International Journal of Computational Intelligence Systems, Vol. 4, No. 5 (September, 2011), 977-990

Splitting Computation of Answer Set Program and Its Application on E-service*

Bo Yang
College of Computer Science and Information, Guizhou University, Guiyang, 550025, China

Department of Physics and Electronics Information Science, Guiyang University, Guiyang, 550005, China

Ying Zhang, Mingyi Zhang
Guizhou Academy of Science, Guiyang, 550001, China

Maonian Wu
College of Science, Guizhou University, Guiyang, 550025, China

E-mail: gzu_wu@yahoo.com

Received 2 March 2011
Accepted 3 November 2011

Abstract

As a primary means for representing and reasoning about knowledge, Answer Set Programming (ASP) has been applying in many areas such as planning, decision making, fault diagnosing and increasingly prevalent e-service. Based on the stable model semantics of logic programming, ASP can be used to solve various combinatorial search problems by finding the answer sets of logic programs which declaratively describe the problems. It's not an easy task to compute answer sets of a logic program using Gelfond and Lifschitz's definition directly. In this paper, we show some results on characterization of answer sets of a logic program with constraints, and propose a way to split a program into several non-intersecting parts step by step, thus the computation of answer sets for every subprogram becomes relatively easy. To instantiate our splitting computation theory, an example about personalized product configuration in e-retailing is given to show the effectiveness of our method.

Keywords: logic program, answer set, splitting, E-service

1. Introduction

The Internet has been reaching almost all aspects of our lives, many online services emerged as the times require, including e-government, e-business, e-learning, e-commerce, e-recruitment, and so on. Many Artificial Intelligent(AI) techniques got successful application in the field of e-services. Lu *et al* offered a comprehensive and systematic survey on the new field of e-service intelligence, which deals with fundamental roles, social impacts and practical applications of various intelligent technologies on the Internet based e-service applications.[1] In all subfields of e-services, AI technologies including expert systems, machine learning,

*This is an extended version of a paper presented at the FLINS2010, August 2-4, 2010, Chengdu, China.

artificial neural networks, fuzzy systems *etc.* are playing useful and vital roles. Many intelligent technologies mentioned above involve inductive or deductive reasoning based on known information, which is usually imprecise or incomplete. To deal with imprecise data, fuzzy reasoning is a powerful tool. As for incomplete knowledge, resorting to commonsense reasoning is the most suitable choice. Actually, more than 90% knowledge is commonsense in practical applications. Hence commonsense knowledge representing and reasoning has been being the kernel problem and primary challenge for AI.

Non-monotonicity is the most important feature of commonsense knowledge representing and reasoning. As a primary means for non-monotonic reasoning, Answer Set Programming(ASP) is a paradigm based on the stable model(answer set) semantics of logic programming,[2] it is a method that reduces solving of various combinatorial search problems to finding the answer sets of logic programs which declaratively describe the problems. ASP has been applied successfully in areas like decision making, planning, e-commerce. In Ref.3, A-Prolog is used to build a medium size decision support system, in which operations of a fairly complex subsystem of the Space Shuttle are modeled. Paschke *et al* presented a logical formalism ContractLog for the representation and enforcement of Service Level Agreement(SLA) rules between IT service providers and their customers.[4] In this framework Extended Logic Programming(ELP) plays a very important role for deductive reasoning on SLA rules. Eiter *et al* introduced a new declarative language $\mathcal{K}$ based on non-monotonic logic programming.[5] Transitions between states of knowledge can be described in $\mathcal{K}$, so it is suitable for planning under incomplete knowledge. Tu *et al* described the methodology for developing several conformant planners for a given dynamic domain,[6] one of them is logic programming based and can generate parallel plan. Tiihonen *et al* created a web-based product configurator that provides intelligent support for tailoring a product through applying an inference engine for the form of logic program.[7]In the context of e-commerce, this tool can be used to provide personalized service, an important subfield of increasingly prevalent e-service.

However, to the best of our knowledge, there are few reports on how to deal with incomplete knowledge in an e-service system. A main cause led to it is that representation and reasoning of incomplete knowledge is very hard and complex. ASP provides a useful approach, but to find all answer sets of a logic program is a problem with comparative complexity. As Dantsin *et al* have shown, logic program under stable model semantics is co-NP-complete.[8]

Splitting is very helpful for simplifying answer sets solving. Lifschitz *et al* gave a conceptual description of splitting,[9,10] in which a set U of literals should be given to generate a base of a program with respect to U. Moreover, the notion of U is extended to splitting a program in series through a monotonic and continuous splitting sequence. In accordance with the original definition of splitting, Turner and Watson addressed Splitting Set Theorem for default theories[11] and epistemic specifications[12] respectively, and Balduccini extended the splitting to programs with consistency-restoring rules.[13] However, none of them pointed out how to construct a suitable set U of literals for splitting a program. In principle, the splitting process always starts from "guessing" an appropriate set U of literals.

Instead of guessing a set U of literals that can split a program, it is more interesting to find a computable way to split a program such that the complexity of answer sets solving can be reduced. Zhang presented "constructive" characterizations for extensions of a default theory and for answer sets of a logic program,[14,15,16] which imply the idea of splitting a default theory (program) into a sequence of default sub-theories (subprograms). And Wu *et al* discussed a method of splitting based on entire set of atoms for Horn logic, a special style of ASP, for the aim of belief revision over Horn logic.[17,18] These works motive us to explore a characterization of answer sets of a logic program with constraints and to propose a stepwise way of splitting. Based on this, a program can be split into subprograms and every subprogram have less rules so that it is easier to compute their answer sets, and the union of answer sets of every subprogram is the answer sets of the original program, neither more nor less than.

Hence we can more easily represent and compute problems perhaps coming with incomplete information from e-service, and other application areas, by ASP. In this paper a simplified example about personalized e-retailing is given to show that ASP provides a natural and compact description for personalizing product

configuration according to customer's favor, which could include incomplete information.

This paper is organized as follows. Section 2 recalls some notions and notations of ASP. Our definition and theoretic results about characterization of answer set are given in Sec.3. Section 4 describes our method for splitting a program and corresponding algorithm. In Sec.5, an example about personalized product configuration in e-retailing is given to justify our splitting algorithm and its advantage. The last section concludes our work and presents the future research interests.

2. Preliminaries

2.1. *Syntax and semantics of logic program*

According to Lifschitz,[10] we consider the alphabet $\boldsymbol{A}\cup\{,\ ,\ \neg,\ \leftarrow,\ \text{not}\}$ in this paper. The nonempty set of symbols $\boldsymbol{A}$ and the set $\{,\ ,\ \neg,\ \leftarrow,\ \text{not}\}$ are disjoint. An element from $\boldsymbol{A}$ is called *atom*. The symbols ",", "¬", "←" and "not" mean "conjunction", "classical negation", "if" and "negation as failure" respectively. Terms defined in this section come from Ref. 10 primarily.

Definition 1. *A positive literal is an atom and a negative literal is an atom preceded by the classical negation symbol "¬". A literal is a positive literal or negative literal.*

Literals L and ¬L are said to be *complementary*. A set of literals is *inconsistent* if it contains a pair of complementary literals, and *consistent* otherwise. By *Lit* we denote the set of all literals.

Definition 2. *A rule r is of the form:*

$$Head \leftarrow L_1,\ldots, L_m, not\ L_{m+1},\ldots, not\ L_{m+k}$$

where Head (head of r, donated by H(r)) is empty or a literal L_0, and the right hand side of r is a finite set of two kinds of rule elements, i.e. literals possibly preceded by the negation as failure symbol "not".

