IEICE TRANS. INF. & SYST., VOL.E87–D, NO.5 MAY 2004

PAPER

Negation as Failure through a Network

Kazunori IRIYA[†*a)], *Nonmember and* **Susumu YAMASAKI**[†], *Member*

SUMMARY This paper deals with distributed procedures, caused by negation as failure through a network, where general logic programs are distributed so that they communicate with each other in terms of negation as failure inquiries and responses, but not in terms of derivations of SLD resolutions. The common variables as channels in share for distributed programs are not treated, but negation as failure validated in the whole network is the object for communications of distributed programs. We can define the semantics for the distributed programs in a network. At the same time, we have distributed proof procedures for distributed programs, by means of negation as failure to be implemented through the network, where the soundness of the procedure is guaranteed by the defined semantics.

key words: *SLDNF resolution, distributed computing*

1. Introduction

Negation as failure is a rule to infer a negated predicate *not A* if the predicate *A* cannot be derived from a given formal system. For the logic program containing negation as failure sign, that is, the general logic program, the proof procedures have been established by means of SLDNF resolution combined with negation as failure (NAF, for short) ([11]). We firstly review its aspects in the following item (a), where some causal theory has been developed with negation as failure in abductive logic programming ([8]). How far negation as failure can be available for causal theory in distributed programming environments may be an interesting problem, because the retrieval in distributed deductive database, for example, is likely to take a similar manner to the reasoning by negation as failure. To have approach to such a problem, we present a rule of negation as failure applicable to the network, where its outline is stated in the following item (b). In the item (c), we have some intuitive insight into a semantic issue for the presented rule.

(a) Negation as failure is built in SLDNF resolution to act on the general logic program, in which we are free from the Horn clause logic representation without negation, enjoying more expressive programs involving negative literals. It is of use for abduction to infer the case from the rule (as a program) and the result (as represented by a goal statement). Abduction is significant in the sense that it is a kind of causal theory for the explanation of the cause (that is, the case) from the effect (that is, the result) with a rule. The SLDNF resolution with some refinements is so well-established in that the relations between the procedure involving refinements of SLDNF resolution and semantics have been much studied. The semantics for the procedure is related to the 3-valued stable model ([15]), which is not necessarily the well founded model ([18]) nor the finite failure stable model ([6]), so that the procedure is apart from the 2-valued stable model ([5]). The general semantics by argumentation theory ([2], [3]) as well as an abstract procedure is given in [9]. A more concrete, complete procedure for the 3-valued stable model is investigated in [14] and fully presented in [20], as well.

(b) We now care the environments constructed by a network of general logic program. There have been some studies on distributed logic programs, as in [13], [16]. This paper does not intend to present any strategy to cope with the difficulties in concurrent and/or distributed logic programming. The paper facilitates distributed proof procedures without common resolution deductions, while it focuses on negation as failure. The common variables of distributed, different logic programs are out of treatments in this paper. The negation as failure is generalized to be implemented through the given network of general logic programs. Because the safe rule for negation as failure (for the safe rule, see [11], [17]) can be kept even in the network of general logic programs, we may make use of negated predicates. This is the case that the programs with negations are distributed and are not to be copied from others. In such a case, we deal with negation as failure through the network, which is inferred by all the finite failures of all distributed programs. In this paper, distributed proof procedures are implemented by means of proof procedures for distributed programs, as suggested in [22]:

(1) The proof procedure for each program contains the succeeding derivation evoked by SLD resolution, and negation as failure. Except the negation as failure in terms of failing derivations, the succeeding derivation is the same as in a proof procedure ([3], [4]) for a single general logic program.

(2) The failing derivation for each program inquires the failure to all the programs through the network. In case that all the failing derivations respond to this inquiry, the negation as failure is supported for the inquiring procedure. Except this requirement for negation as failure through the network, the failing derivation works as in a proof procedure ([3]) for a single general logic program.

(3) Each general logic program responds to the inquiry

Manuscript received May 10, 2002.
Manuscript revised January 7, 2004.

[†]The authors are with the Department of Intelligence Computing and System, Graduate School of Natural Science and Technology, Okayama University, Okayama-shi, 700-8530 Japan.
[*]Presently, with Hewlett-Packard Solutions Delivery, Ltd.
a) E-mail: iriyak@jpn.hp.com

for the failure from other programs.

(c) As far as negation as failure through a network is adopted, each general logic program of the network is regarded as necessary means to detect failure. The more programs there are in a network, the more difficult failure is detected. On the other hand, the more programs there are, the more negated predicates are acquired. To denote the intended, distributed proof procedures, we will have semantics by means of alternating operators to capture the negation as failure through the network. Note that the alternating fixpoint semantics is in general regarded as the 3-valued stable model but often stands for the well-founded model (the least 3-valued stable model) of general logic programs [15], [18], [19], [23]. Extended for a network of general logic programs, a fixpoint semantics can be defined such that the constructed, distributed proof procedures are guaranteed to be sound with respect to the defined semantics.

The paper is organized as follows. In Sect. 2, notations in logic programming and for distributed general logic programs are illustrated. In Sect. 3, for a network of general logic programs, we construct distributed abductive proof procedures by means of negation as failure through a network. In Sect. 4, semantics of a distributed general logic program is defined by means of alternating operators. In Sect. 5, the soundness of distributed proof procedures is shown with respect to the defined semantics. Section 6 gives concluding remarks.

2. Notations for Distributed General Logic Programs

On the basis of [9], [11], [17], [20], notations in logic programming are reviewed, and a negation as failure through a network is illustrated.

