A three-dimensional visualization of communications in distributed program environments

Mariko Sasakura
Okayama University

Susumu Yamasaki
Okayama University

A Three-Dimensional Visualization of Communications in Distributed Program Environments

Mariko Sasakura and Susumu Yamasaki

Department of Intelligence Computing and Systems
Graduate School of Natural Science and Technology, Okayama University
Tsushima-Naka 3-1-1, Okayama, 700-8530, Japan
sasakura@momo.it.okayama-u.ac.jp

Abstract

In order to understand the behaviour of distributed program environments, this paper describes a three-dimensional visualization of communications. Most of previous visualizations presented communications by means of two dimensions of time and process, however, three dimensions are required to represent a message intermediary with visualization functions. We make use of a semantic configuration of messages, which is constructed by the quantities for three parametric factors: (i) *time sequence,* (ii) *identifications of processors, and* (iii) *message exchanges.*

1. Introduction

The behaviour of distributed programs is generally complicated. Although the behaviour of each process of a distributed program is reduced to a sequential program, the communications between processes make the whole behaviour complicated. The visualization of communications in distributed programs may be one of the major approaches to techniques of understanding the behaviour and of debugging. Most visualizations present the communication by two dimensions of time and process, where the visualizations are devoted to the display of the amount of messages and the availability of processors.

In distributed program environments, there are complexities in sending/receiving messages. To check if messages are correctly sent/received, we consider a semantic configuration of messages. As a pattern of semantic configurations, we deal with the visualization of communications in distributed logic program environments.

To make stress on the communication environment, we take a three-dimensional approach to the visualization:

1. The first dimension is to describe sequences of configurations, as time goes and processes are implemented.
2. The second dimension is concerned with the (spatial or logical) extension of distributed programs
3. The third dimension is required for the communication histories to form a configuration of communications in distributed programs for some duration

As a distributed program environment, a distributed logic program is examined for the following reason.

As discussed in Shepherdson [11], negation as failure rule is well established in relation to 3-valued logic models: If a proposition A cannot be proved by a theory P ($P \not\vdash A$), then the negated predicate $not\ A$ may be inferable. It is applicable to the deductive database, to infer $not\ A$ by applying finite searches of the predicate A and failing in its detection. The acquisition of the negated predicate $not\ A$ is generally applicable to abduction (as in Kakas et al. [4]), diagnosis, causal theory and so on.

On the assumption of distributed environments of programs and/or databases, negation as failure is revised for the environments by the idea that the negation as failure is performed at each site of the program or the database. As well, the communications for the negation as failure applications are made clear to be visualized.

Such an idea motivates the formulation of a distributed logic program with negation as failure, which is extended from the distributed program without negation [9], and the study on the visualization of communications for negation as failure in distributed program environments. A distributed logic program is a network of logic programs, where (1) the reasoning for each logic program is defined, and (2) the negation as failure, evoked by each program, is formulated in the whole network.

Under the above backgrounds, we present a three-dimensional visualization of communications in a distributed logic program, which is a network of logic programs. The communication is caused by negation as failure through the network, where each logic program makes reasonings by using the negation as failure through the network. It consists of the displays for: (1) a sequence of configurations, (2) a network of logic programs, and (3) a configuration of negation as failure through the network for some duration.

2. A distributed logic program

We deal with a network of logic programs which contain negation as failure, where negation as failure through the network is formulated and the communications to implement it are visualized.

A distributed general logic program (DGLP, for short) is a tuple

$$< P_1, \ldots, P_n > (n \geq 1),$$

where P_i is the general logic program.

A general logic program is a set of clauses of the form $A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ $(n \geq m \geq 0)$, where A_0, A_1, …, A_m are atoms (positive literals) and $\mathit{not}\ A_{m+1}$, …, $\mathit{not}\ A_n$ are negations of atoms (negative literals). A_0 is the head of the clause and $A_1, \ldots, A_m$, $\mathit{not}\ A_{m+1}$, …, $\mathit{not}\ A_n$ is its body. A literal is a positive literal or a negative literal. The clause is also expressed as $A_0 \leftarrow L_1, \ldots, L_n$, where L_1, …, L_n are literals. The clause containing no "*not*" is a definite clause. The program containing only definite clauses is a definite program.

