JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 30, 669-686 (2014)

DOI:10.1688/JISE.2014.30.3.8

A Network Representation of First-Order Logic That Uses Token Evolution for Inference

HIDEAKI SUZUKI*, MIKIO YOSHIDA+ AND HIDEFUMI SAWAI*
*National Institute of Information and Communications Technology
Kobe, 651-2492, Japan
E-mail: {hsuzuki; sawai}@nict.go.jp
+BBR Inc.
Osaka, 530-0002, Japan
E-mail: yos@bbr.jp

A method to represent first-order predicate logic (Horn clause logic) by a data-flow network is presented. Like a data-flow computer for a von Neumann program, the proposed network explicitly represents the logical structure of a declarative program by unlabeled edges and operation nodes. In the deduction, the network first propagates symbolic tokens to create an expanded AND/OR network by the backward deduction, and then executes unification by a newly developed method to solve simultaneous equations buried in the network. The paper argues the soundness and completeness of the network in a conventional way, then explains how a kind of ambiguous solution is obtained by the newly developed method. Numerical experiments are also conducted with some data-flow networks, and the method's convergence ability and scaling property to larger problems are investigated.

Keywords: horn logic, data-flow network, inference, unification, evolution

1. INTRODUCTION

Network has been one of the most well-studied representation tools of knowledge in Artificial Intelligence. Back in the 1980s, some models that explicitly manipulate human words or concepts have been proposed and pursued by a number of authors: Peirce's existential graphs [19], semantic networks [16, 17], conceptual graphs [6, 23], and so on. Among these, in the conceptual graph, to represent a nested structure in predicate logic, Sowa introduced a hyper-concept named 'proposition' and succeeded in describing *any* knowledge in the form of first-order predicate logic. These highly human-oriented representation schemes, however, require computationally heavy operations for deductive inference, as represented by the semantic networks that need structural matching between graphs.

A more inference-oriented network model, Petri-net, was extensively studied by Murata *et al.* [7, 12-14]. Every atom being expressed as a 'place' and every term and argument set being expressed as a 'token', this model concisely represents Horn clause logic by a 'high-level' Petri-net (Petri-net with labeled arcs), and conducts forward [14] or backward [7] deduction by parallel firing of 'transitions'. The final answer is obtained simply from tokens arriving at the goal transition [14] or by combining unifiers accumulated in the transitions during the deduction [7]. The backward deduction can be regarded as graphical visualization of the SLD-resolution (Selective Linear Definite clause resolution), whereas the forward deduction suffers from a serious problem, token num-

Received February 28, 2013; accepted June 15, 2013.
Communicated by Hung-Yu Kao, Tzung-Pei Hong, Takahira Yamaguchi, Yau-Hwang Kuo, and Vincent Shin-Mu Tseng.

ber's explosion, when the depth of deduction is large.

As a revised method of this previous model, Suzuki *et al.* very recently proposed a concept of the network-based inference system named "Knowledge Transitive Network (KTN)" [27-29]. The KTN is an extension of the authors' former model named ATN (Algorithmically Transitive Network), a data-flow computational model with learning abilities [25, 26]. While propagating numerical tokens with 'reliability' values, the ATN can revise its inner algorithm/function by using the feedback information from the teaching signals (supervised learning). Taking the same scheme, the KTN manipulates symbolic information (such as terms in predicate logic) and infers with ambiguity.

Following [29], the present paper formulates the transformation scheme by which a data-flow graph (DFG [4, 21], *i.e.*, KTN) is constructed from a logic program. All the symbols such as constants, variables, and function symbols comprising a term in predicate logic are directly translated into symbol nodes. This enables the KTN to deal with not only variables but also function symbols explicitly and avoid some common problems in current approaches, including inefficiency owing to symbol grounding. Markov logic network in probabilistic logic programming [8, 18] or the answer set programming for non-monotonic reasoning [1, 11] are based upon the symbol grounding (elimination of variables) and unable to deal with an infinite constant set. The recent 'lifted' approaches on graphical models [3, 15, 22, 30] have pursued for a method to manipulate first-order logic within the framework of probabilistic reasoning; and yet, in formulating, these approaches consider only clauses free of function symbols which the KTN can naturally represent.

