Paraconsistent Annotated Logic Programming - Paralog

Jair Minoro Abe
Department of Informatics, ICET - Paulista University
R. Dr. Bacelar, 1212
04026-002 São Paulo - SP - BRAZIL
and
Institute For Advanced Studies, University of São Paulo
Av. Prof. Luciano Gualberto, Travessa J, 374, Térreo, Cidade Universitária,
05508-900 São Paulo – SP - BRAZIL
e-mail: jmabe@uol.com.br

Kazumi Nakamatsu
School of Humanities for Environment Policy and Technology
Himeji Institute of Technology
Shinzaike 1-1-12, Himeji
670-0092 JAPAN
e-mail : nakamatu@hept.himeji-tech.ac.jp

Bráulio Coelho Ávila
LASIN – PPGIA
PUC-PR – Pontifical Catholic University of Paraná
R. Imaculada Conceição, 1155
80215-901 Curitiba – PR – Brazil
e-mail : avila@ppgia.pucpr.br

Abstract

Inconsistency is a natural phenomenon arising from the description of the real world. This phenomenon may be encountered in several situations. Nevertheless, human beings are capable of reasoning adequately. The automation of such reasoning requires the development of formal theories. Paraconsistent Logic provides tools to reason about inconsistencies. Though inconsistency is an increasingly common phenomenon in programming environments it cannot be handled, at least directly, by Classical logic, on which most of the current logic programming languages are based. Thus, one has to resort to alternatives to classical logic; it is therefore necessary to search for programming languages based on such alternatives. Paraconsistent logic, despite having been initially developed from the purely theoretical standpoint, found in recent years extremely fertile applications in Computer Science, thus solving the problem of justifying such logic systems from the practical standpoint. This work proposes a

International Journal of Computing Anticipatory Systems, Volume 6, 2000
Edited by D. M. Dubois, CHAOS, Liège, Belgium, ISSN 1373-5411 ISBN 2-9600179-8-6

variation of the logic programming language Prolog that allows inconsistency to be handled directly. The proposed language was dubbed Paralog.

Key words: paraconsistent logic programming, non-classical logics and programming, inconsistency and logic programming, inconsistency and Prolog, paraconsistent logic and Computer Science.

1 Introduction

The employment of logic systems allowing reasoning about inconsistent information is an area of growing importance in Computer Science, Data Base Theory and Artificial Intelligence. For instance, if a knowledge engineer is designing a knowledge base *KB*, related to a domain *D*, he may consult *n* experts in that domain. For each expert e_i, $1 \leq i \leq n$, of domain *D*, he will obtain some information and will present it in some logic such as a set of sentences KB_i, for $1 \leq i \leq n$. A simple way of combining the knowledge amassed from all experts in a single knowledge base *KB* is

$$KB = \bigcup_{i=1}^{n} KB_i$$

However, certain KB_i and KB_j bases may contain conflicting propositions, p and $\neg p$. In such case, p might be a logic consequence of KB_i, while $\neg p$ might be a logic consequence of KB_j. Therefore, *KB* is inconsistent and consequently meaningless, because of the lack of models. However, the knowledge base *KB* it not a useless set of information.

There are some arguments favoring this standpoint, as follows:

- certain subsets *KB* may be inconsistent and express significant information. Such information cannot be disregarded;
- the disagreement among specialists in a given domain may be significant. For instance, if physician M_1 concludes patient *X* suffers from a fatal cancer, while physician M_2 concludes that same patient suffers from cancer, but a benign one, the patient will probably want to know the causes of such disagreement. This disagreement is significant because it may lead patient *X* to take appropriate decisions - for instance, to get the opinion of a third physician.

The reasoning for the last item is that it is not always advisable to find ways to exclude formulas identified as causing inconsistency(ies) in *KB*, because many times important information may be removed. In such cases, the very existence of inconsistency is important.

The first efforts to handle inconsistent logic systems were developed by Russian logician Nikolai A. Vasil'ev and by Polish logician Jan Lukasiewicz. Both published independent works in 1910 on the possibility of a logic that would not eliminate

contradictions *ab initio*. These works, however, clung to traditional Aristotelian logic in what concerned paraconsistency. Only in 1948 and 1954 Polish logician S. Jas'kowski and Brazilian logician Newton C. A. da Costa, respectively, built up Paraconsistent logic [1], [16], [18], independently.