A *general logic program* is a set of clauses of the form $A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n (0 \leq m \leq n)$, where $A_0, A_1, \ldots, A_m$ are atoms (positive literals) and $\mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ are negations of atoms (negative literals). A_0 is the *head* of the clause and $A_1 \ldots A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ 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 $L_1, \ldots, L_n$ 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 $L_1, \ldots, L_n$ are literals. The *goal* containing only negative literals is said to be a negative goal. The empty clause containing no head nor body is denoted by $\square$, where the empty clause is regarded as a negative goal.

The empty substitution is denoted by ε. That is, $\varepsilon(x) = x$ for any $x \in Var$. An application of a substitution θ to an expression E (say, a literal, a term, a rule or a goal), $E\theta$, denotes the expression obtained by substituting all the variables in E for terms according to θ. $E\theta$ is said to be an *instance* of E. If $E\theta$ contains no variables, $E\theta$ is called a *ground instance*. A ground atom, clause, goal or program is a ground instance of an atom, a clause, a goal, or a program, respectively.

The composition of substitutions θ and φ is denoted by $\theta\varphi$.

For the substitution as a most general unifier of more than 2 expressions, see [11]. A most general unifier is applied to SLD resolution (with negation as failure). For SLD resolution with negation as failure (SLDNF resolution, for short), see [11]. $\leftarrow A_1\theta, \ldots, A_{i-1}\theta, L_1\theta, \ldots, L_k\theta, A_{i+1}\theta, \ldots, A_m\theta, \mathit{not}\ A_{m+1}\theta, \ldots, \mathit{not}\ A_n\theta$ is derived by SLD resolution from $\leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ and $A \leftarrow L_1, \ldots, L_k$ such that θ is a most general unifier of A_i $(1 \leq i \leq m)$ and A. The derived goal is referred to as a resolvent in later sections.

In SLDNF resolution, we make use of *negation as failure* rules:

$\leftarrow \mathit{not}\ A$ succeeds if $\leftarrow A$ fails, and
$\leftarrow \mathit{not}\ A$ fails if $\leftarrow A$ succeeds
for a ground atom A. For an abductive proof procedure based on SLDNF resolution, we have a refined negation as failure, originally presented in [4]:

$\leftarrow \mathit{not}\ A$ succeeds if $\leftarrow A$ fails by keeping A stored to regard $\mathit{not}\ A$ as inferred, and
$\leftarrow \mathit{not}\ A$ fails if $\leftarrow A$ succeeds
for a ground atom A.

By the refined negation as failure, at the same time when $\leftarrow A$ fails so that $\leftarrow \mathit{not}\ A$ succeeds, $\mathit{not}\ A$ is interpreted as inferred like abduction. We call the corresponding atom A to be abducible. By *Abd*, we stand for the set of abducible atoms, in later sections. Extending the notion of a distributed logic program as in [16], we have a *distributed general logic program*. In Sect. 3, we give detailed descriptions of procedures for a distributed general logic program.

Definition 2.1: A distributed general logic program (DGLP, for short) is a tuple $\langle P_1, \ldots, P_n \rangle$ $(n \geq 1)$, where P_i is the general logic program.

3. Distributed Proof Procedures

In this section we have proof procedures for DGLPs. We first have an intuitive insight into the descriptions of the procedures. 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 take the case that we decide whether $\leftarrow A$ fails by keeping A stored to regard $\mathit{not}\ A$ as inferred, through the network of a distributed general logic program. In this situation, we suppose that the goal $\leftarrow A$ is inquired to each general logic program. If any trial for each program fails, we infer $\leftarrow \mathit{not}\ A$ to succeed. We now see that negation as failure is more rigid in the case of the network of a distributed general logic pro-

gram, than in the usually established case, as illustrated in what follows.

Example 3.1: Assume a DGLP $P1 = \langle P_1, P_2 \rangle$:

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

where p and q are atoms (in propositional logic). When we have only a general logic program P_1, the goal $\leftarrow p$ succeeds, because $\leftarrow q$ fails. On the other hand, for the DGLP $P1$, $\leftarrow q$ cannot fail, because of P_2, for which $\leftarrow q$ succeeds. Hence $\leftarrow p$ cannot succeed for the DGLP. Neither $\leftarrow p$ fails for P_1. Because $\leftarrow not\ q$ does not fail for P_1: Even if $\leftarrow q$ succeeds for P_2, but $\leftarrow q$ does not for P_1.

Example 3.2: Assume a DGLP $P2 = \langle P_1, P_2 \rangle$:

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

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

Therefore we take a negation as failure through a network as follows.

$\leftarrow not\ A$ succeeds if $\leftarrow A$ fails for each general logic program by keeping A stored to regard *not* A as inferred, and

$\leftarrow not\ A$ fails if $\leftarrow A$ succeeds.

3.1 Proof Procedure for General Logic Programs

For a single general logic program, a proof procedure is organized by two derivations, though there are varieties in details ([2]–[4], [9]), where infinite derivations are constructed in [21]. Assume that *Abd* stands for the set of abducibles, where the set of abducibles is the set of atoms, in this paper, for acquiring or abducting their negated literals by negation as failure.

Definition 3.3: Assume a (general logic) program Q, a *succeeding derivation* from (g_1, δ_1) to (g_m, δ_m) is a sequence

$$(g_1, \delta_1), (g_2, \delta_2), \ldots, (g_m, \delta_m)\ \ (m > 0)$$

such that, for each i $(1 \leq i \leq m)$, the goal g_i has the form $\leftarrow I, I'$, and $\delta_i \subseteq Abd$, where (without loss of generality) some selection rule R selects I (a literal), and I' is a (possibly empty) collection of literals.

(suc1) If I is an atom q, then

$$g_{i+1} = C \text{ and } \delta_{i+1} = \delta_i,$$

where C is a resolvent of some clause for the program Q by SLD resolution with the clause g_i on its selected literal I.