Another merit of the KTN is in the inference's ambiguity represented by the token reliability. Here, the reliability is roughly defined as inter-token consistency that reflects to what extent unification is solved successfully. The KTN infers with so-called backward deduction. By propagating symbolic tokens, the original DFG is expanded into an acyclic directed graph, to which our newly developed method named "ELiminating Inconsistency by SElection (ELISE)" is applied. ELISE solves unification by numerically maximizing the token reliability which finally produces an ambiguous answer if any contradiction is found in an expanded search graph. Note that this KTN's ambiguity is naturally introduced without any extra parameters in the original logic or the DFG. By contrast, to represent uncertainty, many recent approaches on probabilistic reasoning tag predicate logic with various parameters whose values are optimized with learning: weight of clauses (facts and rules) [8, 18], probability of clauses [5], weight of atoms [30], probability of particular atoms [20], and so on.

In the following, the paper makes a full description of the translation scheme from Horn logic to the DFGs in Section 2. In Section 3, after the semantics of the KTN (Section 3.1) and the detailed algorithm of ELISE (Section 3.2) are described, 'pools' are introduced to select tokens 'locally' in ELISE (Section 3.3). Some convergence experiments of ELISE are conducted in Section 4, and concluding remarks are given in Section 5.

2. KNOWLEDGE REPRESENTATION

2.1 From High-level Petri-net to KTN

Definition 1: Throughout this paper, we take a notation that constants start with an up-

percase letter and variables start with a lowercase letter. *q* is a specific variable for the query. Looking upon a term *f*(*t*1, …, *tn*) as a list expression *f*(*t*1, …, *tn*), we sometimes call a term a **symbol sequence**.

The KTN's basic design can be derived from the high-level Petri-net by making some revisions. Let us consider the following logic program.

Example 1 (Man/Human):

Man(Tom).	(Tom is a man.)	
Human(*x*) ← *Man*(*x*).	(A man is a human.)	(1)
← *Human*(*q*)	(Who is a human?).	

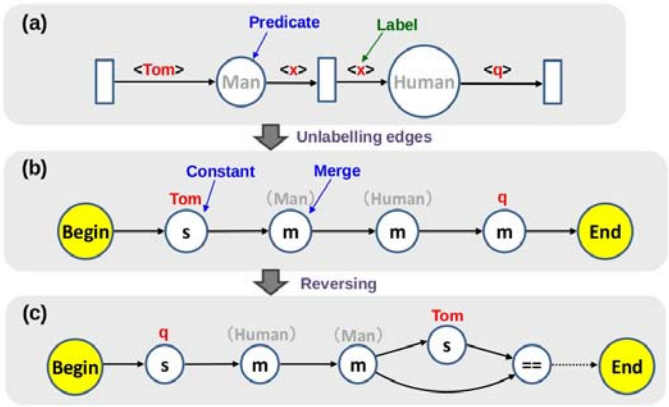

Fig. 1. Derivation of a KTN from a high-level Petri-net for Prog. (1); (a) High-level Petri-net for forward deduction; (b) Data-flow graph for forward deduction; and (c) Data-flow graph for backward deduction.

Fig. 1 (a) shows the high-level Petri-net for forward deduction in Prog. (1). This Petri-net is modified by introducing symbol nodes labeled with 'Tom' and '*q*', changing places into nodes with operation `m`, removing transitions and arc labels, and adding the `B` and `E` nodes at both ends (Fig. 1 (b)). See Table 1 for the meanings of the node operations. Finally, the KTN is derived by reversing the directions of all the edges and introducing an '==' node specifying the binding condition to 'Tom' (Fig. 1 (c)). As compared to the original high-level Petri-net, the KTN's edges are not labeled but its nodes are labeled with operations and terms.

2.2 Transformation from Horn Logic to KTN

If we prepare an appropriate set of node operations (Table 1), we can construct a DFG representing any form of Horn clause. Fig. 2 shows some examples of such transformation, and Figs. 3 to 6 show the general translation rules that define a KTN in an inductive way. The final DFG for a logic program is constructed by making `m` nodes shared between the created subnets.

