Maintenance of Consistency in Well-Founded Models of Extended Logic Programs with Explicit Negation*

Renwei Li and **Luís Moniz Pereira**
Department of Computer Science, Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
{renwei | lmp}@di.fct.unl.pt

Abstract This paper deals with contradictory information in common-sense reasoning. It is often the case that our beliefs will become inconsistent when new knowledge is acquired and thus added. The traditional treatment of inconsistency is to simply regard inconsistent information useless, since everything is implied by, and can thus be derived from, contradictory information. In this paper we use extended logic programs with explicit negation to represent facts, causal laws, and hypothetical beliefs, based on the paraconsistent well-founded model semantics. We use some examples to argue that the essential source of inconsistency of beliefs is due to assumptions which were made to derive beliefs. We present an algorithm to eliminate inconsistency in paraconsistent well-founded models of extended logic programs by using a basic assumption-based truth maintenance system ATMS. The general idea of the algorithm is first to translate extended logic programs into some suitable propositional Horn clauses and then use nogoods returned by ATMS to look for the source of contradictions. We also use some simple examples to show the usefulness of our inconsistency elimination algorithm in common-sense reasoning.

Keywords: Non-Monotonic Reasoning, Belief Revision, Semantics of Logic Programs

1 Introduction

We start with a simple murdur case: The police receives a report from a woman called Joyce about her husband death. According to Joyce, she found her husband Fred died at home when she returned from beach. Joyce also claimed that she stayed in the beach for swimming all the day. In addition, the police also firmly believes that anyone will be tanned if he/she goes to the beach for swimming when the sun is shining. The police noticed that it was shining on the day of murder. A detective discovered that Joyce was not tanned at all.

All the above information can be represented by a first-order theory. It turns out that the first-order theory is inconsistent, since Joyce is both tanned and untanned. In traditional first-order logic, inconsistent information is not useful at all, since anything can be derived from it. In common-sense reasoning, however, inconsistent information can help solve problems by finding the source of inconsistency. In the example above, a possible source of inconsistency is that

*This paper was presented in *IBERAMIA'96: the Fifth Ibero American Conference on Artificial Intelligence*, Cholula, Puebla, Mexico, October 28 – November 1, 1996.

we have actually made an assumption: Joyce was honest. If Joyce was lying, we should not believe what she said. That is to say, Joyce did not go to beach for swimming. Then, the inconsistency is eliminated.

Of course, practical problems in our reality are much more complicated than the above one. But the general principle is the same: When inconsistency arises, we should look for the sources of inconsistency. The source is among assumptions which were made in deriving beliefs. Discarding some assumptions will resume consistency. As another simple example, suppose the police received a phone call to report that the National Bank was being robbed. When the police arrives in the National Bank and finds that there has never been any robbery at all in the recent months. Now there is an inconsistency to handle: the bank was robbed and the bank was not robbed. In practice the police would simply regard the phone call as *a joke call*, which is the source of the inconsistency. If the police does not believe joke calls, there is no inconsistency at all.

In order to handle inconsistency in common-sense reasoning, first of all we need to have a language to represent knowledge and beliefs: basic facts (e.g., Fred was dead), causal laws (e.g., anyone will be tanned if he/she goes to the beach for swimming when the sun is shining), hypothetical beliefs (e.g, the bank was being robbed if the call was not a joke call), etc. In this paper we choose to use extended logic programs with explicit negation to represent knowledge and beliefs. There are two kinds of negations in extended logic programs: explicit negation and negation-as-failure. For the bank robbery example, we can use the following three clauses:

$$\begin{aligned} \neg robbed &\leftarrow . \\ robbed &\leftarrow \neg jokecall. \\ \neg jokecall &\leftarrow not\ jokecall. \end{aligned}$$

Where the symbol $\neg$ is the explicit negation and *not* denotes negation-as-failure. The first clause says that the bank was not robbed, the second says that the bank was robbed if the phone call was not a joke, and the third says that it is not a joke call if you do not know it is a joke call. The precise meaning of clauses will be introduced in later sections.