2 Paraconsistent logics

In this paragraph we establish some terminologies.

Let T be a theory whose underlying logic is L. T is *inconsistent* when it contains theorems of the form A and $\neg A$ (the negation of A). If T is not inconsistent, it is called *consistent*. T is said to be *trivial* if all formulas of T are also theorems of T. Otherwise, T is called *non-trivial*. So, in trivial theories, the extensions of the concepts of formula and theorem coincide. When L is the Classical Logic (or several other ones, such as Intuitionistic Logic), a theory is trivial iff it is inconsistent. A *paraconsistent* logic is a logic that can be used as the basis for inconsistent but non-trivial theories. A theory is called *paraconsistent* if its underlying logic is a paraconsistent logic.

As a consequence, paraconsistent theories do not satisfy the principle of non-contradiction which can be stated as follows: from two contradictory propositions (i.e., one is the negation of the other) one must be false.

3 Paraconsistent Annotated Evidential Logic Programming

The use of annotated formulas in logic programming was introduced by Subrahmanian and by Blair in [27] and [13]. Very significant applications were made subsequently in AI, as well as in Computer Science (some references are [11], [21], [22], [23], [24]). So a number of authors dedicate to study these systems a from foundational point of view: N.C.A. da Costa, V.S. Subrahmanian, J.M. Abe, S. Akama, among others (see, for instance, [1], [2], [3], [5], [8], [9], [10], [18]). These logics have proved to be powerful tool to deal with inconsistencies and paracompleteness in a non-trivial manner (by far the most detailed presentation appears in Abe's [1] thesis). It is to observed that such concepts are more and more common in several contexts in AI, Robotics, and other fields of applications. These papers showed it is possible dud convenient to associate annotations to Horn clauses.

The use of *evidential reasoning* in logic programming was proposed by Subrahmanian in [28], [14]. For that effect, Subrahmanian proposes an infinitely valued paraconsistent logic, in which the truth-values are members of the lattice $\tau = \{x \in \Re \mid 0 \leq x \leq 1\} \times \{x \in \Re \mid 0 \leq x \leq 1\}$ (with usual product ordering).

This lattice possesses a minimum element [0, 0] and a maximum element [1, 1]. The minimum element corresponds to indefinite - underdetermined - and the maximum element corresponds to inconsistent - overdetermined. Intuitively, [1, 0] and [0, 1] correspond, respectively, to true and false in bi-valued logic. The annotations of this

logic system may be considered as points in a unitary square on the Cartesian plane. Let (x, y) be a point of the unitary Cartesian square.

Thus, a p atom is considered perfectly defined when belonging to the line $x + y - 1 = 0$; a p atom is considered overdefined when belonging to the line $x + y - 1 = 0$; and, a p atom is considered underdefined when positioned down line $x + y - 1 = 0$. The degree of inconsistency of atom p is $(x + y - 1).100$, where $x + y \geq 1$. The degree of indetermination of atom p is $(x + y - 1).100$, where $x + y \leq 1$.

4 The Paralog Logic Programming Language

In several events in the real world, evidences [26] play a fundamental role in decision-making. Most human decision-making is based on previous experiences. Therefore, an individual, when faced with decision-making and imprecise information, considers all possibilities - investigates all evidences - and finally decides on the course of action to be taken.

For instance, if the following evidences exist: less than 10% of animals can fly; more than 90% of birds can fly; generally all birds are animals; Tweety is a bird; it is intuitive to represent the proposition *less than 10% of of animals can fly* as:

$$fly(X) : 0.1 - animal(X) : 1.0$$

But how to represent that *over 90% of animals cannot fly*?

In this example, the answer to the following query: *Can Tweety fly* ?, may not be so simple and may lead to erroneous conclusions.

The Paralog language, using the annotation concept, may relate just one evidence to proposition p. However, it has been shown that the use of two evidences related to the same proposition p may increase its expression power. Such two-evidence annotation may be thought of as one evidence favoring p and one evidence opposed to p. No restriction is applied to these evidences excepting that they be within the interval $\{x \in \Re \mid 0 \leq x \leq 1\}$.

In the previous example, the same proposition could be represented in Paralog as follows:

$$fly(X) : [0.1,0.9] - animal(X) : [1.0,0.0]$$

thereby allowing an increase in the expression power of that proposition, leading to proper conclusions. That is, the answer to the previous query must state that there is an 80% inconsistency for the fact that Tweety can fly.