We also write rule r in a brief form:

$$H(r)\leftarrow P(r)\cup \text{not}(N(r))$$

where $P(r)=\{L_1,\ldots, L_m\}$ is called *positive body*, and $N(r)=\{L_{m+1},\ldots, L_{m+k}\}$ *negative body* respectively. Especially, a rule with $H(r)\neq\emptyset$ and $N(r)=\emptyset$ is a *basic* rule. Rule r is called a *fact* if $H(r)\neq\emptyset$ and $P(r)=N(r)=\emptyset$, and a *constraint* if $H(r)=\emptyset$. To distinguish from ordinary rules, for a constraint c, its positive body and negative body are represented as $P(c)$ and $N(c)$ respectively. Thus constraint c is of the form:

$$\leftarrow P(c), \text{not } (N(c)).$$

A set X of literals *satisfies* a constraint c if $P(c)\not\subseteq X$ or $N(c)\cap X\neq\emptyset$; X satisfies a set C_Π of constraints if X satisfies each c in C_Π.

Definition 3. *A program Π is a set of rules, and Π is a basic program if every rule in it is basic.*

A program with constraints can be written as $\Pi=\Pi^*\cup C_\Pi$, where Π^* contains no constraint and C_Π is a set of constraints. By $H(\Pi)$ we denote the set of heads of all rules in Π, *i.e.*, $H(\Pi)=\{H(r)|r\in\Pi\}$. And $N(\Pi)$ is the set of negative bodies of all rules in Π.

Note that an atom is understood here as in propositional logic, however, in application it is usually an atomic sentence formed with object, function and predicate constant. Actually, each atom in $\boldsymbol{A}$ containing variables stands for a set of ground atoms, which are gotten by ground instantiation. Rules in a program are often represented by schemata containing variables. In Example 2 we will see how schematic rules are grounded[10] and treated as propositional logic.

To explain the answer set semantics of an arbitrary program, we start from the notion *consequences* of a basic program.

A set X of literals is *logically closed* if it is consistent or equals *Lit*. Given a basic program Π, X is *closed under* Π if for each rule r: $H(r)\leftarrow P(r)$ in Π, $H(r)\in X$ whenever $P(r)\subseteq X$. It is easy to see that *Lit* is logically closed and closed under any basic program Π. Among all sets of literals which are logically closed and closed under Π, we are interested in the smallest one, denoted by $Cn(\Pi)$. Clearly, such a set always exist.

Definition 4. *Given a basic program Π, elements of $Cn(\Pi)$ are called the consequences of Π. And $Cn(\Pi)$ is called the consequence or stable model of Π.*

For any basic program Π and a set X of literals, to compute $Cn(\Pi)$, a monotonic function T_Π is defined as follows:

$T_\Pi X$ is $\{H(r)|H(r)\leftarrow P(r)\in\Pi,\ P(r)\subseteq X\}$ if X is consistent, and *Lit* otherwise. $Cn(\Pi)$ is the union of sets obtained by iterating T_Π on $\emptyset$, that is, $Cn(\Pi)=\cup_{n\geq 0}T_\Pi{}^n\emptyset$, where $T_\Pi{}^0\emptyset=\emptyset$. Consider an example from Ref. 10:

Example 1 Let Π is:

{ p. ¬q.
r←p, q.

$\neg r \leftarrow p, \neg q.$
$s \leftarrow r.$
$s \leftarrow p, s.$
$\neg s \leftarrow p, \neg q, \neg r. \}$

where each rule is ended by ".". By definition of T_Π, We have:

$T_\Pi{}^0\varnothing = \varnothing$
$T_\Pi{}^1\varnothing = \{p, \neg q\}$
$T_\Pi{}^2\varnothing = \{p, \neg q, \neg r\}$
$T_\Pi{}^3\varnothing = \{ p, \neg q, \neg r, \neg s \}$

For every $n>3$,

$T_\Pi{}^n\varnothing = T_\Pi{}^3\varnothing$

Thus

$Cn(\Pi) = \cup_{n\geq 0} T_\Pi{}^n\varnothing$
$= \{ p, \neg q, \neg r, \neg s \}$

A basic program Π is consistent if $Cn(\Pi)$ is consistent, and if $Cn(\Pi)$ is inconsistent, then Π is inconsistent too.

In order to give the notion of answer sets of an arbitrary program, it is necessary to introduce the notion of *reduct*.

Given an arbitrary program $\Pi = \Pi^* \cup C_\Pi$ and a set X of literals, the reduct of Π relative to X, Π^X, is derived by

- deleting all $c \in C_\Pi$,
- deleting each rule $H(r) \leftarrow P(r) \cup not\ (N(r)) \in \Pi^*$ such that $N(r) \cap X \neq \varnothing$, and
- replacing each remaining rule $H(r) \leftarrow P(r) \cup not\ (N(r)) \in \Pi^*$ by $H(r) \leftarrow P(r)$.

Definition 5. *A set X of literals is an answer set of program $\Pi = \Pi^* \cup C_\Pi$ if $Cn(\Pi^X) = X$ and X satisfies C_Π.*

It is obvious that X is also an answer set of Π^*.

Definiton 6. *Given an answer set X of a program $\Pi = \Pi^* \cup C_\Pi$, the set $GR(X, \Pi)$ of generating rules of X is defined as $GR(X, \Pi) = \{r \in \Pi^* \mid P(r) \subseteq X, N(r) \cap X = \varnothing,$ and X satisfies $C_\Pi\}$.*

Clearly, $H(GR(X, \Pi)$ satisfies C_Π.

We say that a program $\Pi = \Pi^* \cup C_\Pi$ is consistent if it has a consistent answer set; it is inconsistent if one of its answer sets is inconsistent. In Sec.3 we will see that these notions are well-defined.

In general, an arbitrary program Π satisfies exactly one of the following conditions:[10]

- Π has no answer set;
- the only answer set for Π is *Lit*;
- Π has at least one answer set, and all its answer set(s) are consistent.

The following example about *n-coloring* of a graph G illustrates various situations in which whether answer set(s) exist or not. And it also shows actual application of ASP in Graph Theory.

Example 2 The problem of n-coloring of a graph G refers to finding a color schemes of *n* colors for every vertex of G such that for every pair of adjacent vertices (*X*,*Y*) in G, color of *X* is different from that of Y.

Predicate c(*I*) is used to represent that *I* is a color, where variable *I* ranges over the set of colors C={1, …, n}. Predicates ver(*V*) and edge(*V*,*W*) denote that *V* is a vertex of graph G and vertices *V*, *W* are adjacent respectively. By color(*V*,*I*) we mean that vertex *V* is dyed with color *I*. Then the problem can be described by the following schematic rules containing variables:

$1\{color(V,I): c(I)\}1 \leftarrow ver(V)$ (1)
$\leftarrow color(V,I), color(W,I), edge(V,W), c(I)$ (2)

Rules like (1) is a "choice rule" with numerical bounds, by which cardinality of consequence sets from this rule is restricted in a certain scope.[19,20] Numerals before and after the brace are called "lower bound" and "upper bound" respectively. Rule (1) says: if *V* is a vertex of graph G, then from all possible colors c(*I*), choose at least one and at most one color *I* to make color(*V*,*I*) holds. As a matter of fact, rule (1) can be viewed as an abbreviation of a set of rules containing the negation as failure symbol "not". In the aftermentioned process of grounding,[10] we will see the grounded forms of rule (1) and (2). Rule (2) indicates that any two adjacent vertices can not be dyed with same color.