The semantics of extended logic programs with explicit negation may be defined in several ways, e.g. answer-set semantics [10], well-founded model semantics with explicit negation (WFSX) [1], etc. In this paper we adopt the well-founded model semantics with explicit negation WFSX, since it enjoys some desirable structural properties [2]: well-foundedness, cumulativity, rationality, relevance, and partial evaluation. By [1] any program either has a unique WFSX model or does not have any WFSX model. When a program does not have any WFSX model, i.e. the fixpoint operator for computing WFSX is not defined, there is a contradiction due to the closed-world assumption (CWA) in the program. A contradictory program can be revised by abandoning CWA on some literals to restore the consistency of the program [13, 14].

In line with [13, 14], we present an alternative way to eliminate inconsistency in extended logic programs by using a basic assumption-based truth maintenance system (ATMS). The technical contribution of this paper is two-fold. First we relate the contradiction removal semantics of extended logic programs to ATMS structure, especially its *nogoods*. On the other hand, we can use existing optimization techniques to improve efficiency of our basic algorithm to compute the contradiction removal semantics. The concept of *support set* of [13, 14] turns out to be unnecessary in this paper. The result of this paper can be used in common-sense reasoning where complete information is not available and assumptions are made to derive beliefs.

The rest of this paper is organized as follows. In Section 2 we briefly introduce extended logic programs and their WFSX semantics. In Section 3 we review ATMS. In Section 4 we discuss the concept of revisables and contradiction removal sets. In Section 5 we introduce a simple translation which transforms an extended logic program into propositional Horn clauses. In Section 6 we present a basic algorithm to remove contradictions in logic programs by using ATMS and prove its correctness. In Section 7 we discuss some techniques to improve the efficiency of the basic algorithm. In Section 9 we compare our work with other related work .

2 WFSX semantics of extended logic programs

An extended logic program is a set of rules of the form

$$L \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.\ (0 \leq m \leq n)$$

where L and each L_i is either an atom or an atom preceded by explicit negation $\neg$. Atoms and atoms preceded by explicit negation $\neg$ are generally called objective literals. The set of all ground objective literals of a program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. The symbol *not* stands for negation as failure. By $not\ \{a_1, \ldots, a_n, \ldots\}$ we mean $\{not\ a_1, \ldots, not\ a_n, \ldots\}$. An interpretation I of an extended logic program P is denoted by $T \cup not\ F$ with the coherence principle (if $\neg L \in T$ then $L \in F$), where T and F are two subsets of $\mathcal{H}(P)$. Objective literals in T are said to be *true* in I, objective literals in F *false by default* in I, and in $\mathcal{H}(P) - I$ *undefined* in I.

WFSX follows from WFS [18] for normal programs together with the coherence requirement relating the two forms of negation. We start with the definition of the Gelfond–Lifschitz module operator $\frac{P}{I}$.

Definition 2.1 (Module Operator $\frac{P}{I}$) Let P be an extended logic program, I an interpretation. Nondeterministically perform the following three operations: (i) Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$; (ii) Remove from P all rules containing an objective literal L in the body such that $\neg L \in I$; (iii) Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$. $\frac{P}{I}$, P modulo I, is the program obtained from the above result by replacing all the remaining default literals by proposition $\mathbf{u}$.

Obviously, the resulting program $\frac{P}{I}$ is non–negative, i.e. no default literals appear in it. Thus there is a unique Fitting-least 3-valued model, $least(\frac{P}{I})$, which is a generalization of van Emden-Kowalski least model operator for definite logic programs [16]. Note that $\frac{P}{I}$ is not unique because of the nondeterministic operations.