The Paralog language is based on a non-classical - annotated (evidential) paraconsistent - logic that renders the use of non-logic extensions unnecessary. The semantics of a Paralog program is based on the semantics on Herbrand's minimal model. The new

resources introduced by Paralog are therefore application-independent and based on lst order annotated logic, complete and sound in relation to the semantics employed.

5 Syntax of Paralog

The implementation of Paralog is based in Edimburgh's syntax, plus new syntax elements related to Evidential Logic Programming [28].

Definition 5.1 (Alphabet) The basic Paralog alphabet possesses the same set of symbols of the standard Prolog, plus the symbol ":" and the evidential symbol. Paralog symbols are:

1. letters: *a*, *b*, ... , *z*, *A*, *B*, ... , *Z*
2. digits: 0,1,. . . , 9
3. special symbols: _, +, -, /, * and the space
4. punctuation marks: (,),. ,","
5. connective symbols: & (conjunction), <-- (implication), not (negation)
6. annotation symbol: ":"
7. annotation symbol[1]: [μ_1, μ_2]
8. evidential constant: μ_1, μ_2

Definition 5.2 (Expression) A Paralog *expression* is any finite sequence of its alphabet symbols.

Definition 5.3 (Atom) A Paralog *atom* is:

1. every expression made up of letters and digits, starting with a minuscule;
2. an expression made up of digits, having at most an occurrence of the symbol ".";
3. every expression - including a space - delimited by quotation marks.

Definition 5.4 (Constant) A Paralog *constant* is defined as:

1. an atom; or
2. an element of the lattice τ called evidential constant.

Definition 5.5 (Variable) A Paralog *variable* is defined as:

1. an expression of letters dud digits the first element of which is a capital letter; or
2. the symbol "_", called *anonymous* variable.

Definition 5.6 (Term) A Paralog *term* is inductively defined as:

1. a variable is a term;
2. a constant is a term;
3. if *f* is an atom and *f* has the role of a functional *n*-ary symbol and t_1,. . . , t_n are terms, then $f(t_1,. . . , t_n)$ is a term; and

[1] The lattice used is defined in paragraph 3.

4. a Paralog e expression is a term if and only if it is obtained by applying one of the foregoing - 1 to 3 - conditions.

Definition 5.7 (Atomic Formula) An evidential *atomic formula* is the expression of form $p(t_1,\ldots,t_n)$: $[\mu_1, \mu_2]$, for $n > 0$, where $t_1,\ldots,t_n$ are terms and p is au atom in the role of a predicate n-ary symbol. For simplification purposes, when $n = 0$, p():$[\mu_1, \mu_2]$, may be written as p:$[\mu_1, \mu_2]$.

Thus, an atom may simultaneously play the role of one or more functional symbols or predicates of different arities. This multiple use of the same atom does not result in ambiguity since the context of a program always points out the role that it represents.

Definition 5.8 (Evidential Clause) The set of Paralog evidential clauses if inductively defined as:
1. if p is an evidential atomic formula, then p is an evidential clause, called unitary evidential clause;
2. if p: $[\mu, \nu]$, and q_1 $[\mu_1, \nu_1],\ldots, q_n$ $[\mu_n, \nu_n]$, are evidential atomic formulas, then the expression

$$p\text{: } [\mu, \nu] \text{ <-- } q_1\ [\mu_1, \nu_1],\ldots, q_n\ [\mu_n, \nu_n],$$

is an evidential clause, called non-unitary evidential clause, where p: $[\mu, \nu]$ is the head and

$$q_1\ [\mu_1, \nu_1],\ldots, q_n\ [\mu_n, \nu_n],$$

is the body of the clause;
3. if p_1 $[\mu_1, \nu_1],\ldots, p_n$ $[\mu_n, \nu_n]$,
are evidential atomic formulas, then

$$\text{<-- } p_1\ [\mu_1, \nu_1],\ldots, p_n\ [\mu_n, \nu_n],$$

is an evidential clause, called objective clause;
4. a Paralog expression is a clause if and only if it is obtained by applying one of the foregoing - 1 to 3 - conditions.

Definition 5.9 (Paralog Program) A Paralog *program* is a finite non-empty set of unitary and non-unitary evidential clauses.

A Paralog program is presented below showing the example of Tweety bird proposed on section 4. This example has been adapted from Ng & Subrahmanian's work [25].