The goal is an expression of the form $\leftarrow L_1, \ldots, L_n$, where L_1, …, L_n, where L_1, …, L_n are literals. The empty clause containing no head nor body is denoted by $\Box$.

The reasonings by SLD resolution and negation as failure for the goal are briefly given below. For the basic treatments, see Lloyd [6].

(1) A goal $\leftarrow A_1\theta, \ldots, A_{i-1}\theta, L_1\theta, \ldots, L_k\theta, A_{i+1}\theta, \ldots, A_m\theta, \mathit{not}\ A_{m+1}\theta, \ldots, \mathit{not}\ A_n\theta$ is derived from a goal $\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ and a (program) clause $A \leftarrow L_1, \ldots, L_k$, where θ is a most general unifier of the atoms A_i $(1 \leq i \leq m)$ and A. If a goal reaches $\Box$ by SLD resolution and negation as failure (recursively defined as below), we say that the goal succeeds. If a goal cannot reach $\Box$ by means of finite applications of SLD resolution and negation as failure, we say that the goal (finitely) fails. In this paper, we deal with only finite failure.

(2) Negation as failure is a rule: A goal $\leftarrow \mathit{not}\ A$ succeeds if a goal $\leftarrow A$ fails, and a goal $\leftarrow \mathit{not}\ A$ fails if a goal $\leftarrow A$ succeeds for a ground atom A (that is, an atom containing no variables). We have a refined negation as failure, originally presented in Eshghi and Kowalski [3]:

(i) A goal $\leftarrow \mathit{not}\ A$ succeeds if a goal $\leftarrow A$ fails with the atom A in memory,

(ii) A goal $\leftarrow \mathit{not}\ A$ fails if a goal $\leftarrow A$ succeeds,

where A is a ground atom, sometimes called an abducible.

The combination of SLD resolution with negation as failure is called to be SLDNF resolution (SLD resolution with Negation as Failure). DGLP is supposed to have a network, through which each general logic program (regarded as distributed at a site) can communicate with each other. We now consider the case that we decide whether a goal $\leftarrow A$ fails with an atom A in memory, through the network of a distributed general logic program. In this situation, we suppose that a possible failure of the goal $\leftarrow A$ is inquired to each general logic program. If any trial for each program fails, we infer that the goal $\leftarrow \mathit{not}\ A$ succeeds.

EXAMPLE 1 Assume a DGLP $P =< P_1, P_2 >$ such that

$$\begin{aligned} P_1 &= \{p \leftarrow \mathit{not}\ q\}, \\ P_2 &= \{q \leftarrow r\}, \end{aligned}$$

where p, q and r are atoms (in propositional logic). Because a goal $\leftarrow q$ fails for both the programs P_1 and P_2, we can have a succeeding derivation for a goal $\leftarrow p$ in P_1, which requires the failing derivation for a goal $\leftarrow q$ in both P_1 and P_2.

We thus take a rule of "negation as failure through a network" as follows.

(i) A goal $\leftarrow \mathit{not}\ A$ succeeds if a goal $\leftarrow A$ fails for each general logic program with the ground atom A in memory.

(ii) A goal $\leftarrow \mathit{not}\ A$ fails if a goal $\leftarrow A$ succeeds.

Following the model theories as in You et al. [16] and Yamasaki et al. [13][14], we can have a sound distributed procedure with negation as failure through the network as well as with SLD resolution, with respect to some 3-valued logic model [15]. The distributed procedure is implemented, as shown in what follows.

3. A communication environment of distributed logic programs

3.1. A communication environment

A distributed logic program environment is designed, based on the following approaches. The procedural description of the environment will be shown in section 3.2.

- Each logic program $P_i(1 \leq i \leq n)$, which is a part of a distributed logic program is implemented as an independent program. We call it an *Independent Logic Program* (*ILP*, for short).
- We establish a procedure *Session* for the communication between *ILP*s. A *Session* knows which *ILP* participates in this *Session* and controls messages to/from other *ILP*s. A *Session* realizes a DGLP.
- There may be more than one *ILP* and *Session* in an environment. An *ILP* can participate in more than one *Session*. A *Session* can consist of more than one *ILP*.
- A process *Reasoning* is a sequence of derivations that begin with a given goal.
- In a *Session*, more than one *Reasoning* can be executed simultaneously.
- A *Session* can visualize a state of *Reasoning*s by histories of messages.