Definition 2.2 We define $least(P)$, where P is a non–negative program, as the set of literals $T \cup not\ F$ obtained as follows: (i) Let P' be the non–negative program obtained by replacing in P every negative objective literal $\neg L$ by a new atomic symbol, say $'\neg\_L'$; (ii) Let $T' \cup not\ F'$ be the least 3–valued model of P' [16]; (iii) $T \cup not\ F$ is obtained from $T' \cup not\ F'$ by reversing the replacements above.

Although $least(P)$ uniquely exists for every non–negative program P, it might not satisfy the coherence principle, and hence it isn't always an interpretation.

Definition 2.3 (Coh operator) Let $QI = QT \cup not\ QF$ be a set of literals. We define $Coh(QI)$ to be the interpretation $T \cup not\ F$ such that $T = QT$ and $F = QF \cup \{\neg L \mid L \in T\}$.

Definition 2.4 (Φ operator) Let P be an extended logic program, I an interpretation. We define $\Phi_P(I) = \bigcup Coh(least(P_i))$, where P_i is some $\frac{P}{I}$.

It can be shown that the operator Φ_P for any program P is monotonic, and hence there is a least fixpoint $lfp(\Phi_P)$.

Definition 2.5 (Paraconsistent WFSX) Let P be an extended logic program, its paraconsistent WFSX model is defined to be the least fixpoint $lfp(\Phi_P)$. If $L \in lfp(\Phi_P)$, we write $P \overset{p}{\models} L$. P is said to be contradictory iff for some positive literal L, both $P \overset{p}{\models} L$ and $P \overset{p}{\models} \neg L$. P is said to be consistent iff P is not contradictory.

For example, the paraconsistent WFSX model of the program for the joke-call is $\{\neg robbed, robbed, \neg jokecall\} \cup$ not $\{robbed, \neg robbed, jokecall\}$. In what follows we will write $\mathrm{WFSX}_p(P)$ for $lfp(\Phi_P)$.

3 ATMS

ATMS [5] is an assumption-based truth maintenance system used as part of an overall reasoning system or problem-solving system consisting of two components: a domain dependent problem solver which performs inferences and a domain independent ATMS which records inferences and maintains truth or reasons. The problem solver communicates with ATMS through justifications in the form of propositional Horn clauses. In this paper we only use the following basic capabilities of ATMS.

The inputs of ATMS are a set of propositions N, called nodes, a set of propositional Horn clauses J, called justifications, and a special subset A of N, called assumptions. When N, J and A are transmitted to ATMS, it computes the following structure:

- All the minimal subset δ of A such that $J \cup \delta$ is inconsistent. These sets are called *nogoods*.
- For every node n, all the minimal subset δ of A such that $J \cup \delta \models n$. δ is called an environment of n, and the set of all *environments* of a node n is called the *label* of n. ATMS guarantees that each label of each node is consistent, sound, complete and minimal with respect to the justifications [5].

For example, let $N = \{a, b, c, d\}$, $A = \{c, d\}$, and J be $\{a \leftarrow c.\ \ b \leftarrow d.\ \ \bot \leftarrow a, b\}$. Then, the ATMS will compute the following labels for each node[1]: $\{a : \{\{c\}\}; \quad b : \{\{d\}\}; \quad c : \emptyset; \quad d : \emptyset; \quad \bot : \{\{c, d\}\}\}$. Given justifications J and assumptions A, we use $Lable^{J,A}(L)$ to denote the label of L, and use $Env^{J,A}(L, X)$ to denote X is an environment of L.

[1]By de Kleer [5] if $n \in A$ then n has at least one environment $\{n\}$, which is omitted in this paper.

