Knowledge Assimilation and Proof Restoration through the Addition of Goals*

Hisashi Hayashi

Department of Computer Science
Queen Mary and Westfield College
University of London
Mile End Road, London E1 4NS, U.K.
email: hisashi@dcs.qmw.ac.uk

Abstract

Normal proof procedures in abductive logic programming assume that a given program does not change until the proof is completed. However, while a proof is being constructed, new knowledge which affects the proof might be acquired. This paper addresses two important issues: 1. How is it confirmed that the proof being constructed is not affected by the addition of a clause? 2. If affected, how are the invalid parts of the proof restored? The abductive proof procedure used in this paper is Kakas and Mancarella's procedure and is extended to prepare for proof checking and proof restoration. It is shown that any invalid part of a proof can be restored if some additional goals are solved. These additional goals can be added before a proof is completed.

Keywords: Abduction, Logic Programming, Knowledge Acquisition.

1 Introduction

In most literature on logic programming, the design of a proof procedure does not take into account the dynamic nature of a program. If this kind of proof procedure is used, every time the program is revised and changed, the goal has to be proved again under the new program. However, it is easy to imagine that

*This paper will appear in the proceedings of AIMSA'98 which will be published by Springer-Verlag to which the copyright of this contibution will be moved. The Springer-Verlag LNCS series homepage is `http://www.springer.de/comp/lncs/index.html`

if the program is changed only a little bit, the old proof is still valid in a lot of cases.

The purpose of this project is to reuse the same proof unless it becomes invalid and to change only limited parts of the proof if the proof becomes invalid. Indeed in a lot of cases, it takes time to make a proof. When a whole proof or a part of a proof needs to be constructed within limited time, it is not a good idea to make a brand new proof from the beginning every time programs are corrected.

Consider planning in robotics. Suppose that a robot is always gathering information through its sensors and that the database of the robot is always changing.

Suppose that the robot starts constructing a plan at time, say, 10:00:00 and finishes at 10:01:15. If the plan is made based only on the information at 10:00:00, it is impossible to say that the plan is still valid at 10:01:15 because the database might be updated between 10:00:00 and 10:01:15. This means that the robot is not sensing the outer world while making the plan. (Even if the robot senses the outer world, it is not using new information during that time.)

For this reason, some criteria are needed by which to confirm that the plan is still valid after updating the database. Also it is essential to reuse some parts of the plan to avoid reconstructing a brand new plan and save time.

In *partial order planning*, *causal links* or *protected links* are used for replanning. (See standard textbooks such as [20].) In this paper, a more general theory which can be applied to any proof in abductive logic programming will be introduced. Of course, it is assumed that the contents of the databases (or programs) being used are updated as the time goes on.

The proof procedure discussed in the present paper is for abductive logic programming. The abductive proof procedure adopted in this paper is based on the Eshghi-Kowalski (E-K) procedure [4] and the Kakas-Mancarella (K-M) procedure [11, 12]. The E-K procedure is an extension of SLD resolution and the K-M procedure is an extension of the E-K procedure.

The rest of the paper is organised as follows. In Section 2, abductive logic programming is introduced briefly. In Section 3, the K-M procedure is introduced. The procedure is extended to prepare for the expansion of the program. Based on the additional information which is obtained by the extended K-M procedure, a proof restoration procedure is introduced in Section 4 which is the most important section. This proof restoration procedure can also check the validity of a proof. After showing a result of experiments in Section 5 and the related works are discussed in Section 6, the conclusion is discussed in Section 7.

2 Abductive Logic Programming

Before defining an abductive framework, some basic words are defined. It is assumed that the readers of this paper are familiar with the concepts of logic programming.

In this paper, variables are expressed by letters and numerals starting with an upper case letter. Constants, predicate symbols, and function symbols are expressed by letters and numerals starting with a lower case letter. Intuitively, the negation *not* is so called *negation as failure* but it is not, strictly speaking, in the sense that negative literals are treated as hypotheses.

Definition 2.1 *A* **literal** *is either a positive literal or a negative literal. A* **positive literal** *is an atom. A* **negative literal** *is of the form not P where P is an atom. The* **contrary** *of a positive literal P is the negative literal not P. The contrary of a negative literal not P is the positive literal P. The contrary of a literal L is expressed as L^*.*

Definition 2.2 *A* **clause** *is of the form $L \Leftarrow L_1, ..., L_n$ where L is an atom and $L_1, ..., L_n$ are literals. The clause $L \Leftarrow$ can be expressed as L.*

Note that $\Leftarrow$ is used only for clauses of logic programs and $\leftarrow$ refers to the implication of classical logic.