(suc2) If I is *not* q (where q is an atom) and $q \in \delta_i$, then

$$g_{i+1} = \leftarrow I' \text{ and } \delta_{i+1} = \delta_i.$$

(suc3) If I is *not* q (where q is an atom), $q \in Abd \setminus \delta_i$, and there is a failing derivation from $(\{\leftarrow q\}, \delta_i \cup \{q\})$ to $(\emptyset, \delta')$ for Q, then

$$g_{i+1} = \leftarrow I' \text{ and } \delta_{i+1} = \delta'.$$

A refutation is a succeeding derivation to $(\Box, \delta)$ for some $\delta \subseteq Abd$.

Definition 3.4: Assume a general logic program Q, a *failing derivation* from (f_1, δ_1) to (f_m, δ_m) is a sequence

$$(f_1, \delta_1), (f_2, \delta_2), \ldots, (f_m, \delta_m)\ \ (m > 0)$$

such that, for each i $(1 \leq i \leq m)$, the goal set f_i has the form $\{\leftarrow K, K'\} \cup f_i'$ and $\delta_i \subseteq Abd$, where (without loss of generality) the clause $\leftarrow K, K'$ has been selected (to continue the search), some selection rule R selects K (a literal), and f_i' is a set of goals. The goal set f_i is not empty for $1 \leq i < m$, where $f_m = \emptyset$.

(fail1) If K is an atom q, then

$$f_{i+1} = C' \cup f_i' \text{ and } \delta_{i+1} = \delta_i,$$

where C' is the set of all resolvents of clauses for Q by SLD resolution with the selected clause on its selected literal, and $\Box \notin C'$.

(fail2) If K is *not* q (where q is an atom), $q \in \delta_i$ and K' is not empty, then

$$f_{i+1} = \{\leftarrow K'\} \cup f_i' \text{ and } \delta_{i+1} = \delta_i.$$

(fail3) If K is *not* q (where q is an atom), $q \in Abd \setminus \delta_i$, and there is a succeeding derivation from $(\leftarrow q, \delta_i)$ to $(\Box, \delta')$ for some $\delta' \subseteq Abd$, then

$$f_{i+1} = f_i' \text{ and } \delta_{i+1} = \delta'.$$

3.2 Distributed Proof Procedures

We give a formal definition of proof procedures for a DGLP, where the set of abducibles should be stored every time the failing derivations required. A proof procedure for each general logic program with negation as failure through the network is constructed recursively.

We need 6 stages for a given DGLP, to define distributed proof procedures. For the formulation, we add one more stage for negation as failure through the network. We organize the procedures as follow.

Definition 3.5 (*A succeeding derivation*): Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, a succeeding derivation from (G_1, Δ_1) to (G_m, Δ_m) for P_j $(1 \leq j \leq n)$ is a sequence

$$(G_1, \Delta_1), (G_2, \Delta_2), \ldots, (G_m, \Delta_m)\ \ (m > 0)$$

such that, for each i $(1 \leq i \leq m)$, G_i has the form $\leftarrow I, I'$, and $\Delta_i \subseteq Abd$, where (without loss of generality) some selection rule R selects I, and I' is a (possibly empty) collection of literals.

(suc1) If I is an atom q, then

$$G_{i+1} = C \text{ and } \Delta_{i+1} = \Delta_i,$$

where C is a resolvent of some clause in P_j by SLD resolution with the clause G_i on its selected literal I;

(suc2) If I is *not* q (where q is an atom) and $q \in \Delta_i$, then

$$G_{i+1} = \leftarrow I' \text{ and } \Delta_{i+1} = \Delta_i.$$

(suc3) If I is *not* q (where q is an atom), $q \in Abd \setminus \Delta_i$, and there is a network failing derivation from $(\{\leftarrow q\}, \Delta_i \cup \{q\})$ to $(\emptyset, \Delta')$ for the DGLP P, then

$$G_{i+1} = \leftarrow I' \text{ and } \Delta_{i+1} = \Delta'.$$

A refutation is a succeeding derivation to $(\square, \Delta)$ for some $\Delta \subseteq Abd$.

Definition 3.6 (*A failing derivation*): Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, a failing derivation from (F_1, Δ_1) to (F_m, Δ_m) for P_j $(1 \le j \le n)$ is a sequence

$$(F_1, \Delta_1), (F_2, \Delta_2), \ldots, (F_m, \Delta_m) \ \ (m > 0)$$

such that, for each i $(1 \le i \le m)$, F_i has the form $\{\leftarrow K, K'\} \cup F'_i$ and $\Delta_i \subseteq Abd$, where (without loss of generality) the clause $\leftarrow K, K'$ has been selected (to continue the search), some selection rule R selects K, and F'_i is a set of goals. F_i is not empty for $1 \le i \le m$, where F_m is empty.