If there are two colors, say 1 and 2, are used, and graph G is a rectangle with four vertices v_0, v_1, v_2 and v_3, the following facts describe the used colors and the structure of G:

F={c(1). c(2).
ver(v_0). ver(v_1). ver(v_2). ver(v_3).
edge(v_0,v_1). edge(v_1,v_2).
edge(v_2,v_3). edge(v_3,v_0).}

Based on F, rule (1) is grounded to a collection of rules without variables as follows:

G_1={color(v_0,1)←ver(v_0), c(1), c(2), not color(v_0,2).
color(v_0,2)←ver(v_0), c(1), c(2), not color(v_0,1).
color(v_1,1)←ver(v_1), c(1), c(2), not color(v_1,2).
color(v_1,2)←ver(v_1), c(1), c(2), not color(v_1,1).
color(v_2,1)←ver(v_2), c(1), c(2), not color(v_2,2).
color(v_2,2)←ver(v_2), c(1), c(2), not color(v_2,1).

color(v_3,1)←ver(v_3), c(1), c(2), not color(v_3,2).
color(v_3,2)←ver(v_3), c(1), c(2), not color(v_3,1). }

And rule (2) is grounded as:

G_2={←color(v_0,1), color(v_1,1), edge(v_0,v_1), c(1).
←color(v_0,2), color(v_1,2), edge(v_0,v_1), c(2).
←color(v_1,1), color(v_2,1), edge(v_1,v_2), c(1).
←color(v_1,2), color(v_2,2), edge(v_1,v_2), c(2).
←color(v_2,1), color(v_3,1), edge(v_2,v_3), c(1).
←color(v_2,2), color(v_3,2), edge(v_2,v_3), c(2).
←color(v_3,1), color(v_0,1), edge(v_3,v_0), c(1).
←color(v_3,2), color(v_0,2), edge(v_3,v_0), c(2). }

Let Π_1=F∪G_1∪G_2, consider a set of literals X= F∪{color(v_0,1), color(v_1,2), color(v_2,1), color(v_3,2)}, then

Π_1^X= F∪{color(v_0,1)←ver(v_0), c(1), c(2).
color(v_1,2)←ver(v_1), c(1), c(2).
color(v_2,1)←ver(v_2), c(1), c(2).
color(v_3,2)←ver(v_3), c(1), c(2). }

It is obvious that Cn(Π_1^X)=F ∪ {color(v_0,1), color(v_1, 2), color(v_2,1), color(v_3,2)}=X, and for each constraint *c* in G_2, P(*c*)⊈X. Thus, X is an answer set of Π_1. Similarly, it is easy to verify that F ∪ {color(v_0,2), color(v_1,1), color(v_2,2),color(v_3,1)} is another answer set of Π_1.

Adding a new edge into graph G will lead to completely different result. Suppose that {edge(v_1,v_3)} is added into F, then the following two grounded rules will appear in G_2:

←color(v_1,1), color(v_3,1), edge(v_1,v_3), c(1).
←color(v_1,2), color(v_3,2), edge(v_1,v_3), c(2).

Now it is impossible to find a set X of literals such that Cn(Π_1^X)=X and for each constraint *c* in G_2, X satisfies *c*, then Π_1 has no answer set in this case.

Obviously, the more number of vertices or colors, the more complicated the grounded forms of rule (1) and (2) are, and hence make the task of solving answer set more difficult. In Sec.4, we will see that splitting a program is very helpful for simplifying the computation of answer set.

Example 3 shows a program with *Lit* as its answer set. For simplicity of presentation, remaining examples in this section are written in propositional language.

Example 3 Let Π_2 is:

{ a. ¬b.
c←¬b .
d←c, not b.
¬d←c,b
b←a, c.
←not b. }

It is clear that Cn({r∈Π_2^*| N(r)=∅}) contains complementary literals b and ¬b, *i.e.* Cn({r∈Π_2^*| N(r)=∅})=*Lit*, and for constraint c':

←not b.

N(c')∩*Lit*≠∅. So Π_2 is inconsistent, its unique answer set is *Lit*.

2.2. *Splitting*

In Ref.10, Lifschitz gave the original definition of splitting.

Definition 7. *For any program Π no containing constraints, any set U of literals, we say that U splits Π if for every rule H(r)←P(r)∪not(N(r)) in Π, P(r)∪N(r) ⊆U whenever H(r) ∈U.*

By $b_U(\Pi)$ we denote the set of rules in Π whose heads belong to U, the *base* of Π(relative to U). And for any C⊆U, $e_U(\Pi,C)$ stands for the program obtained from Π by

- deleting each rule H(r)←P(r)∪not(N(r)) such that P(r)∩(U\C)≠∅ or N(r)∩C≠∅,
- replacing each remaining rule H(r)←P(r)∪not(N(r)) by H(r)←(P(r)\U)∪not (N(r)\U).

The following result from Ref.10 shows the effect of splitting.

Theorem 1.[10] *Let U be a set of literals that splits a program Π. A consistent set of literals is an answer set for Π if and only if it can be represented in the form C_1∪C_2, where C_1 is an answer set for $b_U(\Pi)$ and C_2 is an answer set for $e_U(\Pi\backslash b_U(\Pi),C_1)$.*

Example 4 Consider the following program:

Π_3={a.
b←a, not c.
c←a, not d. }

Let U={a,d}, then $b_U(\Pi_3)$ is {a.}, and the only answer set of $b_U(\Pi_3)$ is C_1={a}. Furthermore, $\Pi_3\backslash b_U(\Pi_3)$ is:

{ b←a, not c.
c←a, not d. }

And $e_U(\Pi_3\backslash b_U(\Pi_3), C_1)$ is:

{ b←not c.
c. }

Obviously C_2={c} is the only answer set of the program $e_U(\Pi_3\backslash b_U(\Pi_3),C_1)$. It is easy to verify that C_1∪C_2={a,c} is the unique answer set of Π_3.

If U_1={a, c, d}, then, clearly, it is also a set of literals that can split Π_3. In fact:

$b_{U1}(\Pi_3)$ ={a.

c←a, not d.}

The only answer set of $b_{U1}(\Pi_3)$ is C_1'={a, c}. So $e_{U1}(\Pi_3 \backslash b_{U1}(\Pi_3), C_1')$ is ∅, whose answer set is ∅ too. Obviously, $C_1' \cup \emptyset$={a,c} is also the answer set of Π_3.

Note that some subsets of {a, b, c, d} split Π_3 and give the same answer set {a, c}, but others, e.g., {a, b} can not split Π_3.

Splitting is very useful for computing answer sets. However, it is not so convenient to "guess" a suitable initial splitting set U(in Example 4 we need test 2^4 subsets of {a, b, c, d}). To deal with this puzzle, we want to find a tractable method, which splits a big program Π into several subprograms such that solving answer sets for smaller subprograms is easier and the sum of answer sets of subprograms is just an answer set of Π. This is what this paper aims at.

By defining the concepts of compatibility and auto-compatibility for general default theories, Zhang presented a simple and natural characterization of extensions of general default theories and developed a class of default theory, named auto-compatible default theory.[14,15] Results about default theories can be easily transformed to ASP which is also a kind of formalism for non-monotonic inference. In Ref.16, Zhang *et al* proposed a finite characterization of answer sets for nested program, which is very helpful for exploring existence of answer sets. In particular, for any given finite program Π, each answer set of Π can be represented by a finite set of generating rules that captured by a Λ-operator and notion of compatibility. Following ideas mentioned, we present some notions and results for computing answer set in terms of splitting in the next section.

3. Λ -operator and Characterization of Answer Set

Intuitively, for all rules applicable to generating an answer set, their heads would be disjoint with their negative bodies, and would satisfy any constraint. Therefore, we introduce the concept of *compatibility*, which characters a necessary condition for the negative body of any applicable rule when generating an answer set.

Definition 8. *A program $\Pi=\Pi^* \cup C_\Pi$ is compatible if $H(\Pi^*) \cap N(\Pi^*)=\emptyset$ and for each $c \in C_\Pi$, $P(c) \not\subseteq H(\Pi^*)$ or $H(\Pi^*) \cap N(c) \neq \emptyset$. In particular, the empty program $\emptyset$ is compatible.*

Clearly, a basic program is compatible, and any nonempty set of constraints is incompatible. A class of subprograms of compatible program is also compatible, that is what the following proposition says.

Proposition 1. *If program $\Pi=\Pi^* \cup C_\Pi$ is compatible then any $\Pi' \subseteq \Pi^*$ is also compatible.*

A rule can be used in the process of solving answer set if its positive body are facts or can be derived from facts step by step. In a way similar to the operator T_Π mentioned in Sec.2, we define Λ-operator, which characters another necessary condition for the positive body of any acceptable rule, as follows.