Table 1. Node operations.

Name	Oper. code	Input num.	*Synchronous/ Asynchronous*	t'	r'	Function
Begin	B	0	(No input)		1	Begin propagation
End	E	1+	Async.			End propagation
Symbol	s	1+	Async.	o	r	Create a symbol token
Apply	a	2	Sync.	$t_0(t_1)$	$\min(r_0, r_1)$	Apply the 0th input function to the 1st input
Merge	m	1+	Async.	t	r	Transfer a token freely
Gate	g	2	Sync.	t_0	$\min(r_0, r_1)$	Transfer the 0th input if the 1st input exists
Combine	∋	2+	Sync.	$(t_0, t_1, \ldots)$	$\min(r_k)$	Combine the arguments
Split	∈	1+	Async.	$t'_k = t_k$	$r'_k = r$	Split the arguments
Equal	==	2	Sync.		$\min(r_0, r_1)$ $\exp(-\kappa(\mathrm{Dis}(x_0, x_1))^2)$	Check if the two inputs are equal
And	&	2+	Sync.		$\min(r_k)$	Logical AND

In the third column, '1+' means that the node can have one or more incoming edges. g, a, ∈, and ∋ are 'choosy' nodes whose incoming edges (and tokens on them) are ordered (numbered 0, 1, …). An 'Asynchronous' node fires every time a token arrives at an incoming edge, and a 'Synchronous' node fires only if operand tokens arrive at all of the incoming edges. The 5th and 6th columns (t' and r') specify the output token's term and reliability produced by the node's firing during forward propagation, respectively. (See Section 3 for the meaning of the 'forward'.) Here, t or t_k is the symbol of the (kth) incoming token, r or r_k is its reliability, t' or t'_k is the term of the (kth) outgoing token, r' or r'_k is its reliability, and o is the node's original (current) symbol sequence. See Eqs. (3) and (4) for the definition of κ and Dis(…), respectively.

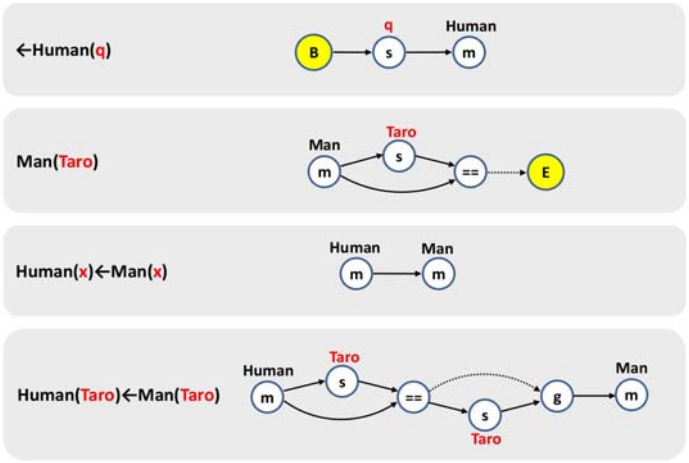

Fig. 2. Examples of the transformation of Horn logic to DFGs. At each row, a logic/clause (on the left side) and a corresponding subgraph (on the right side) are shown. For easier understanding, some 'Merge' nodes (ms) and edges are labeled with predicate names (*Human*, *Man*, *etc.*) and terms (x, y, *wife*(x), *etc.*), but these labels are not actually used for the inference. Because the fifth top clause includes two atoms in the body of the rule, the outgoing edges of the m node in the center of the DFG are marked with a thick black arc denoting 'AND'. This and the lowest rows create new symbol nodes s with z and s with x, respectively. These nodes represent variables that are not determined by tokens from outside. Note that in the lowest two rows, a function symbol *wife* is represented by a symbol node s with *wife*.