Definition 2.3 *An* **abductive framework**[1] *is a tuple $\langle P, Ab \rangle$, where P is a* **program***, a set of clauses and Ab is a set of* **abducibles***, a set of literals, such that P does not include a clause of the form $L \Leftarrow L_1, ..., L_n$ such that L is an element of Ab. An abducible is a* **positive abducible** *if it is a positive literal. An abducible is a* **negative abducible** *if it is a negative literal.*

To avoid transforming the negative literal *not* p to the positive literal $neg(p)$ which is called a **non-base abducible** in [11], all negative literals whose atoms are mentioned in P or Ab are normally regarded as abducibles unless otherwise mentioned.

As far as the semantics of the procedure in the present paper is concerned, the completion semantics is used. The completion of the program P and abducibles Ab is defined as follows.

Definition 2.4 *Given a program P and a set of literals Ab (abducibles), $comp(P, Ab)$ is the least set such that for any positive literal q which is not in Ab,*

- *all the clauses in P defining q are:*

$$q \Leftarrow L_{1,1}, ..., L_{1,n_1} \ \dots \ q \Leftarrow L_{m,1}, ..., L_{m,n_m}$$

if and only if

- *$comp(P, Ab)$ includes:*

$$q \leftrightarrow (L_{1,1} \wedge ... \wedge L_{1,n_1}) \vee ... \vee (L_{m,1} \wedge ... \wedge L_{m,n_m})$$

[1] Integrity constraints are not included for simplicity.

3 An Abductive Proof Procedure

In this section, the abductive proof procedure based on the procedure introduced by Kakas and Mancarella [11, 12] is introduced. The Kakas-Mancarella (K-M) proof procedure is an extension of the Eshghi-Kowalski (E-K) proof procedure [4] which simulates SLDNF by abduction. The E-K procedure is an extension of SLD resolution.

In the original definition, an abductive derivation and a consistency derivation are defined separately. In the following definition, however, they are defined as a single derivation. Although the following definition is the same as the original definition in essence, a new set called defending set is used to prepare for the expansion of the given program. Recording additional information in a defending set does not affect the proof procedure at all. Therefore, the following proof procedure works in the same way as the original one.

Definition 3.1 *The* **resolvent** *of the set of literals* $\{P, L_1, ..., L_n\}$ *on* P *by the clause* $P \Leftarrow L_{n+1}, ..., L_m$ *is the set of literals* $\{L_1, ..., L_m\}$.

Definition 3.2 *A* **goal** *is either of the form* (pos, Ls) *where* Ls *is a set of literals, or of the form* (neg, Cs) *where* Cs *is a set of sets of literals.*

Intuitively, $(pos, \{L_1, ..., L_n\})$ means that the goal is to prove $L_1 \wedge ... \wedge L_n$ and $(neg, \{\{L_{1,1}, ..., L_{1,n_1}\}, ..., \{L_{m,1}, ..., L_{m,n_m}\}\})$ means that the goal is to prove $\neg(L_{1,1} \wedge ... \wedge L_{1,n_1}) \wedge ... \wedge \neg(L_{m,1} \wedge ... \wedge L_{m,n_m})$.

Definition 3.3 *A* **goal list** *is of the form* $[G_1, ..., G_n]$ *where* $G_1, ..., G_n$ $(n \geq 0)$ *are goals.*

Intuitively, the goals are proved in this order. (i.e. from G_1 to G_n) Note that [] is one of goal lists.

Definition 3.4 *A* **defending set** *is a set whose elements are of the form* (L, Ls) *where* L *is a literal and* Ls *is a set of literals.*

Intuitively, the element of a defending set (L, Ls) means that the set of literals $\{L\} \cup Ls$ (which means that $\leftarrow L \wedge Ls$ has to be proved) and the positive literal $L (\in \{L\} \cup Ls)$ were selected when the consistency derivation rule c1 was applied. (The consistency derivation rules will be defined soon.) The procedure tried to prove that L does not hold and if L is proved to be true, one of the literals in LS has to be proved to be false. Therefore, when a clause defining L is added to the program, the defending sets have to be checked.