Definition 9. *For a program $\Pi=\Pi^* \cup C_\Pi$, let $\Pi'=\{ r \in \Pi^* \mid N(r)=\emptyset\}$, then $\Lambda(\Pi)=\cup_{0\leq n}\Pi^n$ or $\perp$ when $\Pi^{n+1}=\perp$ for some $n\geq 0$, where Π^0 is defined as Eq.(1). And For $n\geq 0$ and $\Pi^0 \neq \Pi_{Lit}$, Π^n is given as Eq.(2), where $\perp$ stands for undefined and Π_{Lit} is a special program with Lit as its answer set.*

Note that if $H(\{r \in \Pi - C_\Pi | P(r) \subseteq H(\Pi^n)\})$ is consistent then $\Pi^n \neq \Pi_{Lit}$ and $\Pi^n \subseteq \Pi^{n+1}$ for any $n\geq 0$ and $\Pi^{n+1} \neq \perp$. In what follows we always consider only the case where $\Lambda(\Pi) \neq \perp$.

From Def.9, it is obvious that the answer set of a basic program is just its stable model, *i.e.*. if $N(\Pi)=\emptyset$ and $C_\Pi=\emptyset$, then $H(\Lambda(\Pi))$ is the stable model of Π and $H(\Lambda(\Pi)) = \cup_{0\leq n} T_\Pi{}^n \emptyset$.

Example 5 $\Lambda(C_\Pi)=\emptyset$ for any nonempty set of constraints C_Π.

From Def.9 we immediately get some important properties for the operator Λ: monotonicity and idempotence *etc.* Some proofs for these properties see Appendix.

Proposition 2. *If $\Lambda(\Pi) \neq \Pi_{Lit}$ and $\Lambda(\Pi) \neq \perp$ then $\Lambda(\Pi) \subseteq \Pi^*$ and $H(\Lambda(\Pi))$ satisfies C_Π.*

Proposition 3. *Λ is monotonic, i.e., if $\Pi_1 \subseteq \Pi_2$ and $\Lambda(\Pi_2) \neq \perp$, then $\Lambda(\Pi_1) \subseteq \Lambda(\Pi_2)$.*

Lemma 1. *If $\Lambda(\Pi) \neq \Pi_{Lit}$ and $\Lambda(\Pi) \neq \perp$ then $\Lambda(\Lambda(\Pi))=\Lambda(\Pi)$.*

Lemma 2. *If $\Lambda(\Pi) \neq \Pi_{Lit}$ and $\Lambda(\Pi) \neq \perp$ then for any $r \in \Pi^*$, $P(r) \subseteq H(\Lambda(\Pi))$ if and only if $r \in \Lambda(\Pi)$.*

Theorem 2. *Program $\Pi=\Pi^* \cup C_\Pi$ has an inconsistent answer set if and only if $Cn(\{r \in \Pi^* | N(r)=\emptyset\})=Lit$ and $N(c) \neq \emptyset$ for any $c \in C_\Pi$.*

Corollary 1. *If Lit is an answer set of program Π, then it is the unique answer set of Π.*

By Theorem 2 and Corollary 1, we are interested in only consistent programs. How to determine whether a program has a consistent answer set or not and how to compute its answer sets if they exist? This is intractable by the definition of answer sets since it is needed to test all consistent sets of literals. Now we establish a characterization of answer sets of a consistent program, by which computing answer set is based on only this program itself.

Theorem 3. *Program $\Pi = \Pi^* \cup C_\Pi$ has a consistent answer set if and only if there is a subset Π' of Π^* such that:*
(i) $\Pi' \cup C_\Pi$ is compatible;
(ii) $\Lambda(\Pi') = \Pi'$;
(iii) For $r \in \Pi - \Pi'$, $P(r) \not\subseteq H(\Pi')$ or $N(r) \cap H(\Pi') \neq \emptyset$.

Intuitively, any rule which is not compatible with the set GR(X,Π) or whose positive body can not be derived from GR(X,Π) would be inapplicable.

From the proof of Theorem 3 (See Appendix) we get its equivalent version, that is:

Theorem 3*. *A consistent set of literals S is an answer set of a program $\Pi = \Pi^* \cup C_\Pi$ if and only if there is a subset Π' of Π^* such that:*
(i) $\Pi' = \{r \in \Pi^* \mid P(r) \subseteq S$ and $N(r) \cap S = \emptyset\}$ and $S = H(\Pi')$;
(ii) $\Pi' \cup C_\Pi$ is compatible;
(iii) $\Lambda(\Pi') = \Pi'$;
(iv) For $r \in \Pi - \Pi'$, $P(r) \not\subseteq S$ or $N(r) \cap S \neq \emptyset$.

Now we conclude that an answer set of a program is a minimal set satisfying Theorem 3.

Theorem 4. *(Minimality of answer sets) If X and Y are answer sets of a program $\Pi = \Pi^* \cup C_\Pi$ and $X \subseteq Y$, then $X = Y$.*

Example 6 Any nonempty set of constraints has no answer sets.

Corollary 2. *If program $\Pi = \Pi^* \cup C_\Pi$ is compatible then it has just one answer set $H(\Lambda(\Pi^*))$.*

Actually, results from Theorem 3 and its corollary give the characterization of consistent answer set for a program in terms of compatibility.

4. Splitting a Program

Although stemming from Lifschitz's definition, our description of splitting is slightly different from the former. In our opinion, for any program Π, a collection

$$\Pi^0 = \begin{cases} \{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\} & \text{if } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \text{ is consistent} \\ & \quad \text{and } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \cap P(c) \neq P(c), \\ & \quad \text{or } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \cap N(c) \neq \emptyset \text{ for each } c \in C_\Pi \\ \Pi_{Lit} & \text{if } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \text{ is inconsistent and } P(C_\Pi) = \emptyset \\ \bot & \text{if } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \text{ is inconsistent and } P(C_\Pi) \neq \emptyset \\ & \quad \text{or if } H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \text{ is consistent and} \\ & \exists c \in C_\Pi \; H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \cap P(c) = P(c) \text{ and} \\ & \quad H(\{r \in \Pi' \mid P(r) \subseteq \cup_{0 \le n} T^n_{\Pi'}(\emptyset)\}) \cap N(c) = \emptyset \end{cases} \quad (1)$$

$$\Pi^{n+1} = \begin{cases} \{r \in \Pi - C_\Pi \mid P(r) \subseteq H(\Pi^n)\} & \text{if } H(\{r \in \Pi - C_\Pi \mid P(r) \subseteq H(\Pi^n)\}) \text{ is consistent} \\ & \quad \text{and } H(\{r \in \Pi - C_\Pi \mid P(r) \subseteq H(\Pi^n)\}) \cap P(c) \neq P(c) \\ & \quad \text{or } H(\{r \in \Pi - C_\Pi \mid P(r) \subseteq H(\Pi^n)\}) \cap N(c) \neq \emptyset \text{ for each } c \in C_\Pi) \\ \bot & \text{if } \Pi^n = \Pi_{Lit} \text{ or } \Pi^n = \bot \\ & \quad \text{or } H(\{r \in \Pi - C_\Pi \mid P(r) \subseteq H(\Pi^n)\}) \text{ is inconsistent} \\ & \quad \text{or } H(\{r \in \Pi - C_\Pi \mid P(c) \subseteq H(\Pi^n)\} \text{ is consistent and} \\ & \exists c \in C_\Pi \; H(\{r \in \Pi - C_\Pi \mid P(c) \subseteq H(\Pi^n)\}) \cap P(c) = P(c) \text{ and} \\ & \quad H(\{r \in \Pi - C_\Pi \mid P(c) \subseteq H(\Pi^n)\}) \cap N(c) = \emptyset) \end{cases} \quad (2)$$

of subprograms of Π, $\{\Pi_i\}(i\le n)$, is a finer splitting of Π if

- for each $\Pi_i(i\le n)$ and any $r'\in\Pi_i$, there is a $r\in\Pi$ such that $H(r)=H(r')$, $P(r')\subseteq P(r)$ and $N(r')\subseteq N(r)$
- $H(\Pi_i)\cap H(\Pi_j)=\varnothing$ for any $i\neq j$, and
- Π has a consistent answer set S if and only if each $\Pi_i(i\le n)$ has a consistent answer set S_i such that $S=\cup_i S_i$.