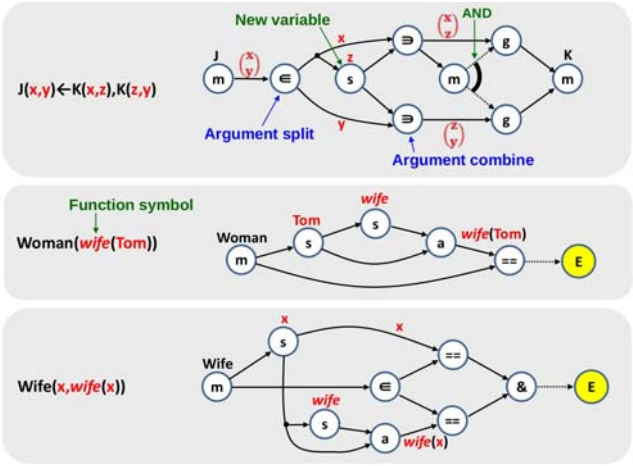

Fig. 2. (Cont'd) Examples of the transformation of Horn logic to DFGs. At each row, a logic/clause (on the left side) and a corresponding subgraph (on the right side) are shown. For easier understanding, some 'Merge' nodes (ms) and edges are labeled with predicate names (*Human*, *Man*, *etc.*) and terms (*x*, *y*, *wife*(*x*), *etc.*), but these labels are not actually used for the inference. Because the fifth top clause includes two atoms in the body of the rule, the outgoing edges of the m node in the center of the DFG are marked with a thick black arc denoting 'AND'. This and the lowest rows create new symbol nodes s with z and s with x, respectively. These nodes represent variables that are not determined by tokens from outside. Note that in the lowest two rows, a function symbol *wife* is represented by a symbol node s with *wife*.

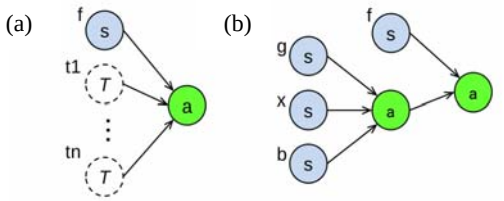

Fig. 3. Translation rule for a term. DFGs for (a) (*t*1, …, *tn*) and (b) *f*(*g*(*x*, *b*)). Here and in the subsequent figures, a dotted node is a 'tentative' node T that is to be replaced with an actual subgraph before finishing the translation. (b) is made by first creating a subnet for *f*(*t*) using a T for the parameter *t*, creating a subnet for *g*(*x*, *b*), then replacing the T with the subnet for *g*(*x*, *b*).

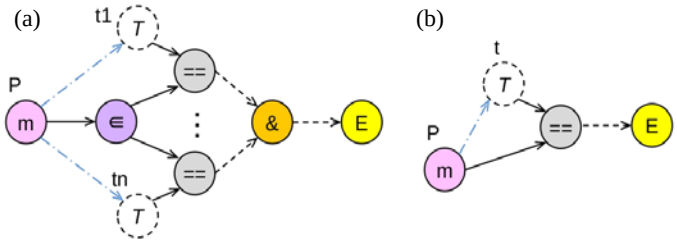

Fig. 4. Translation rule for a fact. DFGs for (a) '*P*(*t*1, …, *tn*).' and (b) '*P*(*t*)'. Here and in subsequent figures, dashed arrows are for 'logical' tokens that convey only reliability. Chain arrows from the 'm' node to the T nodes are actually connected to all the Symbol nodes in the subnets *t*1, …, *tn* to make them fire. All the arguments of a predicate is split by a '∈' node and connected to variable binding node '=='s. The other input edges of the '=='s are taken from subnets for *t*1, …, *tn*. The output edges of the '==' nodes go to a '&' node whose output is finally connected to an 'E' node.