Example 5.1 A Paralog program on Tweety bird

```
flies(X): [0.1,0.9] <--
     animal(X): [1.0,0.0].
flies(X): [0.9,0.1] <--
     bird(X): [1.0,0.0].
animal(x) : [1.0, 0.0] <--
     bird(X): [1.0,0.0].
bird(tweety): [1.0,0.0].
```

In this example, the fact that Tweety is a bird is represented as:

```
bird(tweety): [1.0,0.0].
```

This clause may be read as: "It is known, with absolute favorable evidence and with no contrary evidence that Tweety is a bird". It may be seen that in this clause Tweety is written is small letters. This is necessary because of the Paralog language syntax. That is, the use of a name starting with a capital letter indicates the intention of defining a variable - for instance, a variable called Tweety - and not an atom.

Definition 5.10 The definition of Paralog language in Backus-Naur Form (BNF) notation is presented below:

< program >: : =<clause>< remaining program >
<clause>: : =< fact > .| < rule > .
< remaining_program >: : =<clause>)∅[2]
< fact >: : =<atom>: <annotation> | *<atom> (< argument >): <annotation>*
< rule >: : =<head> <-- < body >
<head>: : =< fact >
< body >: : =< fact > | < fact >< remaining body >
< remaining_body >: : = & < fact >)& < fact >< remaining body >)∅
< argument >: : =<atom>< remaining_arg > | <variable>< remaining arg >
< remaining_arg >: : =, < argument >), < argument >< remaining arg >) ∅
<atom>: : =<small_ letter>< n letters >
<variable>: : =<capital_ letter>< n_letters >
< n_letters >: : =<number>< n_letters > | < letter >< n_letters >) ∅
< letter >: : =<small_ letter>) <capital_letter>
<annotation>: : = [< annotational constant >, < annotational constant >]
<annotational_ constant>: : = 0. <number><number> | 1.00
<number>:: = 0|1|2|3|4|5|6|7|8|9
<small_letter>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<capital_letter>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

It may be observed that the definition of the non-terminal symbol *<annotation>* is related to the lattice τ adopted for this implementation. It is enough to redefine this non-terminal symbol to use other lattices.

Thus, a query to Paralog may be structured in BNF as:

< goal >: : =< fact >< remaining goal > .
< remaining_goal >: : = & < fact > |& < fact >< remaining goal >)∅

6 Semantics of Paralog

[2] This represents the empty expression

6.1 Declarative and Procedural Semantics

Similarly to a standard Prolog program, a P_0 program and a Q_0 query must undergo a regularization process, where:

- every anonymous variable is replaced by a new distinct variable;
- all atoms occurring in more than one role will be renamed, so that at the end of that process there are no atoms with the same name in different roles.

The P_1 program and the Q_2 query resulting from this regularization process are equivalent to P_0 dud Q_0 respectively. Differently from the standard Prolog, a Paralog program must also undergo two more syntactic transformations:
1. elimination of the negation; and
2. closure.

These two syntactic transformations are necessary to eliminate the facilities introduced by the Paralog language that are not part of the ELPS syntax.
In the first transformation, program P_1 and Q_1 query undergo the negation elimination process. In this process, all occurrences of *not* p:[μ, ν] evidential atomic formulas are replaced by p: [ν, μ] equivalent atomic formulas. The elimination of P_1 and Q_1 negation results in program P_2 and Q_2 query.

The last syntactic transformation is the closure, as described in [13], proving that program $CL(P_2)$ resulting from this transformation is logically equivalent to P_2; that is I is a P2 model if and only if I is a $CL(P_2)$ model.

Program $CL(P_2)$ and query $CL(Q_2)$ resulting from this syntactic transformation process are equivalent to P_0 and Q_0 in Paralog, respectively. Program $CL(P_2)$ is equivalent to a ELP E and query $CL(Q_2)$ is equivalent to a C query, if and only if:

1. a clause p:[μ, ν] occurs in $CL(P_2)$ if and only if there is a clause p:[μ, ν] <-- in E;
2. a non-unitary Paralog e clause p:[μ, ν] <-- q_1 [μ_1, ν_1] &. . . & q_n [μ_n, ν_n], occurs in $CL(P_2)$ if and only if there is a clause p:[μ, ν] <-- q_1 :[μ_1, ν_1] &. . . & q_n :[μ_n, ν_n] in E;
3. $CL(Q_2)$ is in form <-- C_1 :[μ_1, ν_1] &. . . & C_n :[μ_n, ν_n], if dud only if C is in form C_1 :[μ_1, ν_1] &. . . & C_n :[μ_n, ν_n].