Based on above comprehension of splitting, a program is split in following steps:

Step 1:

$\Pi_0=\{r\in\Pi'|P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\}=\Pi^0$, where $\Pi'=\{r\in\Pi|N(r)=\varnothing\}$, If $H(\Pi_0)$ is consistent, and $H(\{r\in\Pi'|P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\})\cap P(c)\neq P(c)$ or $H(\{r\in\Pi'\mid P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\})\cap N(c)\neq\varnothing$ for each $c\in C_\Pi$.

Step 2:

For $n\ge 0$, $\Pi_{n+1}=\{r'|r\in\Pi-C_\Pi-\cup_{i\le n}\Pi_i$, and $H(r')=H(r)$, $P(r')=P(r)-H(\cup_{i\le n}\Pi_i)$, $N(r')=N(r)-H(\cup_{i\le n}\Pi_i)\}$, if $H(\Pi_n)$ is consistent, and $H(\{r\in\Pi-C_\Pi|P(r)\subseteq H(\cup_{i\le n}\Pi_i)$, $N(r)\cap H(\cup_{i\le n}\Pi_i)=\varnothing\})\cap P(c)\neq P(c)$ or $H(\{r\in\Pi-C_\Pi|P(r)\subseteq H(\cup_{i\le n}\Pi_i)$, $N(r)\cap H(\cup_{i\le n}\Pi_i)=\varnothing\})\cap N(c)\neq\varnothing$ for each $c\in C_\Pi$.

Basically, r' is obtained from r by eliminating $H(\cup_{i\le n}\Pi_i)$ from body of r, where $P(r)\subseteq H(\cup_{i\le n}\Pi_i)$, $N(r)\cap H(\cup_{i\le n}\Pi_i)=\varnothing$.

Algorithm implementing the splitting is given as follows:

```
FUNCTION BASIC(Π)
{Πb:=∅;
 FOR each r∈Π DO
   IF N(r)=∅ THEN Πb:=Πb∪{r};
 RETURN (Πb);
}
FUNCTION T_Π(Πb)
{Cn:=∅;
 DO
  {result:=Cn;
   FOR each r∈Πb DO
     IF P(r)⊆Cn THEN
       {Cn:=Cn∪H(r);
        Πb:=Πb−{r};}
   IF Cn is inconsistent THEN RETURN (Lit);
  } UNTIL (result=Cn)
 RETURN (result);
}
FUNCTION SPLITTING(Π)
{i:=0; j:=0;
 token:=false;
 find:=true;
 sat:=false;
 Π′:=BASIC(Π);
 X:=T_Π(Π′);
 IF X=Lit THEN
  {FOR each c∈C_Π DO
     IF N(c)=∅ THEN
       {token:=true;
        EXIT FOR;}
   IF token:=true
     THEN RETURN(no splitting)
     ELSE RETURN(unique inconsistent answer set);
  }
 Π:=Π−Π′−C_Π;
 Π_0:=∅;
 FOR each r∈Π′ DO
   IF P(r)⊆X THEN Π_0:=Π_0∪{r};
 FOR each c∈C_Π DO
   IF P(c)⊆H(Π_0) AND N(c)∩H(Π_0)=∅
     THEN {sat:=true;
           EXIT FOR;}
 IF sat=true THEN RETURN(no splitting);
 Π_u:=Π_0;
 DO
  {Π_{i+1}:=∅;
   Π_u′=Π_u;
   FOR each r∈Π DO
    IF P(r)⊆H(Π_u) AND N(r)∩H(Π_u)=∅
      THEN
      {Π_{i+1}:=Π_{i+1}∪{r};
       Π_u′=Π_u′∪Π_{i+1};
       FOR each c∈C_Π DO
         IF P(c)⊆H(Π_u′) AND N(c)∩H(Π_u′)=∅
           THEN
             {Π_{i+1}:=Π_{i+1}−{r};
              Π_u′=Π_u′−{r}; }
       IF H(Π_{i+1})∪N(Π_{i+1}) is inconsistent
         THEN OUTPUT(no answer set for Π_{i+1});
      }
   IF Π_{i+1}≠∅ THEN
        {Π:=Π−Π_u−Π_{i+1};
         FOR each r∈Π_{i+1}
```

```
            DO
              {P(r):=P(r)−H(Π_u);
               N(r):=N(r)−H(Π_u);}
            Π_u:= Π_u∪Π_{i+1};
            i:=i+1;
           }
      ELSE  find:=false;
 }UNTIL(find=false)
 FOR j=0 to i DO  OUTPUT (Π_j);
 RETURN ( );
}
```

Function BASIC generates a basic program Πb whose rules are picked out from Π, function T_Π implements T_Π^n on Πb, and function SPLITTING returns the sequence of $\{\Pi_i\}(i\leq n)$ from Π. Generally speaking, the complexity of directly computing answer sets for Π is $O(2^{|\Pi|})$; after splitting Π into $\{\Pi_i\}(i\leq n)$, the total complexity of computing answer sets for $\{\Pi_i\}$ is $O(\Sigma_{i\leq n}2^{|\Pi i|})$, which is much less than the former.

Now we present a basic theorem shown in Appendix, which guarantees the correctness of the above splitting notion and algorithm.

Theorem 5. *Program Π has an answer set S if and only if there is a finer splitting $\{\Pi_i \mid i\geq 0\}$ of Π such that $S=\cup_i S_i$, where S_i is an answer set of Π_i.*

5. An Example in E-retailing

With the development of Internet, more and more producers or companies retail their products or services on the web. In these applications, of course as well as other forms of e-service, providing personalized service to users according to their demands is very helpful for building a one-to-one relationship between the customer and the service provider, consequently enhance the user satisfactions.[1] In this section, we will give a simplified example to show the application of ASP on service personalization in an e-retailing system, and our splitting method and its advantage are justified by the example.

Through a web-based retailing system, a PC retailer can sell products or services to users on the web. Computer is a kind of typical configurable product, usually customers have some special requirements or preference on some components of the machines they are to buy, or want to know the prices corresponding to various configurations. To meet these requirements, a reasoning mechanism should be included in the e-retailing system. Some knowledge is incomplete when building such a mechanism, e.g., the retailer has no idea about which type of CPU the customer prefers to, then various possibilities should be considered.

Assume on the selling webpage of a PC e-retailing system there is an item list with option boxes for each class of components of a PC, such as CPU, mainboard, and so on. Each item in such a list refers to one type of a component. Predicate component(X) states that X is a PC component, e.g., CPU. Usually there are various types for a component, binary predicate hastype(Y,X) means that component X has a type of Y.

Normally, a customer prefer one type of component to others of the same class, so he will choose the corresponding option box along with the type he preferred. Once a type Y is labeled, predicate choose(Y) hold, this means Y will be chosen as a part of the anticipated PC. However, it is possible, although infrequently, the customer makes no preference for a class of components, then every type of this component has the equal chance to be chosen. Furthermore, for any component, it is not allowed more than one type is preferred and hence to be chosen. These notions are captured by the following schematic rules:

$$1\{\text{choose}(Y)\colon \text{hastype}(Y,X)\}1 \leftarrow \text{component}(X) \qquad (3)$$

$$\leftarrow \text{component}(X), \text{hastype}(Y_1,X), \text{hastype}(Y_2,X), \text{choose}(Y_1), \text{choose}(Y_2), Y_1\neq Y_2 \qquad (4)$$

where variable X ranges over all components, Y, Y_1 and Y_2 denote types of component X.