Example 2 (Family relationship):

[Fact] *Wife*(Mary, Paul). (2a)
[Fact] *Child*(Tom, Paul). (2b)
[Rule] *Child*(*x*, *y*) ← *Wife*(*y*, *z*), *Child*(*x*, *z*). (2c)
[Goal] ← *Child*(*q*, Mary). (2d)

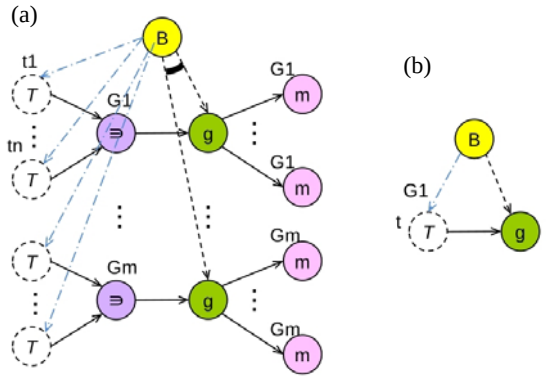

Fig. 5. Translation rule for a goal. DFGs for (a) '← *G*1(*t*1, ..., *tn*), …, *Gm*(…).' and (b) '← *G*1(*t*).'. The thick black arc on the outgoing edges of the node 'B' denotes logical 'AND' between the edges. For each atom in the body, a '∋' node is prepared which receives inputs from tentative nodes for *t*1, …, *tn* and combines them. Though in Fig. 5, 'g's are inserted after the '∋'s, the '∋'s' outputs may go directly to the 'm' nodes for the predicates *G*1, …., *Gm* because the 'g's' 1st inputs come from 'B' which always fires at the beginning of the operations.

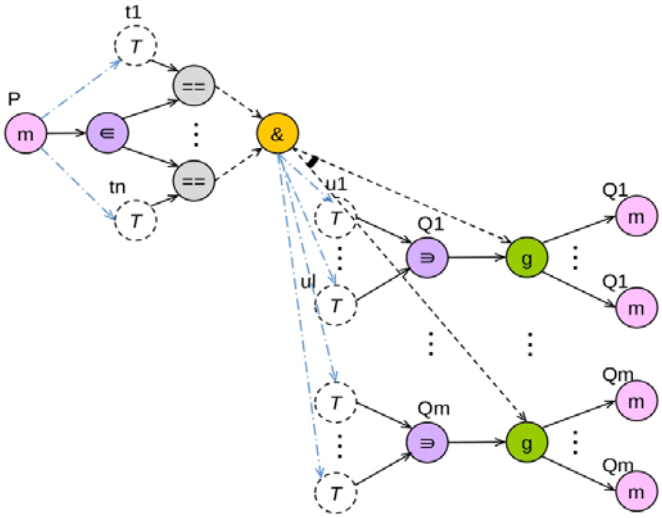

Fig. 6. Translation rule for a rule, namely, DFG for '*P*(*t*1, …, *tn*) ← *Q*1(*u*1, …, *ul*), …, *Qm*(…).'. The network for a rule is obtained by combining the networks for the component fact and goal. The 'g' nodes succeeding the '∋' nodes and Symbol nodes in the subnets for *u*1, …, *ul* receive inputs from the '&' of the fact.

Fig. 7 shows Example 2 and its translation result. After translating the clauses into subnets, they were combined by sharing 'm' nodes (for *Wife* and *Child*) among the subnets. Since the atom *Child* appears at both the head and body of the clauses, the final DFG has a loop.

As shown in Fig. 7, a KTN has the original Horn logic's constants and variables as nodes. The binding conditions between them are explicitly specified by the KTN's topology. Some other translation results for test programs are shown in Table 2 and Fig. 8. Table 2 suggests that the KTN's size (*i.e.*, the numbers of constituent nodes and edges) increases approximately linearly with the clause number in a logic program.

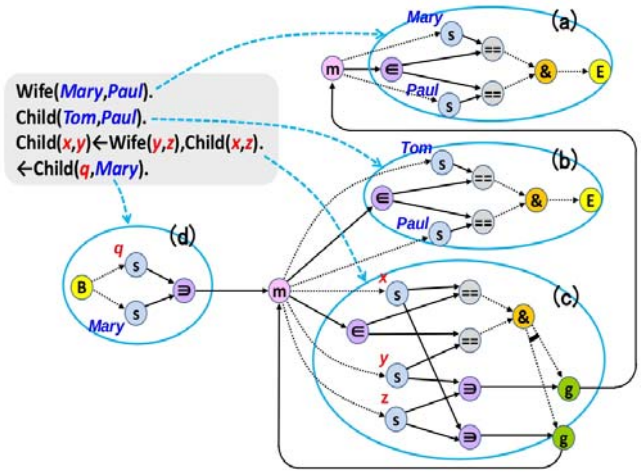

Fig. 7. Transformation from logic program (Example 2) to KTN. The subnets superscripted with (a), (b), (c), and (d) stand for Clauses (2a), (2b), (2c), and (2d), respectively.