Definition 3.5 *A* **goal set** *is of the form* (GL, Δ, R) *where* GL *is a goal list,* Δ *is a set of literals (abducibles), and* R *is a defending set*[2].

[2]In [7], a set of defending sets is called *Reject* and R is used often to refer to a set of defending sets.

The goal set (GL, Δ, R) means that all the goals in GL have to be proved with the abducibles in Δ assumed to be true. R will be used to restore the correctness when a clause is added to the program.

Definition 3.6 *A* **derivation** *from* (G_1, Δ_1, R_1) *to* (G_n, Δ_n, R_n) *under the abductive framework* $\langle P, Ab\rangle$ *is a sequence of goal sets:*

$$(GL_1, \Delta_1, R_1), ..., (GL_n, \Delta_n, R_n)$$

such that for each $i (1 \le i \le n-1)$, GL_i *is of the form* $[F_{i,1}, ..., F_{i,x_i}]$ $(x_i > 0)$, *if* $F_{i,1}$ *is of the form* $(pos, \{L_1, ..., L_k\})$ $(k > 0)$ *and* $L_u (\in \{L_1, ..., L_k\})$ *is selected, then* $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ *is obtained by one of the following* **abductive derivation rules***,*

a1 *If* L_u *is not an abducible in* Ab *and a clause in* P *whose head is* L_u *is chosen, then* $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(pos, Ls), F_{i,2}, ..., F_{i,x_i}]$ *where* Ls *is the resolvent of* $\{L_1, ..., L_k\}$ *on* L_u *by the chosen clause.*

a2 *If* L_u *is an abducible in* Ab *and* $L_u \in \Delta_i$, *then* $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(pos, Ls), F_{i,2}, ..., F_{i,x_i}]$ *where* $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.

a3 *If* L_u *is a positive abducible in* Ab, $L_u \notin \Delta_i$, *and not* $L_u \notin \Delta_i$, *then* $\Delta_{i+1} = \{L_u\} \cup \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(pos, Ls), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ *where* $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.

a4 *If* L_u *is a negative abducible in* Ab, $L_u \notin \Delta_i$, *and* $L_u^* \notin \Delta_i$, *then* $\Delta_{i+1} = \{L_u\} \cup \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(neg, \{\{L_u^*\}\}), (pos, Ls), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ *where* $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.

else if $F_{i,1}$ *is of the form* $(neg, \{C_1, ..., C_k\})$ $(k > 0)$, $C_v (\in \{C_1, ..., C_k\})$ *is selected,* C_v *is of the form* $\{L_{v,1}, ..., L_{v,y}\}$ $(y > 0)$, *and* $L_{v,l} (\in \{L_{v,1}, ..., L_{v,y}\})$ *is selected, then* $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ *is obtained by one of the following* **consistency derivation rules***,*

c1 *If* $L_{v,l}$ *is not an abducible in* Ab, *then* $\Delta_{i+1} = \Delta_i$, $R_{i+1} = \{(L_{v,l}, \{L_{v,1}, ..., L_{v,l-1}, L_{v,l+1}, ..., L_{v,y}\})\} \cup R_i$, *and* $GL_{i+1} = [(neg, Cs \cup \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ *where* Cs *is the set*[3] *of all the resolvents of* C_v *each of which is obtained by resolving* C_v *on* $L_{v,l}$ *by a clause in* P.

c2 *If* $L_{v,l}$ *is an abducible in* Ab *and* $L_{v,l} \in \Delta_i$, *then* $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_v \setminus \{L_{v,l}\}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.

c3 *If* $L_{v,l}$ *is an abducible in* Ab *and* $L_{v,l}^* \in \Delta_i$, *then* $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, *and* $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.