Also being a “choice rule”, rule (3) says: if X is a component, then from all types Y of X such that hastype(Y,X) holds, choose at least one and at most one Y to make choose(Y) holds.

Rule (4) indicates that any two different types of the same component can not be chosen at the same time.

Technical parameters should be taken into consideration when configuring a PC, if a type Y_1 of component X_1 is incompatible with type Y_2 of component X_2, then predicate incompatible(Y_1, Y_2) holds, Y_1 and Y_2 should not be chosen simultaneously. This can be represented as a constraint:

$$\leftarrow \text{component}(X_1), \text{component}(X_2), \text{hastype}(Y_1,X_1), \text{hastype}(Y_2,X_2), \text{choose}(Y_1), \text{choose}(Y_2), X_1\neq X_2 , \text{incompatible}(Y_1,Y_2). \qquad (5)$$

Rule (3)~(5), together with other facts, e.g., component(*X*) for all possible components *X*, choose(*Y*) for some types *Y*, form a logic program Π, answer set(s) of Π give the possible configuration(s) of the anticipated machine.

To simplify the presentation, we assume that just three classes of components to be considered, they are mainboard, CPU and memory. Types of each component are shown in Tab.1.

Table 1. Types of PC components

Component class	Types
mainboard	mb_A
	mb_B
CPU	cpu_A
	cpu_B
memory	mem_A
	mem_B

According to Tab.1, we can list the first group G_1 of rules(facts) of Π:

component(mainboard).
component(cpu).
component(memory).
hastype(mb_A, mainboard).
hastype(mb_A, mainboard).
hastype(mb_B, mainboard).
hastype(cpu_A, cpu).
hastype(cpu_A, cpu).
hastype(mem_A, memory).
hastype(mem_B, memory).

Actually, rules in G_1 are basic facts of the system, although will appear in answer set of Π, not the results we concerned.

Now rule (3)~(4) can be grounded to the second group G_2 of rules, they are:

choose(mb_A)←component(mainboard),
hastype(mb_A, mainboard),
hastype(mb_B, mainboard),
not choose(mb_B).

choose(mb_B)←component(mainboard),
hastype(mb_A, mainboard),
hastype(mb_B, mainboard),
not choose(mb_A).

choose(cpu_A)←component(cpu),
hastype(cpu_A, cpu),
hastype(cpu_B, cpu),
not choose(cpu_B).

choose(cpu_B)←component(cpu),
hastype(cpu_A, cpu),
hastype(cpu_B, cpu),
not choose(cpu_A).

choose(mem_A)←component(memory),
hastype(mem_A, memory),
hastype(mem_B, memory),
not choose(mem_B).

choose(mem_B)←component(memory),
hastype(mem_A, memory),
hastype(mem_B, memory),
not choose(mem_A).

←component(mainboard),
hastype(mb_A, mainboard),
hastype(mb_B, mainboard),
choose(mb_A), choose(mb_B).

←component(cpu),
hastype(cpu_A, cpu),
hastype(cpu_B, cpu),
choose(cpu_A), choose(cpu_B).

←component(memory),
hastype(mem_A, memory),
hastype(mem_B, memory),
choose(mem_A), choose(mem_B).

The former six rules of G_2 are grounded forms of rule (3), and the latter three ones are grounded from rule (4).

If a customer labels cpu_A and mem_B as preference, and assume that mb_B and cpu_A are incompatible, then we get the third group G_3 of rules:

choose(cpu_A).
choose(mem_B).
incompatible(cpu_A,mb_B).

←component(cpu), component(memory),
hastype(cpu_A, cpu), hastype(mem_B, memory),
choose(cpu_A), choose(mb_B),
incompatible(cpu_A,mb_B).

Program $\Pi=G_1\cup G_2\cup G_3$ describes basic facts about PC configuration, customer's requirements and reasoning rules based on these facts. Obviously the only answer set of Π is $G_1\cup${choose(cpu_A), choose (mem_B), choose(mb_A), incompatible(cpu_A,mb_B)}.

Next we will show the splitting computation of Π. First function BASIC finds all basic rules of Π:

$\Pi b= G_1\cup${choose(cpu_A). choose(mem_B).
incompatible(cpu_A, mb_B).}

Then function T_Π gives consequence of Πb easily:

$Cn(\Pi b) = \cup_{0\le n} T_{\Pi b}{}^{n}\varnothing$
$= G_1 \cup \{choose(cpu\_A), choose(mem\_B), incompatible(cpu\_A, mb\_B)\}$

Finally, function SPLITTING returns the following $\Pi_i (i \le n)$:

$\Pi_0 = G_1 \cup \{choose(cpu\_A).\ choose(mem\_B).\ incompatible(cpu\_A, mb\_B).\}$

$\Pi_1 = \{choose(mb\_A) \leftarrow not\ choose(mb\_B).$
$choose(cpu\_A) \leftarrow not\ choose(cpu\_B).$
$choose(mem\_B) \leftarrow not\ choose(mem\_A).\}$

$\Pi_2 = \varnothing$

In fact, while computing Π_1, there are two choices of rules:

$\Pi_{1\text{-}1} = \{choose(mb\_A) \leftarrow not\ choose(mb\_B).$
$choose(cpu\_A) \leftarrow not\ choose(cpu\_B).$
$choose(mem\_B) \leftarrow not\ choose(mem\_A).\}$

$\Pi_{1\text{-}2} = \{choose(mb\_B) \leftarrow not\ choose(mb\_A).$
$choose(cpu\_A) \leftarrow not\ choose(cpu\_B).$
$choose(mem\_B) \leftarrow not\ choose(mem\_A).\}$

However, $\Pi_{1\text{-}2}$ is eliminated because of the constraint:

$\leftarrow component(cpu), component(memory), hastype(cpu\_A, cpu), hastype(mem\_B, memory), choose(cpu\_A), choose(mb\_B), incompatible(cpu\_A, mb\_B).$

Answer sets for Π_0, Π_1 are $G_1 \cup \{choose(cpu\_A), choose(mem\_B), incompatible(cpu\_A, mb\_B)\}$ and $\{choose(cpu\_A), choose(mem\_B), choose(mb\_A)\}$ respectively, the union of them exactly equals to the answer set of Π.

Through splitting, numbers of rules in Π_0, Π_1 are much less than those of Π, their answer sets are very easy to compute.

Tiihonen *et al* have shown that basic logic program with weight constraint rules is quite suitable for naturally describing product configuration problem.[7,21] There product's configuration information are represented as weight constraint rules, together with customer's determinate requirements for some components, the unique answer set of these rules corresponds to the expected configuration. However, possibility of incomplete information was not discussed. Example presented in this section is very simple. Whereas incomplete knowledge possibly emerged in such applications is considered. Basically, any subfield of e-service related to incomplete knowledge representing and reasoning is the congruent application domain for ASP.

6. Conclusion

With the notion of compatibility and Λ-operator, we described the characterization of answer set for a logic program with constraints. From the characterization we can check if a program has answer set, and if so, a stepwise method presented in this paper can be used to split a program into subprograms, such that the union of answer sets of subprograms is the answer set of original program. To a certain extent, this simplifies the task of solving answer set for logic program. In any area involving knowledge representation and reasoning, it is a desirable result.

An example about personalized product configuration in e-retailing is given to show that ASP provides a natural and compact description for personalizing product configuration. Our splitting method and its advantage are justified by this example. It is just one of many applications of ASP. In the context of software diagnosis[22] or alias analysis[23], ASP is applicable for addressing incomplete knowledge or frame problem in the two areas. Next we plan to apply our method on these issues.