At the end of these syntactic transformations, the resulting program and query can be handled as well-behaved ELPS, where resolution-SLDe [28] can be applied.

6.2 Operational Semantics

Paralog inference engine offers an operational semantics for the implemented language; its execution is based on the resolution-SLDe method.

In this inference engine the selection function f maps target C_1 $[\mu_1, \nu_1]$ & . . . & C_n $[\mu_n, \nu_n]$, in literal C_i :$[\mu_i, \nu_i]$ where $[\mu_i, \nu_i] = \sup\{:[\mu_1, \nu_1], \ldots, [\mu_n, \nu_n]\}$. However, the selection function *f* employed by the Paralog inference engine is not the same standard selection function described in [20].

Also, the procedure to select the program clauses does not follow standard strategy. In this inference engine program clauses are selected so that the selected clause C:$[\mu, \nu]$ has evidence $[\mu, \nu]$ equal to the supreme of the set formed by the candidate clauses evidences.

These two selection strategies cause the inference engine refutation procedure to simulate a search similar to the best-first search in the program clauses $CL(P_2)$ refutation tree.

Thus, given a program P and a query Q, the Paralog e inference engine provides as an answer an evidence $[\mu, \nu]$, as previously described in this section, so that $[\mu, \nu] \in \tau$. In the cases in which $[\mu, \nu] \neq [0, 0]$, the answer also includes a replacement Θ for the variables of Q.

The following example shows a Paralog program and a query in the form of p(b) : [1.0, 0.0].

Example 6.1 A Paralog program

```
p(a): [1.0, 0.0].
p(x): [1.0, 0.0] <--
      q(x):[0.0, 1.0] &
      r(x): [1.0, 0.0].
r(a): [1.0, 0.0].
r(b): [1.0,0.0].
q(a): [0.0, 1.0].
q(b): [0.0,1.0].
```

In this example, the Paralog inference engine provides an evidence [1.0,0.0] for an answer. That means that the answer obtained is perfectly defined.

7 Programming in Paralog

The development of computationally efficient programs in Paralog must exploit two aspects in this language:

1. the declarative aspect that describes the logic structure of the problem, and
2. the procedural aspect that describes how the computer solves the problem.

However, it is not always an easy task to conciliate both aspects. Therefore, programs to be implemented in Paralog should be well defined to evidence both the declarative aspect and the procedural aspect of the language.

It must be pointed out that programs in Paralog, like programs in standard Prolog, may be easily understood or reduced - when well defined - by means of addition or elimination of clauses, respectively.

A small knowledge base in the domain of Medicine is presented as a Paralog program. The development of this small knowledge base was subsidized by the information provided by three experts in Medicine. The first two specialists - clinicians - provided six[3] diagnosis rules for two diseases: disease1 and disease2. The last specialist - a pathologist - provided information on four symptoms: symptom1, symptom2, symptom3 and symptom4. This example was adapted from da Costa and Subrahmanian's work [16].

Example 7.1 A small knowledge base in Medicine implemented in Paralog

```
disease1(X):  [1.0,  0.0]
<--
     symptom1(X):
[1.0,0.0] &
     symptom2(X):
[1.0,0.0]
disease2(X): [1.0,0.0] <-
-
     symptom1(X):
[1.0,0.0] &
     symptom3(X):
[1.0,0.0]
disease1(X): [0.0,1.0] <-
-
disease2(X): [1.0,0.0].
disease2(X): [0.0,1.0] <-
-
```

```
disease1(X): [1.0,0.0].
disease1(X): [1.0,0.0] <--
symptom1(X): [1.0,0.0] &
symptom4(X): [1.0,0.0] .
disease2(X): [1.0,0.0] <--
symptom1(X): [0.0,1.0] &
symptom3(X): [1.0,0.0] .
symptom1(john) : [1.0,0.0].
symptom1(bill): [0.0,1.0].
symptom2(john) : [0.0,1.0].
symptom2(bill): [0.0,1.0].
symptom3(john): [1.0,0.0].
symptom3(bill): [1.0,0.0].
symptom4(john): [1.0,0.0].
symptom4(bill): [0.0,1.0].
```

In this example, several types of queries can be performed. Table 1 below shows some query types, the evidences provided as answers by the Paralog inference engine and their respective meaning.