[3] This can be an empty set.

c4 *If $L_{v,l}$ is a positive abducible in Ab, $L_{v,l} \notin \Delta_i$, and not $L_{v,l} \notin \Delta_i$, then $\Delta_{i+1} = \{not\ L_{v,l}\} \cup \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.*

c5 *If $L_{v,l}$ is a negative abducible in Ab, $L_{v,l} \notin \Delta_i$, and $L^*_{v,l} \notin \Delta_i$, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(pos, \{L^*_{v,l}\}), (neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.*

and else if $F_{i,1}$ is of the form $(X, \emptyset)$, where X is either pos or neg, then $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ is obtained by the following **controlling derivation rule**.

cdr $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, $GL_{i+1} = [F_{i,2}, ..., F_{i,x_i}]$

From a derivation, it is possible to trace the proof construction. Abductive derivation rules try to prove that the selected literal is true. Consistency derivation rules try to prove that the selected literal is false or at least one literal in the selected set of literals is false.

For *safety* reason, in the predicate case, it is prohibited to select a nonground abducible when one of the rules [a2], ..., [a4], [c2], ..., [c5] is applied. This restriction prevents so called "floundering". Note that even if the selected (non-abducible) literal is not ground, [a1] and [c1] can be applied by using *unification*.

Definition 3.7 *A derivation from $([(pos, \{L_1, ..., L_n\})], \emptyset, \emptyset)$ to $([], \Delta_m, R_m)$ under the abductive framework $\langle P, Ab \rangle$ is a* **(complete) proof** *for $L_1 ... L_n$ under the abductive framework $\langle P, Ab \rangle$*

Definition 3.8 *A derivation from $([(pos, \{L_1, ..., L_n\})], \emptyset, \emptyset)$ to (GL_m, Δ_m, R_m) under the abductive framework $\langle P, Ab \rangle$ is an* **incomplete proof** *for $L_1 ... L_n$ under the abductive framework $\langle P, Ab \rangle$.*

Note that complete proofs are the limiting case of incomplete proofs.

Example 3.1 Consider the following program where negative literals, $ab(V)$, and $super(W)$ are abducibles.

clause 1: $fly(X) \Leftarrow bird(X), not\ ab(X)$
clause 2: $fly(penguin) \Leftarrow super(penguin)$
clause 3: $bird(penguin)$

A proof for $not\ fly(penguin)$ is as follows.

ab 1 $?not\ fly(penguin)$

co 1.1 $?\{fly(penguin)\}$ $(not\ fly(penguin) \in \Delta)$

co 1.2 $?\{bird(penguin), not\ ab(penguin)\}, \{super(penguin)\}$
$((fly(penguin), \emptyset) \in R)$

co 1.3 $?\{not\ ab(penguin)\}, \{super(penguin)\}$
$((bird(penguin), \{not\ ab(penguin)\}) \in R)$

ab 1.3.1 $?ab(penguin)$

ab 1.3.2 success $(ab(penguin) \in \Delta)$

co 1.4 $?\{super(penguin)\}$

co 1.5 failure $(not\ super(penguin) \in \Delta)$

ab 2 success

4 Clause Assimilation and Proof Restoration

When or after a proof is constructed by the abductive proof procedure shown in the previous section, if a new clause is added to the given program, the validity of the complete or incomplete proof is no longer guaranteed. However, by proving additional goals, the validity of the proof can be restored.

Definition 4.1 An **added goal** by checking the clause $L \Leftarrow L_1, ..., L_n$ and the set of defending sets R is a set of literals of the form $\{L_1, ..., L_m\}$ such that $(L, \{L_{n+1}, ..., L_m\}) \in R$.

Theorem 4.1 *For any incomplete proof for $L_1, ..., L_k$ from $([(pos, \{L_1, ..., L_k\})], \emptyset, \emptyset)$ to $([F_{t,1}, ..., F_{t,n}], \Delta_t, R_t)$ under the abductive framework $\langle P, Ab\rangle$ and for any clause C which does not define an atom in Ab, if there exists a derivation from $([(neg, AGs), F_{t,1}, ..., F_{t,n}], \Delta_t, R_t)$ to $([], \Delta_s, R_s)$ under the abductive framework $\langle P \cup \{C\}, Ab\rangle$, where AGs is the set[4] of all the added goals by checking C and R_t, then $comp(P \cup \{C\}, Ab) \cup \Delta_s \models \{L_1, ..., L_k\}$.*

Proof[5]*:* The only derivation rule which might be affected by the addition of a clause to a program is the consistency derivation rule c1. The application of c1 is affected by the addition of a clause to a program if and only if the clause defines an atom which was selected when c1 was applied.