Acknowledgements

We want to thank the anonymous reviewers for their helpful comments on an earlier version of this paper. This work is supported by the Natural Science Foundation of China under Grant No. 60963009 and No. 61003203, Natural Science Foundation of Guizhou Province No.[2009]2123, Natural Science and Technology Foundation of Guizhou Province No. SY [2010]3070.

References

1. J. Lu, D. Ruan, and G.Q.Zhang. E-Service Intelligence: An Introduction, in *Studies in Computational Intelligence* (Springer, Berlin, 2007), pp.1-33.
2. M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, in *Proc. of International Logic Programming Conference and Symposium*, eds. R. Kowalski, and K. Bowen(1988), pp.1070-1080.
3. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson and M. Barry, An A-Prolog decision support system for the

Space Shuttle, in *Proc. of PADL2001*, (Springer, Berlin, 2001), pp.169-183.
4. A. Paschke, and M. Bichler, Knowledge representation concepts for automated SLA management, *Decision Support Systems*, 46(1)(2008) 187-205.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity, *ACM Trans. on Comp. Logic*, 5(2) (2004) 206-263.
6. P. Tu, T. Son, M. Gelfond, and A. R. Morales, Approximation of Action Theories and Its Application to Conformant Planning, *Artificial Intelligence*, 175(1)(2011) 79-119.
7. J. Tiihonen, T. Soininen, I. Niemela, and R. Sulonen, A practical tool for mass-customising configurable products, in *Proc. of the 14th International Conference on Engineering Design*, (2003), pp.1290-1299.
8. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, Complexity and expressive power of logic programming, in *Proc. of 12th IEEE Conference on Computational Complexity*, (1997), pp.82-101.
9. V. Lifschitz, and H. Turner, Splitting a logic program, in *Proc. of Int. Conf. on Logic Programming*, (1994)23-38(1994).
10. V. Lifschitz, Foundation of Logic Programming, in *Principles of Knowledge Representation*, (CSLI Publications, 1996), pp. 69-128.
11. H. Turner, Splitting a Default Theory, in *Proc. of AAAI 1996*, (AAAI Press, 1996), pp.645-651.
12. R. Watson, A Splitting Set Theorem for Epistemic Specifications, in *CoRR cs/0003038v1*, (2000).
13. M. Balduccini, Splitting a CR-Prolog Program, in *Logic Programming and Nonmonotonic Reasoning*, (LNCS5753, 2009), pp. 17-29.
14. M. Zhang, A characterization of Extensions of General Default Theories, in *Proc. of AI92*, (1992), pp.134-139.
15. M. Zhang, A New Research into Default Logic, *Inf. Comput.*, 129(2)(1996) 73-85.
16. M. Zhang, Y. Zhang, and F. Lin, A characterization of answer sets for logic programs, *Sci. in China, Series F: Information Science*, 50(1)(2007) 46-62.
17. M. Wu, D. Zhang and M. Zhang, Language Splitting and Relevance-Based Belief Change in Horn Logic, in *Proc. of 25th AAAI Conf. on AI*, eds. W. Burgard and D. Roth(AAAI Press,2011), pp.268-273.
18. M. Wu and M. Zhang, Algorithms and Application in Decision Making for the Finest Splitting of a Set of Formulae, *Knowledge-Based systems*, 23(1)(2010) 70-76.
19. V. Lifschitz, What is Answer Set Programming, in *Proc. of the 23rd AAAI Conf. on AI*, eds. D. Fox and C. P. Gomes(AAAI, 2008), pp. 1594-1597.
20. P. Ferraris and V. Lifschitz, Mathematical Foundations of Answer Set Programming, in *We Will Show Them! Essays in Honour of Dov Gabbay*(King's College Publications, 2005), pp. 615-664.
21. T. Soininen, I. Niemela, J. Tiihonen, and R. Sulonen, Representing Configuaration Knowledge With Weight Constraint Rules, in *Proc. of the AAAI spring 2001 symposium on Answer Set Programming*, (2001), pp. 95-201.
22. W. Mayer, and M. Stumptner, Model-based debugging-state of the art and future challenges, *Electronic Notes in TCS*, 174(4)(2007) 61-82.
23. B. Meyer, Towards a theory and calculus of aliasing, *J. of Object Technology*, 9(2)(2010) 37-74.

Appendix A.

Some proofs of this paper are included in this section.

Proposition 1. *If program $\Pi=\Pi^*\cup C_\Pi$ is compatible then any $\Pi'\subseteq\Pi^*$ is also compatible.*

Proof.

It is obvious from Definition 8. □

Proposition 2. *If $\Lambda(\Pi)\neq\Pi_{Lit}$ and $\Lambda(\Pi)\neq\bot$ then $\Lambda(\Pi)\subseteq\Pi^*$ and $H(\Lambda(\Pi))$ satisfies C_Π.*

Proof.

It is easily proven by induction on n that $\cup_{0\le n}\Pi^n\subseteq\Pi^*$.

Base

n=0. Because $\Lambda(\Pi)\neq\Pi_{Lit}$ and $\Lambda(\Pi)\neq\bot$, then by Definition 2, we know that for set Π' of basic rules, $\Pi'\subseteq\Pi^*$ and $\Pi^0=\{r\in\Pi'|\ P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\}$, with the promise that $H(\{r\in\Pi'|\ P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\})$ is consistent, and $H(\{r\in\Pi'|\ P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\})\cap P(c)\neq P(c)$, or $H(\{r\in\Pi'|P(r)\subseteq\cup_{0\le n}T_\Pi{}^n\varnothing\})\cap N(c)\neq\varnothing$ for each $c\in C_\Pi$. So $\Pi^0\subseteq\Pi'\subseteq\Pi^*$, and $H(\Pi^0)$ satisfies C_Π.

Step

Suppose that for any natural number i, there is $\cup_{0\le i}\Pi^i\subseteq\Pi^*$, and $H(\cup_{0\le i}\Pi^i)$ satisfies C_Π. From $\Lambda(\Pi)\neq\Pi_{Lit}$ and $\Lambda(\Pi)\neq\bot$ we can conclude that $\Pi^{i+1}=\{r\in\Pi-C_\Pi|\ P(r)\subseteq H(\Pi^i)\}$, with the promise that $H(\{r\in\Pi-C_\Pi|P(r)\subseteq H(\Pi^i)\})$ is consistent, and $H(\{r\in\Pi-C_\Pi|P(r)\subseteq H(\Pi^i)\})\cap P(c)\neq P(c)$, or $H(\{r\in\Pi-C_\Pi|P(r)\subseteq H(\Pi^i)\})\cap N(c)\neq\varnothing$ for each $c\in C_\Pi$. Each r in Π^{i+1} is from Π^*, *i.e.*, $\Pi^{i+1}\subseteq\Pi^*$. By induction assumption, $(\cup_{0\le i}\Pi^i)\cup\Pi^{i+1}\subseteq\Pi^*$, that is $\cup_{0\le i+1}\Pi^{i+1}\subseteq\Pi^*$, and $H(\cup_{0\le i+1}\Pi^{i+1})$ satisfies C_Π. □

Proposition 3. *Λ is monotonic, i.e., if $\Pi_1\subseteq\Pi_2$ and $\Lambda(\Pi_2)\neq\bot$, then $\Lambda(\Pi_1)\subseteq\Lambda(\Pi_2)$.*

Proof.

It is immediate from Definition 2. □

Lemma 1. *If $\Lambda(\Pi)\neq\Pi_{Lit}$ and $\Lambda(\Pi)\neq\perp$ then $\Lambda(\Lambda(\Pi))=\Lambda(\Pi)$.*

Proof.

By proposition 2 it is sufficient to show that $\Lambda(\Lambda(\Pi))\subseteq\Lambda(\Pi)$. Then what we need to do is to prove by induction on n that $(\Lambda(\Pi))^n\subseteq(\Lambda(\Lambda(\Pi)))^n$.

Base

It is obvious that $(\Lambda(\Pi))^0\subseteq(\Lambda(\Lambda(\Pi)))^0$.

