25
$\neg\mathrm{pa}(3, 2, 3)$	$\neg\mathrm{in} \wedge \neg\mathrm{pa}(3) \vee \neg\mathrm{in} \wedge \mathrm{pa}(1) \vee$ $\neg\mathrm{in} \wedge \mathrm{pa}(2) \vee \neg\mathrm{in} \wedge \mathrm{pa}(4)$	0.1

For instance, the row on $\mathrm{pa}(1, 2, 3)$ says that for it to be true, the package must initially be at 1 and cannot be inside the truck (otherwise, it would be moved along with the truck), and the computation took 0.1 seconds.The row on $\mathrm{pa}(3, 2, 3)$ says that for it to be true, either the package was initially at 3 or it was inside the truck. The outputs for larger Ds are

[2] On a PIII 1GHz PC with 512MB RAM running SWI-Prolog 3.2.9.

similar. The performance varies for different queries. For simple queries like ta$(1, 2, 3)$, their covers can be computed almost in constant time. The hardest one is for pa$(3, 2, 3)$ which took 25 minutes when $|D| = 7$.

It is interesting to compare our system with an alternative for computing the cover of a query. As we mentioned in Section 2, the set of abductive explanations according to Kakas and Mancarella is actually a cover. One way of computing these abductive explanations is to add, for each proposition $p \in A$, the following two clauses ([Satoh and Iwayama, 1991]): $p \leftarrow \mathsf{not}\,\neg p$ and $\neg p \leftarrow \mathsf{not}\,p$ into the original program, and use the fact that there will be a one to one correspondence between abductive explanations of q under the original program and answer sets of the new program that contain q. So one can use an answer set generator, for example **smodel** or **dlv**, to compute a cover of query by generating all the answer sets in the new program that contain the query. However, the problem here is that there are too many such answer sets in this case. For instance, suppose there are n locations, then the number of answer sets that contain any particular query is in the order of 2^{2n}, roughly one half of the number of complete hypotheses, even for a very simple query like ta$(1, 2, 3)$. We do not know at the moment if there is any efficient way of using an answer set generator to compute a cover set of a query.

7 Future work

There are several directions for extending this work. One of them is to consider rewriting for non-ground logic programs for some restricted yet decidable classes of non-ground goals. Another important one is to extend this to programs with constraints of the form:

$$\perp \leftarrow a_1, ..., a_i, \mathsf{not}\,b_1, ..., \mathsf{not}\,b_n$$

Our new definition of abduction can be extended to include these constraints straightforwardly. The challenge is in extending our rewriting system accordingly.

8 Acknowledgements

We would like to thank Ilkka Niemelä for helpful discussions related to the topics of this paper, and Ken Satoh for comments on earlier version of this paper. The first author's work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grant HKUST6145/98E. The work by the second author was carried out mainly during his visits to Hong Kong University of Science and Technology and the Institute of Information Science, Academia Sinica in Taiwan.

References

[Bol *et al.*, 1991] R. Bol, A. Krzysztof, and J. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–39, 1991.

[Console *et al.*, 1991] L. Console, D. Theseider, and P. Porasso. On the relationship between abduction and deduction. *J. Logic Programming*, 2(5):661–690, 1991.

[Denecker and Schreye, 1998] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for normal abductive programs. *J. Logic Programming*, 34(2):111–167, 1998.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990.

[Dung, 1992] P.M. Dung. On the relation between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[Eshghi and Kowalski, 1989] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In *Proc. 6th ICLP*, pages 234–254. MIT Press, 1989.

[Fung and Kowalski, 1997] T. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–164, 1997.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[Giordano *et al.*, 1996] L. Giordano, A. Martelli, and M. Sapino. Extending negation as failure by abduction: A three-valued stable model semantics. *J. Logic Programming*, 26(1), 1996.

[Kakas and Mancarella, 1990] A. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conf. for AI*, 1990.

[Konolige, 1992] K. Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992.

[Kunen, 1989] K. Kunen. Signed data dependencies in logic programs. *J. Logic Programming*, 7(3):231–245, 1989.

[Lin, 2000] F. Lin. From causal theories to successor state axioms: bridging the gap between nonmonotonic action theories and STRIPS-like formalisms. In *Proc. AAAI '00*, 2000.

[Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. and AI*, 25(3-4):241–273, 1999.

[Poole, 1988] D. Poole. Representing knowledge for logic-based diagnosis. In *Proc. Fifth Generation Computer Systems Conference*, pages 1282–1290, 1988.

[Przymusinski, 1990] T.C. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In *Proc. 7th ICLP*, pages 459–477. MIT Press, 1990.

[Reiter and de Kleer, 1987] R. Reiter and J. de Kleer. Foundations of ATMS. In *Proc. AAAI87*, 1987.

[Satoh and Iwayama, 1991] K. Satoh and N. Iwayama. Computing abduction using the tms. In *Proc. the 8th International Conference on Logic Programming*, 1991.

[Satoh and Iwayama, 1992] K. Satoh and R. Iwayama. A query evaluation method for abductive logic programming. In *Proc. JICSLP'92*, 1992.

[You and Yuan, 1995] J. You and L. Yuan. On the equivalence of semantics for normal logic programs. *J. Logic Programming*, 22:212–221, 1995.