(fail1) If K is an atom q, then

$$F_{i+1} = C' \cup F'_i \text{ and } \Delta_{i+1} = \Delta_i,$$

where C' is the set of all resolvents of clauses in P_j by SLD resolution with the selected clause on its selected literal, and $\square \notin C'$;

(fail2) If K is *not* q (where q is an atom), and $q \in \Delta_i$ and K' is not empty, then

$$F_{i+1} = \{\leftarrow K'\} \cup F'_i \text{ and } \Delta_{i+1} = \Delta_i.$$

(fail3) If K is *not* q (where q is an atom), $q \in Abd \setminus \Delta_i$, and there is a succeeding derivation from $(\leftarrow q, \Delta_i)$ to $(\square, \Delta')$ for some $\Delta' \subseteq Abd$ in P_j, then

$$F_{i+1} = F'_i \text{ and } \Delta_{i+1} = \Delta'.$$

For a DGLP, we extend the failing derivation to a version based on *negation as failure through a network* of the DGLP. That is, we have:

Definition 3.7 (*A network failing derivation*): Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$. If there is a failing derivation from $(\{\leftarrow q\}, \Delta_i \cup \{q\})$ to $(\emptyset, \Delta^j)$ for some $\Delta^j \subseteq Abd$, by each P_j $(1 \le j \le n)$ of the DLP P, then we say that there is a network failing derivation from $(\{\leftarrow q\}, \Delta_i)$ to $(\emptyset, \Delta')$ for the DGLP P, where

$$\Delta' = \cup_{1 \le j \le n} \Delta^j.$$

Example 3.8: As an illustration, assume a DGLP $P2 = \langle P_1, P_2 \rangle$ as in Example 3.2. First, we see that there is a failing derivation from $(\{\leftarrow r\}, \{q\})$ to $(\emptyset, \{q\})$ for P_1, because the rule (fail1) is applicable such that there is no clause derived by SLD resolution.

It follows that there is a failing derivation from $(\{\leftarrow q\}, \{q\})$ to $(\emptyset, \{q\})$ for P_1. And the above also holds for P_2. Therefore, by means of the rule (nfail), we have a network failing derivation from $(\{\leftarrow q\}, \{q\})$ to $(\emptyset, \{q\})$ for $P2$.

By the rule (suc3), we have a succeeding derivation from $(\leftarrow \textit{not } q, \emptyset)$ to $(\square, \{q\})$ for P_1.

Finally, we have a succeeding derivation from $(\leftarrow p, \emptyset)$ to $(\square, \{q\})$ for P_1.

4. Semantics of Distributed General Logic Programs

We generalize the alternating fixpoint semantics for a single general logic program to some fixpoint semantics of a DGLP. The semantics must be guaranteed with respect to which the constructed proof procedures are sound. Note that only failing derivations are examined through the network of a DGLP, but succeeding derivations are out of the network unless containing failing derivations. In relation to this restriction of the procedures, a semantics will be defined.

4.1 Alternating Fixpoint Semantics, Reviewed

Assume a general logic program Q, where its Herbrand base (that is, the set of ground atoms constructed by using function and predicate symbols occurring in Q) is denoted by B_Q. Let

$$\begin{aligned} Q_{gr} = \{&A_0\theta \leftarrow L_1\theta, \ldots, L_n\theta \mid \\ &(A_0 \leftarrow L_1, \ldots, L_n) \in Q \text{ such that} \\ &A_0\theta \leftarrow L_1\theta, \ldots, L_n\theta \\ &\text{is a ground instance for a substitution } \theta\} \end{aligned}$$

be a ground general logic program which contains all the ground clauses obtained from clauses of Q.

As in [19], [21], [23], we can organize some operators to define the alternating fixpoint semantics.

Definition 4.1: Let $K \subseteq B_Q$ for a general logic program Q. We define the set

$$\begin{aligned} Q[K] \quad = \quad \{&a \leftarrow a_1, \ldots, a_m, \bar{a}_{m+1}, \ldots, \bar{a}_n \mid \\ &a \leftarrow a_1, \ldots, a_m, \textit{not } a_{m+1}, \ldots, \textit{not } a_n \\ &\in Q_{gr}\} \cup \{\bar{b} \leftarrow \mid b \in K\}. \end{aligned}$$

A mapping $S_Q : 2^{B_Q} \to 2^{B_Q}$ is defined to be

$$S_Q(K) = U_{Q[K]} \uparrow \omega \cap B_Q,$$

where $U_R \uparrow \omega = \cup_{i \in \omega} U_R \uparrow i$ (ω: the set of natural numbers as a transfinite ordinal number) is a set for a definite program R, defined by

$$U_R \uparrow i = \begin{cases} \emptyset & (i = 0), \\ U_R(U_R \uparrow (i-1)) & (i > 0) \end{cases}$$

in terms of the set

$$U_R(J) = \{a | \exists (a \leftarrow a_1, \ldots, a_n) \in R_{gr} : [a_1, \ldots, a_n \in J]\}$$

for $J \subseteq B_R$.

Definition 4.2: Let Q be a general logic program. Then we define a mapping $\Theta_Q : 2^{B_Q} \rightarrow 2^{B_Q}$ to be $\Theta_Q(K) = \overline{S_Q(\overline{S_Q(K)})}$, where the over-line stands for the complement set with respect to the Herbrand base B_Q.

Note that $S_Q(K) = A_Q(\overline{K})$ for A_Q in [18], being monotonic with respect to "$\subseteq$", so that there is a fixpoint of Θ_Q. We call Θ_Q to be an alternating operator. Also there is a least fixpoint of Θ_Q, which is denoted by $lfp(\Theta_Q)$. It is essentially concerned with the well-founded model. As in [23], the following semantics is a 3-valued stable model which is not necessarily the same as the well-founded model.

Definition 4.3: For a general logic program Q, the pair $(S_Q(K), K)$ is a semantics of Q, if $\Theta_Q(K) = K$ such that $S_Q(K) \cap K = \emptyset$.

Note that $S_Q(K)$ denotes a true set, while K denotes a false set. As well, note that the semantics $(S_Q(K), K)$ of the general logic program Q is the well-founded model if K is the least fixpoint of Θ_Q.

4.2 Fixpoint Semantics for DGLP

Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$. Let $B_P = \cup_{1 \leq k \leq n} B_{P_k}$, where B_P is regarded as the Herbrand base of P. Intuitively speaking, we need an extension

$$\langle (S_{P_1}(F_1), F_1), \ldots, (S_{P_n}(F_n), F_n) \rangle$$

such that $S_{P_k}(F_k) \cap F_k = \emptyset$ $(1 \leq k \leq n)$, and $(F_1, \ldots, F_n)$ is a fixpoint of some operator.