Suppose that the set of literals $\{P, Q_1, ..., Q_n\}$ was selected and the literal P was selected from the set when c1 was applied. After the application of c1, $\{P, Q_1, ..., Q_n\}$ was replaced with the set of all the resolvents each of which is a resolvent of $\{P, Q_1, ..., Q_n\}$ on P by a clause in the program. This information is recorded in R_t because $(P, \{Q_1, ..., Q_n\}) \in R_t$ If a clause of the form $P \Leftarrow Q_{n+1}, ..., Q_m$ is

[4]This set is empty if the given (incomplete) proof is not affected by the assimilation of the clause.

[5]More precise proof is in the full paper.

added to the program, the only thing which has to be done is to prove the goal $(neg, \{\{Q_1, ..., Q_m\}\})$. This is done by adding (neg, AGs) to the goal list.

Example 4.1 Consider the following program where negative literals are abducibles.

clause 1: $innocent(X) \Leftarrow not\ guilty(X)$
clause 2: $guilty(X) \Leftarrow law(L), against(X, L)$
clause 3: $law(eu)$
clause 4: $law(uk)$
clause 5: $against(a, japan)$

A proof for $innocent(a)$ is shown below.

ab 1 $?innocent(a)$

ab 2 $?not\ guilty(a)$

co 2.1 $?\{guilty(a)\}$ $(not\ guilty(a) \in \Delta)$

co 2.2 $?\{law(L), against(a, L)\}$ $((guilty(a), \emptyset) \in R)$

co 2.3 $?\{against(a, eu)\}, \{against(a, uk)\}$ $((law(L), \{against(a, L)\}) \in R)$

co 2.4 $?\{against(a, uk)\}$ $((against(a, eu), \emptyset) \in R)$

co 2.5 failure $((against(a, uk), \emptyset) \in R)$

ab 3 success

If $law(un)$ is added to the program after co 2.3 is derived, this proof becomes invalid and this invalidity is detected because $law(L)$ is recorded in R. The proof is restored by proving $\leftarrow against(a, un)$ under the updated program.

If $law(japan)$ is added to the program after co 2.3 is derived, this proof becomes invalid and this invalidity is detected because $law(L)$ is recorded in R. This proof cannot be restored because $\leftarrow against(a, japan)$ cannot be proved.

5 Experiments

In the case of depth first search, as long as the added clause is not chosen when applying the abductive derivation rule a1, it has been confirmed by experiments that the proof restoration method in Theorem 4.1 is generally faster than the naive method which restarts constructing a proof from the beginning when the program is updated. This is due to the fact that the procedure in the present paper has pruned some branches of the search tree when the naive method restarts constructing a proof from scratch. The proof restoration method in the present paper just adds additional branches which affects the validity of proofs. The exceptional case is the case where it takes little time before a clause is added to the program. In this case, our procedure is nearly the same as the naive method as far as the time to construct a proof is concerned. One of the results of the experiments is shown below.

Example 5.1 Consider the following program in Prolog.

```
cl(hold(F,Tnext), [given+(F,T)]):-Tnext>0, T is Tnext-1.
cl(hold(F,Tnext), [hold(F,T), not(broken(F,Tnext))]):-
   Tnext>0, T is Tnext-1.
cl(broken(pos(F),Tnext), [given+(neg(F),T)]):-T is Tnext-1.
cl(broken(neg(F),Tnext), [given+(pos(F),T)]):-T is Tnext-1.
cl(given+(neg(alive),T), [do(shoot,T), hold(pos(loaded),T)]).
cl(given+(pos(loaded),T), [do(load,T)]).
cl(given+(pos(alive),0), []).
cl(given+(neg(loaded),0), []).
cl(do(load,1000), []).
cl(do(shoot,2000), []).
```

The compiler reads the above program and regards a clause of Prolog of the form:

$$\mathtt{cl}(\mathtt{A}, [\mathtt{L_1}, ..., \mathtt{L_n}]) : -\mathtt{C_1}, ..., \mathtt{C_m}.$$

as the clause of an abductive logic program:

$$A \Leftarrow L_1^{km}, ..., L_n^{km}$$

if $\mathtt{C_1}, ..., \mathtt{C_m}$ hold where for each i $(1 \leq i \leq m)$, if $\mathtt{L_i}$ is of the form $\mathtt{not(F)}$, L_1^{km} is *not* F, else if $\mathtt{L_i}$ is of the form $\mathtt{F}$, L_1^{km} is F. The only abducibles are negative literals.