4 Revisables

Suppose P is contradictory. We need to know what has made it inconsistent before we restore its consistency. Let's look at a simple program $\pi_1 = \{p \leftarrow a.\quad \neg p \leftarrow not\ b.\quad a \leftarrow .\}$. Its paraconsistent WFSX model is $\{p, \neg p, a\} \cup not\ \{p, \neg p, \neg a, b, \neg b\}$. Since both p and $\neg p$ appear in its paraconsistent WFSX model, it is contradictory (by adding rule $\perp \leftarrow p, \neg p$ we can explicitly express that $\perp$ appears in the paraconsistent WFSX model). The rational reasoners may believe that both a and p should appear in its WFSX model. The real source of the inconsistency might be that the assumption *not* b being true is not correct. Why *not* b belongs to the paraconsistent WFSX model is that WFSX has embodied a kind of closed world assumption (CWA): if you know nothing about b, *not* b is supposed to be true.

Our stance with respect to CWA is that CWA holds of a literal L if no contradiction arises after you use CWA on L and CWA does not hold of a literal L if contradiction arises after you use CWA on L. In the example above, no contradiction will arise if we withdraw the CWA assumption on *not* b. To withdraw the CWA assumption on *not* b, it is sufficient to assume that b is true. Actually, no contradiction will arise after we add a rule: $b \leftarrow$.

The default literals such as *not* b in the above example is called a *revisable symbol*, or simply a *revisable*. The concept of revisables can be made more precise in line of [14].

Definition 4.1 The revisables of a program P, denoted by $Revisables(P)$, is the smallest set satisfying the following condition: $not\ L \in Revisables(P)$ iff there are no rules for L in P.

Definition 4.2 Let $Revisables(P)$ be revisables of a contradictory program P. A subset $r \subseteq Revisables(P)$ is called a contradiction removal set of P iff the program $P \cup \{L \leftarrow .\ \mid\ not\ L \in r\}$ is consistent.

Sometimes we may have several contradiction removal sets. The minimal ones are preferred to others.

Definition 4.3 A subset $r \subseteq Revisables(P)$ is called a minimal contradiction removal set of P iff no proper subset of it is a contradiction removal set of P.

We will often write CRS and MCRS to denote *ontradiction Removal Set* and *Minimal Contradiction Removal Set*, respectively.

5 Saturation

In this section we introduce a program transformation which transforms an extended logic program π into a set of propositional Horn clauses, called *saturated program* of π.

Definition 5.1 Let P be an extended logic program. A new propositional Horn program $\varepsilon(P)$ is obtained as follows:

- For each atom a appearing in P, introduce three symbols as follows: $a_\neg, a^*, a^*_\neg$ for $\neg a, not\ a, not\ \neg a$, respectively.
- Replace $\neg a, not\ a, not\ \neg a$ in P with $a_\neg, a^*, a^*_\neg$, respectively, and for each atom a add three additional rules:

$$\{\perp \leftarrow a, a_\neg.\quad a^* \leftarrow a_\neg.\quad a^*_\neg \leftarrow a.\}$$

If L^* appears in $\varepsilon(P)$ and there are no rules for L, we also say that L^* is a revisable. We will write $Revisables(\varepsilon(P))$ to denote that $L_1^*, \ldots, L_n^*$ are revisables of $\varepsilon(P)$.

Obviously, $\varepsilon(P)$ does not change rules defining positive literals. The objective of $\varepsilon(P)$ is to explicitly import the coherence principle into the program. In the following we only consider $\varepsilon(P)$. We also use the following notation:

$$L^* = \begin{cases} a^*, & \text{If } L \text{ is an atom } a; \\ a_\neg^*, & \text{If } L \text{ is of the form } a_\neg \text{ for some atom } a; \\ a, & \text{If } L = a^* \text{ for some atom } a; \\ a_\neg, & \text{If } L = a_\neg^* \text{ for some atom } a. \end{cases}$$

Let $H = \{L_1, \ldots, L_n\}$. We define $H^* = \{L_1^*, \ldots, L_n^*\}$, and write $L \leftarrow H$ to stand for the rule $L \leftarrow L_1, \ldots, L_n$.