For just a general logic program Q, $\Theta_Q(F)$ reflects the possible set of negations as failure for a false set F. As long as we are concerned with negation as failure through the network of a DGLP P, we care the set $\cap_{1 \leq k \leq n} \Theta_{P_k}(F_i)$ $(\cap_k \Theta_{P_k}(F_i)$, for short) for each F_i, where the complement operation is taken with respect to the Herbrand base of the DLP P. It is regarded as a false, common set constructed on the basis of a false set F_i. As we have a transformation of the set F caused by Θ_Q, we associate the replacement of F_i by $\cap_k \Theta_{P_k}(F_i)$ with a transformation whose fixpoint is available for a semantic construction. For simplicity, let Θ_{P_k} be Ψ_k. We now define a mapping $\Psi_P : (2^{B_P})^n \rightarrow (2^{B_P})^n$ to be

$$\Psi_P(F_1, \ldots, F_n) = (\cap_k \Psi_k(F_1), \ldots, \cap_k \Psi_k(F_n)).$$

As is easily seen, Ψ_P is monotonic with respect to the componentwise subset inclusion, so that there is its fixpoint. Hence we adopt the definition:

Definition 4.4: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$. If $(F_1, \ldots, F_n)$ is a fixpoint of Ψ_P such that $S_{P_k}(F_k) \cap F_k = \emptyset$ $(1 \leq k \leq n)$, we say that $\langle (S_{P_1}(F_1), F_1), \ldots, (S_{P_n}(F_n), F_n) \rangle$ is a (fixpont) semantics of P.

In accordance with the illustrations of the proof procedures, we have examples on semantics.

Example 4.5: We have a DGLP $P2 = \langle P_1, P_2 \rangle$ such that

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

where p and q are ground atoms (in propositional logic). Now assume

$$(F_1, F_2) = (\{q, r\}, \{q, r\}),$$

where the set pair is equivalent if equivalent componentwise. We see that:

$$\begin{aligned} &\Psi_1(F_1) \cap \Psi_2(F_1) = F_1, \ \text{and} \\ &\Psi_1(F_2) \cap \Psi_2(F_2) = F_2, \end{aligned}$$

such that $S_{P_1}(F_1) \cap F_1 = \{p\} \cap \{q, r\} = \emptyset$, and $S_{P_2}(F_2) \cap F_2 = \emptyset \cap \{q, r\} = \emptyset$. Therefore

$$\langle (S_{P_1}(F_1), F_1), (S_{P_2}(F_2), F_2) \rangle = \langle (\{p\}, \{q, r\}), (\emptyset, \{q, r\}) \rangle$$

is a semantics of P.

5. Soundness of Distributed Proof Procedures

We have a soundness proof for the presented procedures with respect to the fixpoint semantics of a DGLP.

5.1 Derivations and Relations to Alternating Operator

Following the soundness and completeness of SLD resolution with the least fixpoint semantics (least Herbrand model) for a definite program (see it for [11]), we can have relations between the operator S_{P_i} and the derivations. The proofs can be given by the similar method for those in case of each general logic program, which are presented as in [21].

Lemma 5.1: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, where B_P is the Herbrand base of P.

1. There is a succeeding derivation from$(\leftarrow L_1, \ldots, L_m, \Delta_0)$ to $(\Box, \Delta)$ for P_i $(1 \leq i \leq n)$ with an applied composed substitution θ, then

$$\begin{aligned} &\forall L_k, \forall \sigma : [[(L_k\theta)\sigma = q \in B_P \Rightarrow q \in S_{P_i}(\Delta)] \\ &\wedge [(L_k\theta)\sigma = not\ q \text{ for } q \in B_P \Rightarrow q \in \Delta]] \end{aligned}$$

2. There is a failing derivation from $(\{\leftarrow L_1, \ldots, L_m\}, \Delta_0)$ to $(\emptyset, \Delta)$ for P_i $(1 \leq i \leq n)$, then

$$\begin{aligned} &\forall \sigma, \exists L_k : [[L_k\sigma = q \in B_P \Rightarrow q \in \Theta_{P_i}(\Delta)] \\ &\wedge [L_k\sigma = not\ q \text{ for } q \in B_P \Rightarrow q \in S_{P_i}(\Delta)]] \end{aligned}$$

We now show that the proof procedures can work consistently in the sense that if the set Δ of abducibles is obtained for a general logic program P_k through the network of a DGLP, then $S_{P_i}(\Delta)$ and Δ have no common atom.

Theorem 5.2: Assume that a DGLP $P = \langle P_1, \ldots, P_n \rangle$, and

there is a succeeding derivation from $(\leftarrow L_1, \ldots, L_m, \emptyset)$ to $(\square, \Delta)$ for some P_k $(1 \leq k \leq n)$. For any i $(1 \leq i \leq n)$:

1. $S_{P_i}(\Delta) \cap \Delta = \emptyset$.
2. $\Delta \subseteq \cap_i \Psi_i(\Delta)$.

Proof. (1) By means of the completeness of SLD resolution for a definite program with respect to the least fixpoint semantics, the succeeding derivation is related to $S_{P_i}(\Delta_0)$ for an adequate set Δ_0.