Table 2. Translation results from Logic programs to KTNs.

Program name	Clause number	Node number	Edge number
append	3	43	70
perm	5	69	113
intcalc1	5	71	115
reverse	5	73	119
hanoi	5	102	181
fib	8	107	177
greek	19	127	195
intcalc2	13	204	341

The translation program was implemented in Java. It combines all the logical elements (terms, atoms, and clauses) with a parser part to create graphical elements (nodes and edges). 'append' is a logic program for appending lists, 'perm' is a program for calculating permutation, 'intcalc1' is for addition and multiplication of integers (Example 3 in this paper), 'reverse' reverses the order of a list, 'hanoi' is a program solving the Tower of Hanoi, 'fib' calculates Fibonacci numbers, 'greek' defines the relationship between characters in the Greek myths, and 'intcalc2' is a program for such integer calculation as addition, multiplication, Fibonacci numbers, Ackermann Function, and factorial. The second column stands for the number of Horn clauses in the original logic programs, and the third and fourth columns stand for the numbers of nodes and edges in the detailed KTNs, respectively. Concrete expressions of the logic programs can be found in [2, 9, 24].

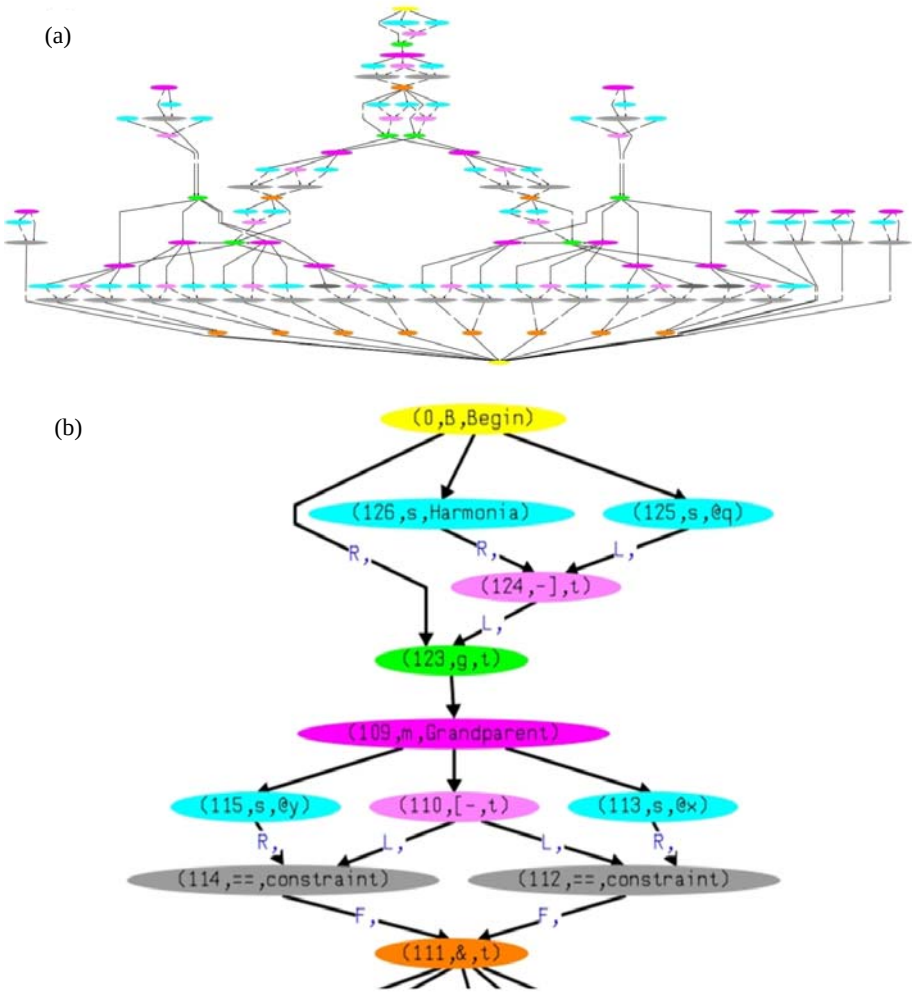

Fig. 8. Translation result for the 'greek' program in Table 2. (a) The general view of the entire KTN, and (b) an enlarged view of the top central part of (a). The first/second/third entry of each node label represents the node number/operation code/node name, respectively. For example, (126, s, Harmonia) stands for a constant node with term 'Harmonia', and (124, -], t) means a 'Combine' (∋) node ('t' has no meaning in this case). The edge labels 'L' and 'R' stand for the first and second incoming edges of a 'choosy' node (g, ∋, or ==), respectively. The graphs are drawn using software named 'aiSee'.

3. DEDUCTIVE INFERENCE

The KTN's semantics is argued in two different levels. The first level deals with truth or falsehood of the network without considering the possibility of finding a solution of unification. The soundness and completeness of the KTN are demonstrated in a conventional way. The second level deals with the unification. ELISE is introduced, and inference ambiguity is represented by the token reliability.

3.1 Soundness and Completeness