Definition 5.2 (Hitting Sets) Let $E = \{E_1, \ldots, E_n\}$, where each E_i is a set. A set $C = \{L_1, \ldots, L_m\}$ is said to be a hitting set of E iff for each $E_i, E_i \cap C$ is not empty. A hitting set is said to be minimal iff no proper subset of it is a hitting set.

Definition 5.3 Let π be a program. Assume that L^* appears in the body of a rule of π. We define $\rho(\pi, L^*)$ as follows: Let all rules defining L be as follows:

$$\begin{array}{lll} L & \leftarrow & L_{11}, \ldots, L_{1m_1} \\ L & \leftarrow & L_{21}, \ldots, L_{2m_2} \\ & \ldots & \\ L & \leftarrow & L_{n1}, \ldots, L_{nm_n} \end{array}$$

If there is one rule $L \leftarrow L_{k1}, \ldots, L_{km_k}$, where the body $L_{k1}, \ldots, L_{km_k}$ is empty, then $\rho(\pi, L^*) = \pi$. If there are no rules defining L, then $\rho(\pi, L^*) = \pi$. Otherwise, let all the minimal hitting sets of $\{L_{11}, \ldots, L_{1m_1}\}$, $\{L_{21}, \ldots, L_{2m_2}\}$, $\ldots, \{L_{n1}, \ldots, L_{nm_n}\}$ be $H_1, H_2, \ldots, H_k$. Then define

$$\rho(\pi, L^*) = \pi \cup \{L^* \leftarrow H^* \ \mid \ H \in \{H_1, H_2, \ldots, H_k\}\}$$

The intuition behind $\rho(\pi, L^*)$ is to relate the truth of *not* L to those which defines L.

Definition 5.4 (Saturated program) Let π be a program. We define $\rho(\pi)$ be the smallest set of rules satisfying: (i) $\pi \subseteq \rho(\pi)$; (ii) If L^* appears in the body of one rule of $\rho(\pi)$, then $\rho(\rho(\pi), L^*) \subseteq \rho(\pi)$. We say that $\rho(\pi)$ is a saturated version of π with respect to *not*. A program is said to be saturated with respect to *not* iff $\rho(\pi) = \pi$.

Proposition 5.5 For any program there is an algorithm to compute its saturated version.

6 Basic algorithm

There might be many algorithms for eliminating inconsistency. For example, we could have a general algorithm based on *generate-and-test* strategy. In what follows we present a basic algorithm by using ATMS to remove contradictions. Assume P is a logic program. The algorithm is as follows:

Step 1: Compute $\rho(\varepsilon(P))$ and $(Revisables(\rho(\varepsilon(P))))$. $Revisables(\rho(\varepsilon(P))))$ is computed simply by using the following rule: If L^* appears in the body of a rule of $\rho(\varepsilon(P))$ and there are no rules for L in $\rho(\varepsilon(P))$, then $L^* \in Revisables(\rho(\varepsilon(P)))$.

Step 2: Transmit $\rho(\varepsilon(P))$ as justifications J, $Revisables(\rho(\varepsilon(P)))$ as assumptions A, and all symbols of the forms $a, a_\neg, a^*, a^*_\neg$ for each atom a as nodes N to the ATMS, and run it. The output of the ATMS contains all the information on labels for each node.

Step 3: Judge whether the contradiction arises or not. If $Label(\perp)$ is empty, then the original program is not contradictory or cannot be revised. Check whether the original program is not contradictory. This can be done by calling, e.g. the SLX procedure [2]. If the original program is contradictory but cannot be revised, stop and output this fact. If the original program is not contradictory, jump out of the current iteration. If $Label(\perp)$ is not empty, then the original program is contradictory. If $Label(\perp)$ contains $\emptyset$, the original program is contradictory and cannot be revised, then stop and output this fact. If $Label(\perp)$ contains non-empty sets $C_1, \ldots, C_n$, the original program is contradictory and could be revised, then goes on.