The knowledge base implemented in Example 7.1 may also be implemented in standard Prolog, as shown in Example 7.2.

[3] The first four diagnosis rules were supplied by the first expert clinician and the two remaining diagnosis rules were provided by the second expert clinician.

Item	Query and answer form		Meaning
1	Query	Disease1(bill):[1.0, 0.0]	Does Bill have disease 1 ?
	Evidence	[0.0, 0.0]	The information on Bill's disease1 is unknown
2	Query	Disease2(bill):[1.0, 0.0]	Does Bill have disease 2 ?
	Evidence	[1.0, 0.0]	Bill has disease2
3	Query	Disease1(john):[1.0, 0.0]	Does John have disease 1 ?
	Evidence	[1.0, 1.0]	The information on John's disease1 is inconsistent
4	Query	Disease2(john):[1.0, 0.0]	Does John have disease 2 ?
	Evidence	[1.0, 1.0]	The information on John's disease2 is inconsistent
5	Query	Disease1(bob):[1.0, 0.0]	Does Bob have disease 1 ?
	Evidence	[0.0, 0.0]	The information on Bob's disease1 is unknown

Table 1 Query and answer forms in Paralog

Example 7.2 Knowledge base of Example 7.1 implemented in standard Prolog

```
disease1(X) :-                    disease1(X) :-
symptom1(X),                      symptom1(X),
symptom2(X).                      symptom4(X).
disease2(X):-                     disease2(X) :-
symptom1(X),                      not symptom1(X),
symptom3(X).                      symptom3(X).
disease1 (X):-                    symptom1 (john) .
not disease2(X).                  symptom3( john) .
disease2(X) :-                    symptom3(bill).
not disease1(X).                  symptom4(john)
```

In this example, several types of queries can be performed as well. Table 2 shows some query types provided as answers by the standard Prolog and their respective meaning.

Item	Query and answer form		Meaning
1	Query	Disease1(bill)	Does Bill have disease 1 ?
	Answer	Loop	System enters into an infinite loop
2	Query	Disease2(bill)	Does Bill have disease 2 ?
	Answer	Loop	System enters into an infinite loop
3	Query	Disease1(john)	Does John have disease 1 ?
	Answer	Yes	John has disease1
4	Query	Disease2(john)	Does John have disease 2 ?
	Answer	Yes	John has disease2
5	Query	Disease1(bob)	Does Bob have disease 1 ?
	Answer	No	Bob does not have disease1

Table 2 Query and answer forms in standard Prolog

Starting from Examples 7.1 and 7.2 it can be seen that there are different characteristics between implementing and consulting in Paralog and standard Prolog. Among these characteristics, the most important are:
1. the semantic characteristic; and
2. the execution control characteristic.

The first characteristic may be intuitively observed when the program codes in Examples 7.1and 7.2 are placed side by side. That is, when compared to Paralog, the standard Prolog representation causes loss of semantic information on facts and rules. This is due to the fact that standard Prolog cannot directly represent the negation of facts and rules.

In Example 7.1, Paralog program presents a four-valued evidence representation. However, the information loss may be greater for a standard Prolog program, if the facts and rules of Paralog use the intermediate evidence of lattice $\tau = \{x \in \Re | \ 0 \leq x \leq 1\} \times \{x \in \Re | \ 0 \leq x \leq 1\}$. This last characteristic may be observed in Tables 1 and 2. These two tables show five queries and answers, presented and obtained both in Paralog and standard Prolog program.

The answers obtained from the two approaches present major differences. That is, to the first query: "Does Bill have disease1?", Paralog answers that the information on Bill's disease1 is unknown, while the standard Prolog enters into a loop. This happens because the standard Prolog inference engine depends on the ordination of facts and rules to reach deductions. This, for standard Prolog to be able to deduct an answer similar to Paralog, the facts and rules in Example 7.2 should be reordered. On the other hand, as the Paralog inference engine does not depend on reordering facts and rules, such reordering becomes unnecessary.

In the second query: "Does Bill have disease2?", Paralog answers that "Bill has disease2", while the standard Prolog enters into a loop. This happens for the same reasons explained in the foregoing item.