Step

Assume that $(\Lambda(\Pi))^i\subseteq(\Lambda(\Lambda(\Pi)))^i$. Then for any $r\in(\Lambda(\Pi))^{i+1}$, $P(r)\subseteq H((\Lambda(\Pi))^i)$. By the induction assumption, $P(r)\subseteq H((\Lambda(\Lambda(\Pi)))^i)$, which implies that $r\in(\Lambda(\Lambda(\Pi)))^{i+1}$. So $(\Lambda(\Pi))^{i+1}\subseteq(\Lambda(\Lambda(\Pi)))^{i+1}$.

All of the above shows that $\Lambda(\Lambda(\Pi))=\Lambda(\Pi)$. □

Lemma 2. *If $\Lambda(\Pi)\neq\Pi_{Lit}$ and $\Lambda(\Pi)\neq\perp$ then for any $r\in\Pi^*$, $P(r)\subseteq H(\Lambda(\Pi))$ if and only if $r\in\Lambda(\Pi)$.*

Proof.

"If " is clear.

For the "Only If", we easily show it by the definition of Λ. □

Theorem 2. *Program $\Pi=\Pi^*\cup C_\Pi$ has an inconsistent answer set if and only if $Cn(\{r\in\Pi^*|N(r)=\emptyset\})=Lit$ and $N(c)\neq\emptyset$ for any $c\in C_\Pi$.*

Proof .

It is immediate by definition of answer set. In fact, *Lit* is an answer set of Π if and only if $\Pi_{Lit}=\{r\in\Pi^*|N(r)=\emptyset\}$, $Cn(\{r\in\Pi^*|N(r)=\emptyset\})=Lit$ and $N(c)\cap Lit\neq\emptyset$ for each $c\in C_\Pi$. □

Corollary 1. *If Lit is an answer set of program Π, then it is the unique answer set of Π.*

Proof .

Suppose Π has a consistent answer set S, then $\{r\in\Pi^*|N(r)=\emptyset\}\subseteq\Pi^S$, so $Cn(\{r\in\Pi^*|N(r)=\emptyset\})\subseteq S$. On the other hand, we have $Cn(\{r\in\Pi^*|N(r)=\emptyset\})=Lit$ by Theorem 2. So, $S\subseteq Lit$. Furthermore, $S=Lit$ which is in contradiction to consistency of S. □

Theorem 3. *Program $\Pi=\Pi^*\cup C_\Pi$ has a consistent answer set if and only if there is a subset Π' of Π^* such that:*

(i) $\Pi'\cup C_\Pi$ is compatible;

(ii) $\Lambda(\Pi')=\Pi'$;

(iii) For $r\in\Pi-\Pi'$, $P(r)\not\subseteq H(\Pi')$ or $N(r)\cap H(\Pi')\neq\emptyset$.

Proof.

Define $r^*=H(r)\leftarrow P(r)$, where $r=H(r)\leftarrow P(r)$, not $(N(r))$ and $\Pi^+=(r^*|\ r\in\Pi^*)$. Denote $\Pi(r^*)=\{r\in\Pi^*|H(r)=H(r^*)$ and $P(r)=P(r^*)\}$ and $\Pi^{i^*}=\cup_{r\in\Pi}\Pi(r^*)$ (i^* stands for inverse of *).

It is easy to see that $\Lambda(\Pi)=\Lambda(\Pi^{i^*})$, $(\Pi^*)^S=\{r^*|\ N(r^*)\cap S=\emptyset\}$. By induction it is easy to show that $\Lambda((\Pi^*)^S)=(\Lambda(\Pi^+))^S$.

"If"

Suppose Π' satisfies conditions (i), (ii) and (iii), which implies that $\Lambda(\Pi')\neq\perp$. By the definition of the operator Λ and condition (ii), it is easy to see that $\Pi'\subseteq\Pi^+$, and Π' satisfies C_Π. Let $S=H(\Pi')=H(\Lambda(\Pi'))$. So, it is sufficient to show that S is an answer set of Π. At first, we have $(\Pi^+)^S=\{r^*|\ r\in\Pi^+$ and $N(r)\cap S=\emptyset\}$. Since the stable model of $(\Pi^+)^S$ is $H(\Lambda((\Pi^+)^S)$, we show that $\Lambda((\Pi^+)^S)^{i^*})=\Lambda(\Pi')=\Pi'$, which implies that S is an answer set of Π^*. At first, we have $S=H(\Pi')\subseteq H(\Lambda(((\Pi^*)^S)^{i^*}))$. Next, we can inductively show that $(((\Pi^*)^S)^{i^*})^n\subseteq\Pi'$, which implies that $H(\Lambda(((\Pi^*)^S)^{i^*}))\subseteq\Pi'$. So, we get $\Lambda(((\Pi^*)^S)^{i^*})=\Pi'$. Furthermore, $(\Lambda((\Pi^*)^S)^{i^*})=\Lambda(\Pi')$. Hence, $S=H(\Lambda(\Pi'))=H((\Lambda((\Pi^*)^S)^{i^*})=H(\Lambda((\Pi^*)^S))$.

"Only If"

Suppose that S is an answer set of Π^*. Then S is the stable model of $(\Pi^*)^S$ and $S=H(\Lambda((\Pi^*)^S)$. Let $\Pi'=\{r\in\Pi|r^*=H(r)\leftarrow P(r)\in\Lambda(\Pi^S)\}$, then $\Pi'=(\Lambda((\Pi^*)^S))^{i^*}$ and $\prod'$ is compatible. Hence Π' Satisfies (i). From Lemma 1, we have $\Lambda(\Pi')=\Pi'$, *i.e.*, (ii) is satisfies by Π'. For any $r\in\Pi-\Pi'$, if $P(r)\subseteq H(\Pi')$ and $N(r)\cap H(\Pi')=\emptyset$, then $r^*\in(\Pi^*)^S$ and $r\in\Lambda(((\Pi^*)^S)^{i^*})=\Pi'$, which contradicts the assumption that $r\in\Pi^*-\Pi'$. So, condition (iii) holds for Π'. □

Theorem 4 *(Minimality of answer sets) If X and Y are answer sets of a program $\Pi=\Pi^*\cup C_\Pi$ and $X\subseteq Y$, then $X=Y$.*

Proof.

If one of X and Y is *Lit*, then X=Y by Corollary 1. Suppose neither X nor Y is *Lit* and $X\subset Y$ (*i.e.*, $X\subseteq Y$ and $X\neq Y$). Hence there is $r\in\Pi''-\Pi'$. Let

$\Pi'=\{r\in\Pi\mid P(r)\subseteq X$ and $N(r)\cap X=\emptyset\}$, and

$\Pi''=\{r\in\Pi\mid P(r)\subseteq Y$ and $N(r)\cap Y=\emptyset\}$.

By Theorem 3*, we have $X=H(\Pi')$, $Y=H(\Pi'')$ and both Π' and Π'' are compatible. So, $N(r)\cap X=\emptyset$ resp. $N(r)\cap Y=\emptyset$ by compatibility of Π' resp. Π''. On the other hand, since $r\in\Pi''-\Pi'\subseteq\Pi-\Pi'$ and Π' satisfies conditions in Theorem 3*, then $N(r)\cap X\neq\emptyset$ by (iv) of Theorem 3*.

This contradicts to the previously derived conclusion $N(r) \cap X = \varnothing$. Hence X=Y. □

Corollary 2. *If program $\Pi = \Pi^* \cup C_\Pi$ is compatible then it has just one answer set $H(\Lambda(\Pi^*))$.*

Proof.

It is obvious by Definition 1 and Theorem 3. □

Theorem 5. *Program Π has an answer set S if and only if there is a finer splitting $\{\Pi_i | i \geq 0\}$ of Π such that $S = \cup_i S_i$, where S_i is an answer set of Π_i.*

Proof.

It is easy to be derived by Theorem 2 and Theorem 3. □