Assume that $a \in S_{P_i}(\Delta) \cap \Delta$. Because $a \in S_{P_i}(\Delta)$, there is a succeeding derivation from $(\leftarrow a, \emptyset)$ to $(\square, \Delta')$ in P_i for some $\Delta' \subseteq \Delta$. Then there is a succeeding derivation from $(\leftarrow a, \emptyset)$ to $(\leftarrow \mathit{not}\ r, \emptyset)$ and from $(\leftarrow \mathit{not}\ r, \Delta)$ to $(\square, \Delta')$ such that $r \in \Delta' \subseteq \Delta$, or $\square$ is derived.

Because $a \in \Delta$ (for P_k), there is a network failing derivation from $(\{\leftarrow a\}, \Delta_0 \cup \{a\})$ to $(\emptyset, \Delta_1)$ for the DGLP P such that $\Delta_0 \cup \{a\} \subseteq \Delta_1 \subseteq \Delta$, and there is a failing derivation from $(\{\leftarrow a\}, \Delta_0 \cup \{a\})$ to $(\emptyset, \Delta_2)$ for P_i such that $\Delta_0 \cup \{a\} \subseteq \Delta_2 \subseteq \Delta_1$. Then for some r, there is a failing derivation from $(\{\leftarrow \mathit{not}\ r\}, \Delta_0')$ to $(\emptyset, \Delta_2)$ for $\Delta_0 \cup \{a\} \subseteq \Delta_0'$ in P_i such that there is a succeeding derivation from $(\leftarrow r, \Delta_0')$ to $(\square, \Delta_3)$ for some $\Delta_3 \subseteq \Delta_2$ or $\square$ is derived.

By Lemma 5.1 and monotonicity of S_{P_i}, we see that if there is a succeeding derivation from $(\leftarrow r, \Delta_0')$ to $(\emptyset, \Delta_3)$ for P_i, then $r \in S_{P_i}(\Delta_3) \subseteq S_{P_i}(\Delta)$. Then there are a couple of cases:

(i) $r = a \in S_{P_i}(\Delta) \cap \Delta$.
(ii) $r \in S_{P_i}(\Delta) \cap \Delta$, where $r \neq a$.

In case of (i), because $r = a \in \Delta \cup \{a\}$, $\leftarrow \mathit{not}\ r$ is reduced to $\square$. This contradicts that $a \in \Delta$. Therefore $a \notin S_{P_i}(\Delta) \cap \Delta$. In case of (ii), we reach the case of (i), by the repetitions of the case that $p \in S_{P_i}(\Delta) \cap \Delta$ for some p. Eventually this is a contradiction.

(2) Assume that $a \in \Delta$. By the construction of Δ concerning the proof procedures, we see that there is a network failing derivation from $(\{\leftarrow a\}, \Delta_0 \cup \{a\})$ to $(\emptyset, \Delta')$ for the DGLP P, and thus, there is a failing derivation from $(\{\leftarrow a\}, \Delta_0 \cup \{a\})$ to $(\emptyset, \Delta'')$ for each P_i $(1 \leq i \leq n)$ such that $\Delta_0 \cup \{a\} \subseteq \Delta'' \subseteq \Delta' \subseteq \Delta$. By Lemma 5.1, $a \in \Theta_{P_i}(\Delta'')$. Because of monotonicity of Θ_{P_i}, $\Theta_{P_i}(\Delta'') \subseteq \Theta_{P_i}(\Delta)$. It follows that $a \in \Theta_{P_i}(\Delta)$. Since i is arbitrary, $a \in \cap_i \Theta_{P_i}(\Delta) = \cap_i \Psi_i(\Delta)$. Hence $\Delta \subseteq \cap_i \Psi_i(\Delta)$. *q.e.d.*

We now present the preservation of consistency under the operator Ψ_P and the inclusiveness of consistency.

Lemma 5.3: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, where $\Psi_i = \Theta_{P_i}$ is defined for each i $(1 \leq i \leq n)$. For $X \subseteq \mathit{Abd}$, let $\mathit{Consis}_i(X)$ stands for a predicate to state that $X \subseteq \overline{S_{P_i}(X)}$. Then

1. $\mathit{Consis}_i(X) \Rightarrow \mathit{Consis}_i(\cap_k \Psi_k(X))$.
2. Consis_i is inclusive, that is,

$$\forall H \subseteq \mathit{Abd}\ (\mathit{chain}) : [\forall Y \in H : [\mathit{Consis}_i(Y) \Rightarrow \mathit{Consis}_i(\cup H)]].$$

Proof. (1) By monotoniciy of S_{P_i}, we see that

$$\begin{aligned}
\mathit{Consis}_i(X) &\Leftrightarrow X \subseteq \overline{S_{P_i}(X)} \\
&\Rightarrow S_{P_i}(X) \subseteq S_{P_i}(\overline{S_{P_i}(X)}) \\
&\Rightarrow \overline{S_{P_i}(\overline{S_{P_i}(X)})} \subseteq \overline{S_{P_i}(X)} \\
&\Rightarrow \cap_k \Psi_k(X) \subseteq \overline{S_{P_i}(X)} \\
&\Rightarrow S_{P_i}(\cap_k \Psi_k(X)) \subseteq S_{P_i}(\overline{S_{P_i}(X)}) \\
&\Rightarrow \overline{S_{P_i}(\overline{S_{P_i}(X)})} \subseteq \overline{S_{P_i}(\cap_k \Psi_k(X))} \\
&\Rightarrow \cap_k S_{P_k}(\overline{S_{P_k}(X)}) \subseteq \overline{S_{P_i}(\cap_k \Psi_k(X))}.
\end{aligned}$$

It follows that $\cap_k \Psi_k(X) \subseteq \overline{S_{P_i}(\cap_k \Psi_k(X))}$. That is, $\mathit{Consis}_i(\cap_k \Psi_k(X))$.