In the third query: "Does John have disease1?", Paralog answers that the information on John's disease1 is inconsistent, while the standard Prolog answers that "John has disease1". This happens because the standard Prolog inference engine, after reaching the conclusion that "John has disease1" does not check whether there are other conclusions leading to a contraction. On the other hand, Paralog performs such check, leading to more appropriate conclusions.

In the fourth query: "Does John have disease2?", Paralog answers that the information on John's disease2 is inconsistent, while the standard Prolog answers that "John has disease2". This happens for the same reasons explained in the foregoing item.

In the last query: "Does Bob have diseasel", Paralog e answers that the information on Bob's diseasel is unknown, while the standard Prolog answers that "Bob does not have diseasel". This happens because the standard Prolog inference engine does not distinguish the two possible interpretations for the answer not. On the other hand, the Paralog inference engine, being based on an infinitely valued paraconsistent evidential logic, allows the distinction to be made.

In view of the above, it is demonstrated that the use of the Paralog language may handle several Computer Science questions more naturally.

8 Conclusions

Inconsistency is a natural phenomenon arising from the description of the real world. This phenomenon may be encountered in several situations. Nevertheless, human beings are capable of reasoning adequately. The automation of such reasoning requires the development of formal theories. Paraconsistent Logic, despite having been initially developed from the purely theoretical standy point, found in recent years extremely fertile applications in Computer Science [11], [21], [22], [23], [24] thus solving the problem of justifying such logic systems from the practical standpoint.

Languages such as Paralog, capable of merging Classical Logic Programming concepts with those of inconsistency, widen the scope of Logic Programming applications in environments presenting conflicting beliefs and contradictory information.

Acknowledgements. The authors would like to thank the anonymous referees for the many helpful comments and suggestions provided. The first author is supported by FAPESP grant 97/02328-9.

References

[1] Abe, .J.M., Fundamentos da Lógica Anotada (Foundations of Annotated Logics), Ph.D. Thesis, Universidade de São Paulo, São Paulo, Brazil, 1992 (in Portuguese).

[2] Abe, J.M., On Annotated Model Theory, *Coleção Documentos, Série: Lógica e Teoria da Ciência* n_o 11, Instituto de Estudos Avançados, University of São Paulo, São Paulo, Brazil, June, 1993.

[3] Abe, J.M., On Annotated Modal Logics, *Mathematica Japonica*, 40, n_o 3, 553-560, 1994.

[4] Abe, J.M., J.P.A. Prado & B.C. Ávila, On a class of paraconsistent multimodal systems for reasoning, *Coleção Documentos, Série Lógica e Teoria da Ciência*, IEA-USP, n_o 24, 12p, 1997.

[5] Abe, J.M. & S. Akama, Annotated logics $Q\tau$ and ultraproducts, to appear in *Logique et Analyse*, 1999.

[6] Abe, J.M., Curry algebras $P\tau$, to appear in *Logique et Analyse*, 1999.

[7] Abe, J.M. & S. Akama, A Logical System for Reasoning with Fuzziness and Inconsistencies, Proceedings of the IASTED *International Conference on Artificial Intelligence and Soft Computing* (ASC'99), August 9-12, Honolulu, Hawaii, USA, 221-225, 1999.

[8] Akama, S. & J.M. Abe, Natural Deduction And General Annotated Logics, Proceedings, *The First International Workshop on Labeled Deduction* (LD'98), Freiburg, Germany, 1-14, 1998.

[9] Akama, S. & J.M. Abe, Many-valued and annotated modal logics, IEEE 1998 *International Symposium on Multiple-Valued Logic* (ISMVL'98), Proceedings, pp. 114-119, Fukuoka, Japan, 1998.

[10] Abe, J.M., S. Akama & R. Sylvan, General annotated logics, with an introduction to full accounting logic, *Coleção Documentos, Série Lógica e Teoria da Ciência*, IEA-USP, n_o 39, 17p., 1998.

[11] Ávila, B.C., *An Evidential Logic-Based Paraconsistent Approach to Handle Exceptions in Multiple Inheritance Frame Systems*, Ph.D. Thesis, University of São Paulo, São Paulo, Brazil, 1996, 133 p., (in Portuguese).

[12] Belnap, N.D., *A Useful Four- Valued Logic*, in G. Epstein dud J.M. Dunn, eds. Modern Uses of Many-Valued Logic (D. Reidel, 1979), pp. 8-37.