3.2. Procedural description

In this section, we describe a behaviour of an *ILP* and a *Session* in propositional logic. We also provide a protocol of message on a *Reasoning*. (The procedure is formally given in [15].)

The distributed procedure consists of three derivations: succeeding and failing derivations on an *ILP* and a network failing derivation on a *Session*. The succeeding and failing derivations consist of two module as in [3][15]: one is for the case that a goal is a positive literal and another is for a negative literals. The structure of the distributed procedure are as follows:

1. The succeeding derivation (on an *ILP*)
    - SPmodule (for a positive literal)
    - SNmodule (for a negative literal)
2. The failing derivation (on an *ILP*)
    - FPmodule (for a positive literal)
    - FNmodule (for a negative literal)
3. The network failing derivation (on a *Session*)

The succeeding derivation

The succeeding derivation aims at a proof for the given goal. The succeeding derivation from a goal G of length $h(h \geq 0)$ is a sequence of pairs

$$(G_0, \Delta_0), \ldots, (G_h, \Delta_h),$$

where $G_0, \ldots, G_h$ are goals, and $\Delta_0, \ldots, \Delta_h$ are sets of atoms. This is denoted by $(G_0, \Delta_0) \Rightarrow_{suc} (G_h, \Delta_h)$.

(G_{i+1}, Δ_{i+1}) is constructed from (G_i, Δ_i) by selecting a literal M_k in G_i and executing SPmodule or SNmodule if M_k is a positive or negative literal, respectively.

SPmodule

SPmodule finds a clause whose head is the literal, and replaces the literal in the goal to the body of the clause.

```
SPmodule
 if there is a clause $M_k \leftarrow N_1, \ldots, N_m$
 then
  begin
  $G_{i+1} :=\leftarrow M_1, \ldots, M_{k-1}, N_1, \ldots, N_m, M_{k+1}, \ldots, M_n$;
  $\Delta_{i+1} := \Delta_i$
  end
```

SNmodule

SNmodule checks whether the literal is already examined at first. If the literal is already examined, SNmodule uses the result. If the literal is not yet examined, SNmodule invokes a network failing derivation for the corresponding positive literal. Assume the focused literal $M_k = \sim L$.

```
SNmodule
 if $L \in \Delta_i$
 then
  begin
  $G_{i+1} :=\leftarrow M_1, \ldots, M_{k-1}, M_{k+1}, \ldots, M_n$;
  $\Delta_{i+1} := \Delta_i$
  end
 else
  if there is a network failing derivation
     $((\{\leftarrow L\}, \Delta_i \cup \{L\})) \Rightarrow_{nff} (H', \Delta')$
  then
   begin
   $G_{i+1} :=\leftarrow M_1, \ldots, M_{k-1}, M_{k+1}, \ldots, M_n$;
   $\Delta_{i+1} := \Delta'$
   end
```

The failing derivation

The failing derivation aims at no proof for the given goal. For a set H of goals, a (finitely) failing derivation of length k $(k \geq 0)$ is a sequence of pairs

$$(H_0, \Delta_0), \ldots, (H_k, \Delta_k),$$

where $H_0, \ldots, H_k$ are goal sets, such that $\Box \notin H_j$ for $1 \leq j \leq k$, and $\Delta_0, \ldots, \Delta_k$ are sets of atoms. This is denoted by $(H_0, \Delta_0) \Rightarrow_{ff} (H_k, \Delta_k)$.

(H_{j+1}, Δ_{j+1}) is constructed from (H_j, Δ_j) by selecting a literal M_k in G_i which is one of goals in H_j and executing FPmodule or FNmodule if M_k is a positive or negative literal, respectively. Assume $H_j = H'_j \cup \{G_i\}$, where $G_i :=\leftarrow M_1, \ldots, M_k, \ldots, M_n$.