Step 4: Compute the minimal hitting sets of $C_1, \ldots, C_n$. Suppose there are n hitting sets. We denote them with $H_1, \ldots, H_n$, respectively.

Step 5: For each hitting set H_i, check whether H_i is a MCRS. If it is, a MCRS is found. If it is not, iterate. To be more specific, Let $P_1 = P \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{\neg a \leftarrow . \mid a^*_\neg \in H_i\}$, go back to Step 1. If it stops at Step 3 and says that P_1 is not contradictory, then H_i is a MCRS. If P_1 is a contradictory program with a MCRS or CRS H_{i_j}, then $H_i \cup H_{i_j}$ is a CRS.

Step 6: Compute MCRSs. It is easy. If it has been marked as MCRS in the last step, it is a MCRS. If it is a CRS, just check if there is another CRS which is a proper subset of it. Let M be a MCRS, then $P \cup \{L \leftarrow . \mid L^* \in M \text{ and } L \text{ is an atom}\} \cup \{\neg L \leftarrow . \mid L^*_\neg \in M \text{ and } L \text{ is an atom}\}$ is an output.

Definition 6.1 (Correctness) Let P be a logic program, and R be a contradiction removal algorithm. Assume $R(P)$ is the new program which is obtained by running R on P. R is said to be weakly correct iff $R(P)$ is consistent or cannot be revised with respect to $Revisables(P)$. R is said to be strongly correct iff (i) R is weakly correct; (ii) If P is not contradictory, then $R(P) = P$; (iii) If P is contradictory, then $R(P) = P \cup C$, where $C \subseteq Assumables(P)$.

Theorem 6.2 The basic algorithm is strongly correct.

7 Efficiency consideration

There are many existing techniques for improving efficiency of ATMS. This is one of main reasons for why we want to use ATMS to remove contradictions in logic programs. In basic ATMS, to each node are attached five elements: Data (the proposition itself), Label (a set of environments), Justifications (a list of the derivations of the data), Consequents (a list of justifications in which the node is an antecedent), and Nogoods. Each of environments and justifications also has its own internal structures. It is certainly expensive to construct these structures in the use of ATMS at the first time. But later the information contained in the structure can be effectively and efficiently employed to construct a new ATMS structure when, e.g. a new justification is added. This is really the case in our basic algorithm. After we find a hitting set H_i, we need to check whether $P_1 = P \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{\neg a \leftarrow . \mid a^*_\neg \in H_i\}$ is contradictory or not. It means that we need to use ATMS again to compute the structure. But notice that at the second time ATMS need not compute the structure from the scratch, instead it uses some standard ATMS operations such as *adding justifications*. That is to say, ATMS is incremental. The efficiency of ATMS is achieved by taking advantage of this fact. This is a crucial point compared with [14], where a brute-force algorithm is presented and many iterations are involved in the algorithm.

In addition to the optimizing strategies concerning general ATMS implementation, we can also optimize our basic algorithm in particular ways.

One observation is as follows: Suppose a^* is a revisable for an atom a. Then we know that there are no rules defining a. Thus, the label of a^* is the same as $a_\neg$. Similarly, if $a^*_\neg$ is a revisable for an atom a, then the label of $a^*_\neg$ is the same as a. Therefore, for revisables it is easy to compute their labels. This observation is formalized by the following lemma.

Lemma 7.1 Let a be an atom, $J = \rho(\varepsilon(P))$ and $A = Revisables(\rho(\varepsilon(P)))$. Then:

$$Label^{J,A}(a^*) = Label^{J,A}(a_\neg) \text{ and } Label^{J,A}(a^*_\neg) = Label^{J,A}(a)$$

A second observation is about the next iteration for the program $P_1 = P \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{\neg a \leftarrow . \mid a^*_\neg \in H_i\}$. The first step of the algorithm can almost be omitted by using the following lemma.