[13] Blair, H.A. & Subrahmanian, V.S., Paraconsistent Logic Programming, Proc. 7th Conference on Foundations of Software Technology and Theoretical Computer Science, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 287, pp. 340-360, 1987.

[14] Blair, H.A. & Subrahmanian, V.S., Paraconsistent Foundations for Logic Programming, *Journal of Non-Classical Logic*, 5, 2, pp. 45-73, 1988.

[15] Clocksin, W.F.; Mellish, C.S., *Programming in Prolog* (3rd Ed.), Springer-Verlag, 1987.

[16] da Costa, N.C.A. & Subrahmanian, V.S., Paraconsistent Logics as a Formalism for Reasoning About Inconsistent Knowledge Bases, *Artificial Intelligence in Medicine* I, pp. 167-174, Burgverlag Tecklenburg, West Germany, 1989.

[17] da Costa, N.C.A.; Henschen, L.J.; Lu, .J..J. & Subrahmanian, V.S., Automatic Theorem Proving in Paraconsistent Logics: Theory and Implementation, Proc. 10th International Conference on Automated Deduction, *Lecture Notes in Computer Science*, Vol. 449, pp. 72-86, 1990.

[18] da Costa, N.C.A.; Abe, J.M. & Subrahmanian, V.S., Remarks on Annotated Logic, *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, Vol. 37, pp. 561-570, 1991.

[19] Fitting, M.C., A Kripke-Kleene Semantics for Logic Programming, *J. of Logic Prog.* 4 (1985), pp. 295-312.

[20] Lloyd, J.W., *Foundations of Logic Programming*, Springer-Verlag, 1984.

[21] Nakamatsu, K., J.M. Abe & A. Suzuki, An approximate reasoning in a framework of vector annotated logic programming, *The Vietnam-Japan Bilateral Symposium on Fuzzy Systems And Applications*, VJFUZZY' 98, Nguyen H. Phuong & Ario Ohsato (Eds), HaLong Bay, Vietnam, 521-528, 1998.

[22] Nakamatsu, K. & J.M. Abe, Reasonings Based On Vector Annotated Logic Programs, Proc. of CIMCA'99, *International Conference on Computational Intelligence for Modeling Control and Automation*, Edited by M. Mohammadian, IOS Press – Ohmsha, ISBN 90 5199 474 5 (IOS Press), Netherlands, 396-403, 1999.
[23] Nakamatsu, K., J.M. Abe & A. Suzuki, "Defeasible Reasoning Between Conflicting Agents Based on VALPSN", *American Association for Artificial Intelligence* - AAAI'99 Workshop on Agents' Conflicts, ISBN 1-57735-092-8, TR WS-99-08, AAAI Press – American Association for Artificial Intelligence, Menlo Park, California, USA, 20-27, 1999.
[24] Nakamatsu, K., Y. Hasegawa, J.M. Abe & A. Suzuki, A Framework for Intelligent Systems Based on Vector Annotated Logic Programs, IPMM'99 *The Second International Conference on Intelligent Processing and Manufacturing of Materials*, ISBN 0-7803-5489-3, Editors: J.A. Meech, M.M. Veiga, M.H. Smith & S.R. LeClair, IEEE Catalogue Number: 99EX296, Library of Congress Number: 99-61516, Honolulu, Hawaii, USA, 695-702, 1999.
[25] Ng, R.T. & Subrahmanian, V.S., *Relating Dempster-Shafer Theory to Stable Semantics*, CS-TR-2647,University of Maryland, 1991, 40 pg.
[26] Shafer, G., *A Mathematical Theory of Evidence*, Princeton University Press, 1976.
[27] Subrahmanian, V.S., On the Semantics of Quantitative Logic Programs, *Proc. 4 IEEE Symp. On Logic Programming*, Computer Society Press, San Francisco, Sep., 173-182, 1987.
[28] Subrahmanian, V.S., Towards a Theory of Evidential Reasoning, in *Logic Programming, Logic Colloquium* '87, The European Summer Meeting of the Association for Symbolic Logic, Granada, Spain, July, 1987.
[29] Sylvan, R. & J.M. Abe, On general annotated logics, with an introduction to full accounting logics, *Bulletin of Symbolic Logic*, 2, 118-119, 1996.
[30] van Emden, M.H., Quantitative Deduction and its Fixpoint Theory, *J. of Logic Programming*, 4, 1, pp. 37-53, 1986.