FPmodule

If a focused literal is a positive literal in a failing derivation, FPmodule is invoked. FPmodule finds clauses of which head is the literal, and makes a list of goals, by replacing the literal in the goals to the body of the clauses.

```
FPmodule
  if all the clauses are $M_k \leftarrow N_1^q, \ldots, N_{n_q}^q (1 \le q \le p)$
   such that the head is $M_k$
  then
    begin
    for $(1 \le q \le p)$ do
      $G_{i+1}^q := \leftarrow M_1, \ldots, M_{k-1}, N_1^q, \ldots, N_{n_q}^q,$
            $M_{k+1}, \ldots, M_n;$
    $H_{j+1} := H'_j \cup \{G_{i+1}^1, \ldots, G_{i+1}^p\};$
    $\Delta_{j+1} := \Delta_j$
    end
  else
   begin
   $H_{j+1} := H'_j;$   $\Delta_{j+1} := \Delta_j$
   end
```

FNmodule

FNmodule checks whether the literal is already examined, at first. If the literal is already examined, FNmodule uses the result. If the literal is not yet examined, FNmodule invokes a succeeding derivation for the corresponding positive literal. Assume the focused literal as $M_k = \sim L$.

```
FNmodule
  if $L \in \Delta_i$
  then
   begin
   $H_{j+1} := H'_j \cup \{M_1, \ldots, M_{k-1}, M_{k+1}, \ldots, M_n\};$
   $\Delta_{j+1} := \Delta_j$
   end
  else
   if there is $(\{\leftarrow L\}, \Delta_j) \Rightarrow_{suc} (\Box, \Delta')$
    then
     begin
     $H_{j+1} := H'_j;$   $\Delta_{j+1} := \Delta'$
     end
```

The network failing derivation

A *Session* S consists of *ILP*s. Let $S = \{P_0, \ldots, P_n\}$, where $P_0, \ldots, P_n$ are the *ILP*s. A network failing derivation

$$(H_0, \Delta_0) \Rightarrow_{nff} (H', \Delta')$$

on S is implemented by the NetworkFailingModule:

```
NetworkFailingModule
  begin
  for $(0 \le i \le n)$ do simultaneously
   call a failing derivation of $P_i$;
  if $(H_0, \Delta_0) \Rightarrow_{ff} (H_i, \Delta_i)$ for all $P_i$ $(0 \le i \le n)$
  then
   begin
   $H' = \emptyset;$   $\Delta' = \Delta_0 \cup \ldots \cup \Delta_n$
   end
  end
```

A network failing derivation is invoked by a message from a succeeding derivation. A failing derivation is activated by a message from a network failing derivation. Table 1 shows messages we define. They are sent at the start and an end of the derivations.

We have the following properties.

Table 1. Messages

Message	Sender	Receiver	Comments
SFAIL	*ILP*	*Session*	The start of a network failing derivation. It is sent by a succeeding derivation.
SFAILR	*Session*	*ILP*	The end of a network failing derivation. It is received by a succeeding derivation.
FAIL	*ILP*	*Session*	The start of a failing derivation.
FAILR	*Session*	*ILP*	The end of a failing derivation.

PROPERTY 1 *Derivation calls are represented in a tree: nodes are derivations (succeeding, failing and network failing) and arcs connect a calling derivation to a called derivation.*

PROPERTY 2 *The end message e_i is sent/received after the end message e_j in a path from the root node to a leaf node in a derivation call tree.*

4. A visualization method for communications in distributed logic programs

4.1. Overview

A three-dimensional visualization is performed by a *Session*. A *Session* uses a history of messages to visualize a behaviour of distributed logic programs. The axes of the visualization represent different parametric factors of messages.

A message has three parametric factors: a time at which the message is sent, a path which the massage belongs to and a hierarchical level in the derivation call tree. We map time into x-axis, paths into z-axis and hierarchical levels into y-axis. A *Session* records the history of messages, and visualizes them at any time (usually by user's instructions).

In a *Session*, more than one *Reasoning* can be executed simultaneously. The visualization can display them in a figure. Our implemented system can highlight a *Reasoning* by colouring the history of messages for the *Reasoning*.

The advantages of the visualization are as follows:

- We can show the state of a *Session*.
- We can know how derivation calls are evoked in a *Reasoning*.
- If there is a derivation that does not send an end message, we can find it in a visualized figure.

4.2. Implementation

The system we have implemented consists of two parts: a *Session* and an *ILP*. A *Session* controls and records messages and knows which *ILP*s are participated in, when a *Reasoning* is being performed. An *ILP* has a logic program and it can perform derivations.