Lemma 7.2 Let $P_1 = P \cup \{a \leftarrow \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{\neg a \leftarrow \mid a^*_\neg \in H_i\}$. Then, we have

1. $\varepsilon(P_1) = \varepsilon(P) \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{a_\neg \leftarrow . \mid a^*_\neg \in H_i\}$.
2. $\rho(\varepsilon(P_1)) = \rho(\varepsilon(P)) \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{a_\neg \leftarrow . \mid a^*_\neg \in H_i\}$.
3. $Revisables(\rho(\varepsilon(P_1))) = Revisables(P) - H_i$.

A third observation is still about the next iteration for the program $P_1 = P \cup \{a \leftarrow . \mid a^* \in H_i \text{ and } a \text{ is an atom } \} \cup \{\neg a \leftarrow . \mid a^*_\neg \in H_i\}$. As said before, ATMS is incremental. In Step 2 working on P_1, ATMS can provide effective and efficient way to compute the new ATMS structure.

8 An example

In many practical domains we often encounter inconsistency when new knowledge is acquired. For the example in the introduction, we can deal with it as follows. We use the following propositions: d for *Fred was dead*, b for *Joyce went to beach to swim*, t for *Joyce was suntanned*, s for *the sun shined*, l for *Joyce was a liar*. Then we can have the following logic program:

$$\{d \leftarrow . \quad b \leftarrow \text{not } l. \quad t \leftarrow s, b. \quad s \leftarrow . \quad \neg t \leftarrow .\}$$

Since there is no evidence to l, Joyce was assumed to be honest; she was not a liar. Thus, b belongs to the paraconsistent WFSX model , which leads to an inconsistency: both $\neg t$ and t belongs to the paraconsistent WFSX model. Using the basic algorithm, we can find a MCRS $\{not\ l\}$. After adding $\{l \leftarrow .\}$ to the program, the inconsistency is eliminated. The police can reach a conclusion: Joyce was a liar and she did not go to beach to swim.

Of course, practical cases might be much more complicated. But we can always follow the same idea to deal with them.

9 Related work and conclusions

In this paper we have argued that inconsistent information is very useful in common-sense reasoning. Technically, we use extended logic programs to represent basic facts, causal laws, and hypothetical beliefs, based on the paraconsistent well-founded model semantics. We use two simple examples to argue that the essential source of inconsistency of beliefs is due to assumptions which were made to derive beliefs. We present an algorithm to eliminate inconsistency in paraconsistent well-founded models of extended logic programs by using a basic assumption-based truth maintenance system ATMS. The technical contribution of the inconsistency elimination algorithm is two-fold: First we relate the contradiction removal semantics of extended logic programs to ATMS structure, especially its nogoods. On the other hand, we can use existing optimization techniques to improve efficiency of our basic algorithm to compute the contradiction removal semantics. As in other ATMS-based systems, the method is applicable for maintaining consistency in well-founded models of extended logic programs with explicit negation

There are two pieces of related work, namely [13, 14] on three-valued (true, false, undefined) contradiction removal, and two-valued (true, false) contradiction removal, respectively. The inconsistency elimination algorithm of this paper is different from [13, 14] in that we make use of a basic ATMS structure to remove contradictions. The use of ATMS also enables us to use existing optimization techniques of ATMS to efficiently remove contradictions (sometimes). As discussed in Section 7, we also have some other optimization techniques. A recent work is [3], which uses some preference relations and pruning rules to enhance efficiency of [14]. The exact comparison between [3] and the ATMS-based algorithm of this paper is yet to be investigated.