Definition 2: An **Exp-DFG** is an acyclic directed DFG created by the expansion of the original DFG (KTN). Its '==' nodes are called **leaves**, and its node with no incoming edges, *i.e.*, B node is called **root**. An Exp-DFG is referred to as TRUE/FALSE if its root is TRUE/FALSE, respectively.

Since the KTN includes a feedback loop in general, we first propagate tokens in the original DFG and expand it, in order to argue the soundness and completeness. A token that propagates for this purpose has the form of (t) or ($t1$, …, tn), where t, $t1$, and so on represent terms. The propagation starts with the firing of the 'Begin' node. When a node fires, the node calculates a new term with a formula in the 5th column of Table 1 and emits it to its outgoing edges.

While the propagation continues, the firing-propagation history is logged in a memory, and after a sufficient number of firing occurs in the DFG, a prototype of an Exp-DFG is created from the history. When a Symbol node with 'o' fires and is expanded several times on account of the feedback loop, different symbols, written for example as o_0, o_1, …, are attached to the expanded nodes.

Then the final Exp-DFG is created from the prototype by the following operations. First, AND relations between edges in the original DFG (shown in Figs. 5 (a) and 6) are copied to the corresponding edges in the Exp-DFG. Second, a new AND relation is inserted between every set of outgoing edges carrying the same term from a node. All the other edges in the Exp-DFG are regarded as OR edges.

Since the Exp-DFG has AND/OR structure, if we specify logical values, TRUE or FALSE, of all the leaves in an Exp-DFG, the logical value of its root can be also determined. By specifying the leaves' truth or falsehood, we are specifying an interpretation, under which all the variables' values are determined through unification (when unifiable).

Proposition 1: (Soundness and Completeness): Let G be a subgraph of the Exp-DFG such that G's root is the root of the Exp-DFG and G does not include an OR junction, and U be unification. Then,

There exists a G that is TRUE under U.
$\Leftrightarrow$ The Exp-DFG is TRUE under U.

***Proof*:** $\Rightarrow$ (soundness): Because G does not include an OR junction, TRUTH of the root of G means that all the nodes in G are TRUE. Since any addition of an AND or OR edge to a TRUE node cannot make the node be FALSE, G's truthiness is conserved to the entire Exp-DFG.