In our visualization, a message is represented as a coloured cube. Its coordinates are defined by the time, by the *ILP* which sends/receives the message, and by the hierarchical level. The three parametric factors of messages are given by a *Session*.

time: A *Session* has a global clock. The time is set by the clock. The time of a message is determined when the message is sent/received by the *Session*.

path: A path is calculated by the derivation call tree of the *Reasoning*. In the visualization, to give the feeling of a spatial extension of a network, messages that are executed on the same processor are placed in the neighborhood. But the messages have different z-coordinate if they belong to different paths.

hierarchical level: A hierarchical level is also calculated by the derivation call tree. The hierarchical level of a message is the depth of the corresponding derivation in the derivation call tree.

We connect messages with lines according to the semantic configuration of messages. Lines are drawn by applying the following rules:

- In the case of an SFAIL message, connect it to all FAIL messages caused by the SFAIL message.
- In the case of an FAILR message, connect it to an SFAILR message that pairs off with the SFAIL message which called the FAILR message.
- In other cases, connect a message to the next message which has the successive value time on the same values of the path and the hierarchical level.

In the visualization, we add extra cubes that represent the start and the end of a *Reasoning*. The cubes are drawn by a different colour from cubes that represent messages.

EXAMPLE 2 *Figure 1 shows the status of a Session with a Reasoning. The Reasoning has been performed on two ILPs:* $P_1 = \{A \leftarrow not\ B, C, not\ D\}, P_2 = \{B \leftarrow E, E \leftarrow \Box\}$*. The given goal is* $\leftarrow A$ *to* P_1*. In this figure, time goes left to right, paths are placed front to back, and hierarchical level represents top to bottom. We can know this Reasoning already finished because the node for the end of the reasoning is represented.*

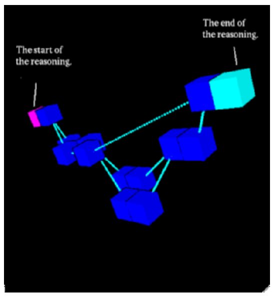

Figure 1. Communications in a *Reasoning* with two *ILP*s.

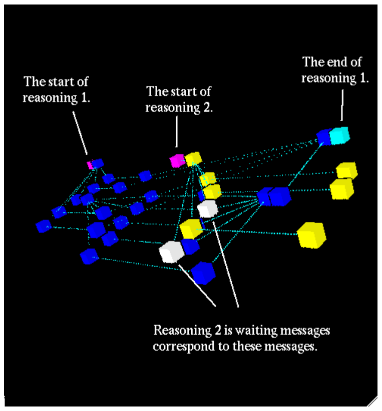
The start of reasoning 1.
The start of reasoning 2.
The end of reasoning 1.
Reasoning 2 is waiting messages correspond to these messages.

Figure 2. Communications in two *Reasoning*s with five *ILP*s.

EXAMPLE 3 *Figure 2 shows the status of a Session with two Reasonings. The Reasonings have been performed on five ILPs:*$P_1 = \{A \leftarrow not\ B\}, P_2 = \{B \leftarrow C\}, P_3 = \{B \leftarrow not\ D\}, P_4 = \{D \leftarrow not\ E\}, P_5 = \{E \leftarrow \Box\}$ *with a given goal* $\leftarrow A$ *to* P_1 *and a given goal* $\leftarrow B$ *to* P_3*. The former Reasoning finished but the latter have not finished. The latter Reasoning is focused in the figure. The start messages are represented as a special colour if the corresponding end messages are not sent to.*

5. Related works

There are many related works in visualization of the behaviour of parallel or concurrent programs. Most of them visualize messages in two-dimension [7][8]. One axis represents time and another is for processors. This type of visualization is simple and general.

Carr et al. [1] implemented a communication library with a visualization for some use in undergraduate courses.

The visualization shows the synchronization among the threads in two-dimension to help students understand complex communications.

Visper [12] presented a graph based on a software visualization tool for some parallel message-passing programming. They provide a two-dimensional graph which extends the time axis as a control axis, and adds the concept of process groups to the process axis.

Is the three-dimensional visualization generally superior to the two-dimensional visualization? Cuckburn and Mckenzie [2] compare the effect of the three-dimensional visualization with the two-dimensional visualization in a document system. They report that the two-dimensional visualization is slightly superior in efficiency, but the three-dimensional visualization is significantly superior in preference. Therefore, an approach to a three-dimensional visualization of communications in distributed program environments is taken.