(2) Assume that $\mathit{Consis}_i(\cup H)$ does not hold for some chain, where $\forall Y \in H : [\mathit{Consis}_i(Y)]$. By the definition of S_{P_i},

$$\exists A : [A \in S_{P_i}(\cup H) \text{ and } A \in \cup H] \Rightarrow \exists Y \in H : [A \in S_{P_i}(Y) \text{ and } A \in Y].$$

This contradicts that $\mathit{Consis}_i(Y)$ for any $Y \in H$. *q.e.d.*

By fixpoint induction, we have the lemma on consistency of the fixpoint semantics. Before the lemma is shown, some terminologies are given.

Definition 5.4: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$. When $(F_1, \ldots, F_n)$ is given such that $S_{P_i}(F_i) \cap F_i = \emptyset$ $(1 \leq i \leq n)$, we say that $(F_1, \ldots, F_n)$ is consistent.

Definition 5.5: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$. When $(F_1, \ldots, F_n)$ is given, we define

$$\Psi_P^\alpha(F_1, \ldots, F_n) = \begin{cases} (F_1, \ldots, F_n) & \text{if } \alpha = 0, \\ \Psi_P(\Psi_P^{\alpha-1}(F_1, \ldots, F_n)) & \text{if } \alpha \text{ is a successor ordinal,} \\ \cup_{\beta<\alpha} \Psi_P^\beta(F_1, \ldots, F_n) & \text{if } \alpha \text{ is a limit order,} \end{cases}$$

where "$\cup_{\beta<\alpha}$" operates componentwise.

As is for the monotonic function, $\Psi_P^\alpha(F_1, \ldots, F_n)$ is a fixpoint of Ψ_P for some ordinal α. By means of the former lemma, we have:

Lemma 5.6: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, where $(F_1, \ldots, F_n)$ is consistent. If $\Psi_P^\alpha(F_1, \ldots, F_n)$ is a fixpoint, it is consistent.

Proof. For each $1 \leq k \leq n$, we define F_k^α to be

$$F_k^\alpha = \begin{cases} F_k & \text{if } \alpha = 0, \\ \cap_i \Psi_i(F_k^{\alpha-1}) & \text{if } \alpha \text{ is a successor,} \\ \cup_{\beta<\alpha} F_k^\beta & \text{if } \alpha \text{ is a limit order.} \end{cases}$$

As is seen,

$$\Psi_P^\alpha(F_1, \ldots, F_n) = (F_1^\alpha, \ldots, F_n^\alpha).$$

Assume that $\Psi_P^\alpha(F_1, \ldots, F_n)$ is a fixpoint. By Lemma 5.3 and fixpoint induction, $F_k^\alpha \subseteq \overline{S_{P_k}(F_k^\alpha)}$, that is, $S_{P_k}(F_k^\alpha) \cap F_k^\alpha = \emptyset$. Hence $\Psi_P^\alpha(F_1, \ldots, F_n)$ is consistent. *q.e.d.*

5.2 Soundness of Proof Procedures

By means of Lemma 5.3 and 5.6, we finally have the proof for the soundness of the distributed proof procedures.

Theorem 5.7: Assume a DGLP $P = \langle P_1, \ldots, P_n \rangle$, and there is a succeeding derivation from $(\leftarrow L_1, \ldots, L_n, \emptyset)$ to $(\Box, \Delta)$ with an applied composed substitution θ for some P_k $(1 \leq k \leq n)$. Then there is a fixpoint semantics of P, $(\Delta_1^f, \ldots \Delta_n^f)$, such that

(1) $\Delta \subseteq \Delta_k^f$,
(2) the fixpoint $(\Delta_1^f, \ldots, \Delta_n^f)$ is consistent, and
(3)

$$\forall L_j, \forall \sigma : [[(L_j\theta)\sigma = q \in B_P \Rightarrow q \in S_{P_k}(\Delta_k^f)] \wedge [(L_j\theta)\sigma = not\ q \text{ for } q \in B_P \Rightarrow q \in \Delta_k^f]].$$

Proof. (1) We have a fixpoint $\Psi_P^\alpha(\emptyset, \ldots, \emptyset, \Delta, \emptyset, \ldots, \emptyset)$ for some ordinal α. We denote the fixpoint by $(\Delta_1^f, \ldots, \Delta_n^f)$. By Theorem 5.2 (2), $\Delta \subseteq \cap_i \Psi_i(\Delta)$. As in the usual case, $\Delta \subseteq \Delta_k^f$.

(2) By Lemma 5.6, the fixpoint must be consistent.

(3) By Lemma 5.1 and by monotoniciy of S_{P_k},

$$(L_j\theta)\sigma = q \in S_{P_k}(\Delta) \Rightarrow q \in S_{P_k}(\Delta_k^f),$$
$$(L_j\theta)\sigma = not\ q \text{ such that } q \in \Delta \Rightarrow q \in \Delta_k^f.$$

This concludes the proof. *q.e.d.*

To have a fixpoint semantics satisfying the conditions in Theorem 5.7, we can alternatively take $(\Delta, \ldots, \Delta)$ as an initial tuple of abducible sets, which Ψ_P is applied to. In this case, $(\Gamma_1^f, \ldots, \Gamma_n^f)$ may be acquired as a fixpoint such that $\Delta_i^f \subseteq \Gamma_i^f$ $(1 \leq i \leq n)$, because $(\emptyset, \ldots, \emptyset, \Delta, \emptyset, \ldots, \emptyset) \subseteq (\Delta, \ldots, \Delta, \Delta, \Delta, \ldots, \Delta)$ in the component inclusion sense of "$\subseteq$".