The program[6] expresses so called the Yale Shooting Problem. At first (at time 0), the turkey is "alive" and the gun is not "loaded". The action "load" makes the gun loaded. When the gun is "loaded", if someone "shoots" the turkey, the turkey is not "alive" afterwards. The action "load" is taken at time 1000. The action "shoot" is taken at time 2000.

given+(pos(F),T) means that F becomes true immediately after T and it continues to be true, by default, until it becomes false. *given+(neg(F),T)* means that F becomes false immediately after T and it continues to be false until it becomes true.

The test query shown below is if the turkey is "not alive" at time 4000. This query will succeed unless the program is changed.

[6]This program is made so that it takes a lot of time to construct a proof.

$? \, hold(neg(alive), 4000)$

After applying (abductive or consistency or controlling) derivation rules 10000 times, the following clause[7] is added to the abductive logic program.

$given{+}(neg(loaded), T) \Leftarrow do(unload, T)$

This does not affects the truth value of the query because "$do(unload, T)$" does not hold for any T.

It took 856.116699 cpu-time[8] for the naive method to answer the query while our method answered in 579.649963 cpu-time. While the naive method applied derivation rules 31992 times, our procedure applied derivation rules 22242 times.

6 Related Works

Truth maintenance (TM) systems have close relationships with the semantics of abductive logic programming as is surveyed in [10]. The TM system records what have been calculated and reuses them afterwards. Even if the database is updated, only limited parts of the records are corrected. There are mainly two TM systems, the justification-based TM system (JTMS) [2] and the assumption-based TM system (ATMS) [1], both of which are propositional logic. The JTMS records only one set of assumptions whereas the ATMS records different sets of assumptions at one time. Although the ATMS cannot use negation in justifications which correspond to clauses in logic programming, the JTMS can use negation corresponding to negation as failure. The relationships between the JTMS and abductive logic programming are discussed in [3, 6, 12]. The relationships between the ATMS and abductive logic programming are discussed in [18]. A proof procedure to compute generalised stable models was developed [21] using the JTMS. An abductive proof procedure for the ATMS was developed in [8]. Non-monotonic extensions of the ATMS were developed in [19, 9]. A Prolog-like theorem prover for predicate SLD resolution based on the TM systems was developed in [22, 23]. None of the above TM systems allows updates of justifications during the calculation.

A number of abductive planners [15, 17, 14, 25] use the cycle procedure [13] to cope with dynamic environments. It is straightforward to combine the cycle procedure with the procedure in the present paper. In [15, 17, 14], definitions of predicates cannot be changed. Although the treatment of undefined predicates is written in [15], the occurrences of "observation predicates" are restricted to integrity constraints. In [25], when the invalidity of a plan is detected, the plan is constructed from the beginning.

[7]This clause corresponds to "`cl(given+(neg(loaded), T), [do(unload, T)]).`" in the above Prolog program. This Prolog clause can be added by using the built-in predicate "`assert`".

[8]The Prolog compiler used in this experiment is ECLiPSe.

7 Conclusions and Future Works

In Section 4, it was shown that even if some parts of the complete or incomplete proof become invalid, these invalidities can be detected and the validity can be restored by proving added goals. The extended K-M procedure and the proof restoration procedure have been tested using Prolog as is shown in Section 5. By the experiments, it was found that when the K-M procedure spends long time before a clause is added to the program, our procedure can save a lot of time in the depth first search if the added clause is not chosen when the abductive derivation rule a1 is applied.

Various planning algorithms [5, 16, 24] use abduction. As a future work, the algorithm in the current paper will be applied to planning so that it can replan efficiently.

Acknowledgements

I am grateful to Dr. Murray Shanahan for discussion.

References

[1] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[2] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[3] P. M. Dung. An abductive foundation for non-monotonic truth maintenance. In *World Conference on Fundamentals of Artificial Intelligence*, 1991.

[4] E. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In *International Conference on Logic Programming*, pages 234–254, 1989.

[5] K. Eshghi. Abductive planning with event calculus. In *International Conference and Symposium on Logic Programming*, pages 562–579, 1988.

[6] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In *International Conference on Logic Programming*, pages 421–441, 1990.