Another related topic, not addressed in this paper, is the relationship between WFSX semantics and ATMS structure. We have shown that we can use ATMS to remove contradictions but we haven't said how to use ATMS to compute WFSX. There have been some work [15, 7] on relationships between stable models [9] of logic programs and ATMS , and some other work [8, 17, 11] on relationships between autoepistemic logic (or default logic) and Doyle's TMS [6]. There is also some work on computing 3-valed stable models by using ATMS [12] and AMTS implementation of logic programming [4]. However, It has not been clear to us how to compute WFSX by using ATMS or TMS, neithere has it been clear whether we really need to use ATMS or TMS to compute WFSX semantics since ATMS is very expensive. In this paper we use ATMS because we want to, in a long run, maintain consistency whenever new logic programming rules are added, and thus may exploit the incremental computational advantage of ATMS. If we need to revise a program for consistency only for once, it is not necessary to use ATMS.

Acknowledgements

This work was supported by a JNICT PRAXIS post-doctoral fellowship under PRAXIS XXI/BPD/4165/94 to the first author, and a JNICT PRAXIS project PROLOPPE under the grant PRAXIS/3/31/TIT/24/94. We would also like to thank José Júlio Alferes and Carlos Damásio for helpful discussions.

References

[1] Alferes, J.J., and Pereira, L. M., *Reasoning with Logic Programming*, Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996

[2] Alferes, J.J., Damásio, C.V., and Pereira, L.M., A Logic Programming System for Non-Monotonic Reasoning, *Journal of Automated Reasoning* 14, 1995, 93–147

[3] Damásio, C. V., Nejdl, W. and Pereira, L.M., REVISE: An Extended Logic Programming System for Revising Knowledge Bases, KR'94, 1994

[4] Brogi, A., Lamma, E., and Mello, P., ATMS for implementing logic programming, *ECAI 92*, 114–118

[5] de Kleer, J., An assumption-based truth maintenance system, *Artificial Intelligence*, 28, 1986, 127–162

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

[7] Eshghi, K., Computing stable models by using ATMS, *AAAI 90*, 1990, 273 – 277

[8] Fujiwara, Y. and Honiden, S., Relating the TMS to autoepistemic logic, *IJCAI 89*, 1989

[9] Gelfond, M., and Lifschitz, V., The stable model semantics for logic programming, In. *Proc. of 5th Logic Programming Symposium*, R. Kowalski and K. Bowen (eds.), 1988, 1070-1080

[10] Gelfond, M. and V. Lifschitz, Logic programs with classical negation, *Logic Programming: Proc. of the 7th Int'l Conf.*, D. Warren and P. Szeredi, (eds.), MIT Press, 1990, pp. 579-597

[11] Junker, U., Konolige, K., Computing the extensions of autoepistemic and default theories with a truth maintenance system, *AAAI 90*, 1990, 279 – 283

[12] Lamma, E. and Mello, P., Computing 3-valued stable models by using the ATMS, *Proc. of Italian Conf. on Artificial Intelligence, LNAI 728*, 1993

[13] Pereira, L. M., Aparício, J. N., and Alferes, J. J., Contradiction removal within well-founded semantics, *Proc. of Logic Programming and Nonmonotonic Reasoning, LPNMR'91*, Nerode, A., Marek, W., Subrahmanian, V.S., (eds.), The MIT Press, 1991, 105-119

[14] Pereira, L. M., C. V. Damásio, and Alferes, J. J., Diagnosis and debugging as contradiction removal, *Proc. of Logic Programming and Nonmonotonic Reasoning, LPNMR'93*, Pereira, L. M. and Nerode, A., (eds.), The MIT Press, 1993, 316–330

[15] Pimental, S.G. and Cuadraro, J.L., A truth maintenance system based on stable models, *AAAI 89*

[16] Przymusinska, H. and przymusinski, T., Semantics issues in deductive databases and logic programs, Banerji, R., (ed.), *Formal Techniques in AI: A Source Book*, North-Holland, 1990, 321–367

[17] Reinfrank, M., Dresssler, O., Brewka, G., On the relation between truth maintenance and auto-epistemic logic, *IJCAI*, 1989

[18] Van Gelder, A. , Ross, K. and Schlipf, J.S., The well-founded semantics for general logic programs, *J. ACM* 38 1991, 620-650