6. Concluding Remarks

In this paper, a network of general logic programs is captured as a distributed general logic program. This framework is different from the distributed definite program ([17]) but concerned with negation as failure through the network. To avoid a difficulty of common variables in different programs, we care only negation as failure through the network, based on the safe rule, apart from the common uses of SLD resolutions. It also differs from the view in [10] of logic programs as multi-agents. We have the sound, distributed proof procedures for a distributed general logic program, which comprise the proof procedure for each general logic program by means of negation as failure through the network. The negation as failure through the network must be implemented so that negation as failure is supported by all the programs of the network.

A fixpoint semantics of a distributed general logic program is defined, to capture negation as failure through the network and to guarantee the soundness of the distributed proof procedures. Complete abductive proof procedures are to be designed on the basis of ideas on complete procedures for a general logic program as in [9], [21].

Algebraic analyses in concurrency are applicable to distributed systems ([1], [7]). Functional programming may involve concurrency by means of algebraic structures ([12]). In distributed logic programming, nondeterminism for SLD resolution and nonmonotonicity caused by negation as failure are so characteristic that the model theory (based on the alternating operator) is considered as adequate. At the same time, to make the proof procedures clearer for the design of implementation techniques, meta-expressions for distributed procedures must be needed. As a logic programming paradigm, distributed proof procedures can be implemented, nevertheless, the implementation had better be described in some meta-language. The way of such a description is left as an interesting subject.

References

[1] G. Bruns, Distributed Systems Analysis with CCS, Prentice-Hall, 1996.
[2] P.M. Dung, "Negation as hypothesis: An abductive foundation for logic programming," Proc. 8th ICLP, pp.3–17, 1991.
[3] P.M. Dung, "An argumentation—Theoretic foundation for logic programming," J. Log. Program., vol.22, pp.151–177, 1995.
[4] K. Eshghi and R.A. Kowalski, "Abduction compared with negation by failure," Proc. 6th ICLP, pp.234–255, 1989.
[5] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programs," Proc. 5th ICLP, pp.1070–1080, 1988.
[6] L. Giordano, A. Martelli, and M.L. Sapino, "Extending negation as failure by abduction: A three-valued stable model semantics," J. Log. Program., vol.26, pp.31–67, 1996.
[7] C.R.A. Hoare, Communicating Sequential Processes, Prentice-Hall, 1986.
[8] A.C. Kakas, R.A. Kowalski, and F. Toni, "Abductive logic programming," J. Log. Comput., vol.2, pp.719–770, 1992.
[9] A.C. Kakas and F. Toni, "Computing argumentation in logic programming," J. Log. Comput., vol.9, pp.515–562, 1999.
[10] R.A. Kowalski and F. Sadri, "From logic programming to multi-agent systems," Ann. Math. Artif. Intell., vol.25, pp.391–419, 1999.
[11] J.W. Lloyd, Foundations of Logic Programming, 2nd, Extended Edition, Springer-Verlag, 1993.
[12] F. Nielson, ed., ML with Concurrency, Monograph in Computer Science, Springer-Verlag, 1996.
[13] P.M. Pereira and R. Nasr, "DELTA-PROLOG: A distributed logic programming language," Proc. International Conference of Fifth Generation Computer Systems, pp.283–291, 1984.
[14] L.M. Pereira, N.A. Joaquim, and J.J. Alferes, "Derivation procedures for extended stable models," Proc. 12th IJCAI, pp.863–868, 1991.
[15] T. Przymusinski, "Well-founded semantics coincides with three-valued stable semantics," Foundamenta Informaticae, vol.13, pp.445–463, 1990.
[16] R. Ramanujam, "Semantics of distributed definite clause programs," Theor. Comput. Sci., vol.68, pp.203–220, 1989.
[17] J.C. Shepherdson, "A sound and complete semantics for a version of negation as failure," Theor. Comput. Sci., vol.65, p.343–371, 1989.
[18] A. Van Gelder, K.A. Ross, and J.S. Schlipf, "The well-founded semantics for general logic programs," J. ACM, vol.38, pp.620–650, 1990.
[19] A. Van Gelder, "The alternating fixpoint of logic programs with negation," J. Comput. Syst. Sci., vol.47, pp.185–221, 1993.

[20] S. Yamasaki and Y. Kurose, "Soundness of abductive proof procedure with respect to constraint for non-ground abducibles," Theor. Comput. Sci., vol.206, pp.257–281, 1998.
[21] S. Yamasaki and Y. Kurose, "A sound and complete proof procedure for a general logic program in non-floundering derivations with respect to the 3-valued stable model semantics," Theor. Comput. Sci., vol.266, pp.489–512, 2001.
[22] S. Yamasaki and M. Sasakura, "Towards distributed programming systems with visualizations based on nonmonotonic reasoning," Proc. SSGRR 2001, vol.76, 2001.
[23] J.-H. You and L.Y. Yuan, "On the equivalence of semantics for normal logic programs," J. Log. Program., vol.22, pp.211–222, 1995.

Kazunori Iriya received B.Eng. and M.Eng in information technology from Okayama University, in 1994 and 1996, respectively. He is now with Hewlett-Packard Solutions Delivery, Ltd. He has been involved in the development of Operation Support System and Business Support System for the Japanese telecom carriers.

Susumu Yamasaki received B.Eng. and M.Eng in electrical engineering and Eng.D. in information science from Kyoto University, in 1970, 1972 and 1980, respectively. During 1974–1987, he had been a member of the academic staff at Dept. of Information Science, Kyoto University. In 1985/86 he was a visiting fellow at Dept. of Computer Science, University of Warwick, U.K. In 1987, he joined Okayama University as a professor in computing and artificial intelligence. His research interests include theoretical computer science, computational logic and automated reasoning. He is a member of EATCS, AAAI, Mathematical Society of Japan.