[7] H. Hayashi. Abductive proofs in dynamic databases. Technical Report 744, Department of Computer Science, Queen Mary and Westfield College, University of London, 1997.

[8] K. Inoue. An abductive procedure for the CMS/ATMS. In *ECAI90 International Workshop on Truth Maintenance*, 1990.

[9] U. Junker. A correct non-monotonic ATMS. In *International Joint Conference on Artificial Intelligence*, pages 1049–1054, 1989.

[10] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5, 1997.

[11] A. C. Kakas and P. Mancarella. Database updates through abduction. In *International Conference on Very Large Databases*, pages 650–661, 1990.

[12] A. C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Pacific Rim International Conference on Artificial Intelligence*, pages 438–443, 1990.

[13] R. Kowalski. Using meta-logic to reconcile reactive with rational agents. In *Meta-Logic and Logic Programming*, pages 227–242, 1995.

[14] R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. Department of Computer Science, Imperial College, University of London, 1996.

[15] R. A. Kowalski and F. Sadri. An agent architecture that unifies rationality with reactivity. Department of Computer Science, Imperial College, University of London, 1997.

[16] L. R. Missiaen, M. Denecker, and M. Bruynooghe. CHICA, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, 1995.

[17] J. A. Dávila Quintero. *Agents in logic programming*. PhD thesis, Department of Computer Science, Imperial College, University of London, 1997.

[18] R. Reiter and J. deKleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *AAAI87*, pages 183–188, 1987.

[19] W. L. Rodi and S. G. Pimentel. A non-monotonic ATMS using stable bases. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 485–495, 1991.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[21] K. Satoh and N. Iwayama. Computing abduction using the TMS. In *International Conference on Logic Programming*, pages 505–518, 1990.

[22] M. Shanahan. *Exploiting dependencies in search and inference mechanisms.* PhD thesis, King's College, University of Cambridge, 1987.

[23] M. Shanahan. An incremental theorem prover. In *International Joint Conference on Artificial Intelligence*, pages 987–989, 1987.

[24] M. Shanahan. Event calculus planning revisited. In *European Conference on Planning*, pages 390–402, 1997.

[25] M. Shanahan. Reinventing shakey. In *Working Notes of the AAAI Fall Symposium on Cognitive Robotics*, to appear.

Appendix

In this paper, the deletion of clauses from a program is not considered. However, the deletion of clauses can be simulated by the addition as follows. The program P is changed to $P!$ for this purpose.

Definition 7.1 *Given a set of clauses P, $P!$ is the least set of clauses such that for each clause C in P:*

$$a \Leftarrow L_1, ..., L_n$$

$P!$ contains the clause $C!$:

$$a \Leftarrow L_1, ..., L_n, not\ x$$

where x is a new atom called a **deletion atom** *which does not occur in any other clause in $P!$. $C!$ is called the* **corresponding clause** *of C.*

Similarly, the addition of a clause is as follows.

Definition 7.2 *Whenever a clause C:*

$$a \Leftarrow L_1, ..., L_n$$

where $a, L_1, ..., L_n$ mention none of the deletion atoms of clauses in $P!$, is added to the program P, the clause $C!$:

$$a \Leftarrow L_1, ..., L_n, not\ x$$

is added to $P!$ where x is a new atom called a **deletion atom** *which does not occur in any other clause in $P!$. $C!$ is called the* **corresponding clause** *of C.*

The deletion of a clause can be simulated by the addition of the deletion atom of the corresponding clause.

Definition 7.3 *Whenever a clause C is removed from the program P, the clause x is added to the program $P!$ where x is the deletion atom of the corresponding clause of C.*

Example 7.1 *Consider the following program P.*

clause 1: $fly(X) \Leftarrow bird(X), not\ ab(X)$
clause 2: $fly(penguin) \Leftarrow super(penguin)$
clause 3: $bird(penguin)$

$P!$ is as follows.

clause 1!: $fly(X) \Leftarrow bird(X), not\ ab(X), not\ del_1$
clause 2!: $fly(penguin) \Leftarrow super(penguin), not\ del_2$
clause 3!: $bird(penguin) \Leftarrow not\ del_3$

None of del_1, del_2, del_3 holds and all of not del_1, not del_2, not del_3 hold. The deletion of clause 2 *from P is simulated by the addition of the clause del_2 to $P!$.*