VisuaLinda [5] visualizes parallel Linda programs. The system is similar to ours. The visualization function is implemented in a server. The system visualizes the behaviour of parallel programs in three-dimensions. However, since VisuaLinda covers general programs, its visualization consists of two parametric factors: time and processors.

Because our visualization specializes in distributed logic programs, it makes use of three parametric factors: time sequence, identification of processors (procedures) and hierarchical level. Therefore it naturally demonstrates the behaviour of distributed logic programs in three-dimensions.

6. Concluding remarks

We propose a three-dimensional visualization of communications in distributed logic programs. The visualization makes use of a semantic configuration of messages of which three parametric factors are mapped to the three axes respectively. The visualization function is included in the constructed procedure *Session* which realizes the reasoning caused by a DGLP. It can demonstrate messages communicated through several sequences *Reasoning*s which are executed simultaneously in the *Session*. We implement a visualization system, where we can select a sequence *Reasoning* from its output figure, and focus on the messages of the *Reasoning*. We could also see the status of active *Reasoning*s waiting for messages. The visualization may be useful to show dynamic changes of a distributed logic program: that is, increments/decrements of *ILP*s.

While we deal with the visualization of communications in distributed logic program environments, the visualization is applicable to other parallel/distributed program environments, as we have visualized a program structure of a parallel program [10].

References

[1] S. Carr, C. Fang, T. Jozwowski, J. Mayo, and C.-H. Sene. A communication library to support concurrent programming courses. *SIGCSE Bull.*, 34(1):360–4, 2002.

[2] A. Cockburn and B. Mckenzie. 3d or not 3d? evaluating the effect of the third dimension in a document management system. *CHI2001 Conference on Human Factors in Computing Systems*, pages 434–41, 2001.

[3] K. Eshghi and R. Kowalski. Abduction compared with negation by failure. *Proc. of 6th ICLP*, pages 234–255, 1989.

[4] A. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *J. of Logic and Computation*, 2:719–770, 1992.

[5] H. Koike, T. Takada, and T. Masui. VisuaLinda: A framework for visualizing parallel linda programs. *Proceedings of 1997 IEEE Symposium on Visual Languages*, pages 176–82, 1997.

[6] J. Lloyd. *Foundations of Logic Programming, 2nd, Extended Edition*. Springer – Verlag, 1993.

[7] W. Mueller, A. Meyer, and H. Zabel. A visual framework for the scripting of parallel agents. *Proceedings of 2000 IEEE Symposium on Visual Languages*, pages 77–80, 2000.

[8] T. Oshiba and J. Tanaka. "3d-pp": Three-dimensional visual programming system. *Proceedings of 1999 IEEE Symposium on Visual Languages*, pages 189–90, 1999.

[9] R. Ramanujam. Semantics of distributed definite clause programs. *Theoretical Computer Science*, 68:203–220, 1989.

[10] M. Sasakura, K. Joe, Y. Kunieda, and K. Araki. Naraview: an interactive 3d visualization system for parallelization of programs. *International Journal of Parallel Programming*, 27(2):111–129, 1999.

[11] J. Shepherdson. Negation in logic programming. *Foundations of Deductive Databases and Logic Programming*, pages 19–88, 1987.

[12] N. Stankovic and K. Zhang. Visual programming for message-passing systems. *International Journal of Software Engineering and Knowledge Engineering*, 9(4):392–423, 1999.

[13] S. Yamasaki and Y. Kurose. Soundness of abductive proof procedure with respect to constraint for non-ground abducibles. *Theoretical Computer Science*, 206:257–281, 1998.

[14] S. Yamasaki and Y. Kurose. A sound and complete procedure for a general logic program in non-floundering derivations with respect to the 3-valued stable model semantics. *Theoretical Computer Science*, 266:489–512, 2001.

[15] S. Yamasaki and M. Sasakura. Towards distributed programming systems with visualizations based on nonmonotonic reasoning. *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet(CD-ROM) 76*, 2001.

[16] J.-H. You and L. Yuan. On the equivalence of semantics for normal logic programs. *J. of Logic Programming*, 22:211–222, 1995.