$\Leftarrow$ (completeness): If the Exp-DFG, that is an AND/OR graph, is TRUE, every OR node in the Exp-DFG has at least one TRUE outgoing edge. By cutting all the other edges in all the OR nodes, we can transform the Exp-DFG into a G while conserving the truthiness of the root. ❑

The proposition states that for deductive inference of a KTN, we only have to identify a minimum TRUE subgraph *G* while growing the Exp-DFG. If we cannot find such a subgraph *G* in the current Exp-DFG, we have to propagate tokens in the original DFG further and grow the Exp-DFG. Note that for a general Horn logic program there can be two or more solutions. To obtain them, we have to repeat the above operation until we find an appropriate number of *G*s. The final answers of the deduction are given by the terms for *q* under the *U*s.

3.2 ELISE

Though in the previous subsection, we argued semantics of the KTN only in terms of truth or falsehood of the Exp-DFG, here we extend the truth value to a continuous real value within [0, 1] and represent some ambiguity. From this point of view, we deal with a token in the form of (*t*, *r*) or (*t*1, …, *tn*; *r*) in this subsection. Here, *t*, *t*1, … represent terms, and *r* represents 'reliability'.

Definition 3: A **branch** of the Exp-DFG is a subgraph with just one incoming edge, and its **root** is the terminal node of the incoming edge. A branch is called TRUE/FALSE if its root is TRUE/FALSE, respectively. **Reliability** is a real value within [0, 1] that represents the inter-term consistency between tokens. This also specifies the logical TRUTH of a node at which a token is staying, and finally determines the logical TRUTH of a branch or the Exp-DFG. We call the node/branch/Exp-DFG's logical TRUTH **reliability** as well.

The token reliability *r* is basically the product of 'consistency factors' formulated as

$$\exp(-\kappa(\mathrm{Dis}(u, v))^2), \tag{3}$$

where κ is a predefined constant (selection coefficient) and

$$\mathrm{Dis}(u, v) = \begin{cases} 0 & \text{if } u = v \\ 1 & \text{if } u \neq v \end{cases} \tag{4}$$

is an inter-term distance function. (Although the square of Dis(*u*, *v*) in Eq. (3) is meaningless under Eq. (4), it would become meaningful when we introduce such a more complex distance function as Levenshtein distance [10] in the future.)

As described in the previous section, the Exp-DFG can grow infinitely with an increase in the depth of the backward deduction, but if we are able to cut the Exp-DFG's redundant branches during the expansion, it will narrow down the search space and accelerate the deduction. The token reliability *r* is used not only to select tokens for unification but also to identify such branches. *r* is calculated by our newly developed unification algorithm named ELISE (ELiminating Inconsistency by SElection).

ELISE is basically an agent-based evolutionary method to solve simultaneous equations (binding conditions) buried in the network. The detailed algorithm of ELISE is as follows:

(1) [Initial Setting] Substitute the Exp-DFG's variable nodes with initial constant

symbols chosen randomly.

(2) [Forward Propagation] Based on the variable nodes' original (current) terms, create tokens with term-reliability pairs (t, r)s and propagate them towards the 'End' node. At each node, the output token (t', r') is calculated using the formulas in Table 1.

(3) [Backward Propagation] Create a 'correction' token with a term-reliability pair at the 'End' node and propagate it backward until tokens arrive at the variable nodes. During this process, the reliability value is basically conserved at each node from an outgoing token to incoming tokens, except that the conflict occurs between them (see below for the conflict processing). The term is also conserved, but in such a node as '==' and 'a', new correction terms are produced so that the input/output tokens might become consistent with each other around the node. (An == node creates correction terms t_1 and t_0 on the 0th and 1st incoming edges, respectively, and an a node that receives the correction term $\hat{t}'$ produces the correction terms $\hat{t}'_0$ and $\hat{t}_1$ that satisf $\hat{t}'_0(t_1) = \hat{t}'$ and $t_0(\hat{t}_1) = \hat{t}'$, respectively. Hereafter, $\hat{x}$ represents the correction of x.) During the backward propagation, if conflict occurs between the correction terms at a node, token selection takes place. Based on r, tokens are selected or eliminated according to their logical relationship. Tokens are randomly selected among 'AND'-tokens (*i.e.*, tokens on edges connected with AND operations), and tokens are selected in proportion to r among 'OR'-tokens (*i.e.*, tokens on edges connected with OR operations). The random selection among AND-tokens ensures that the outgoing AND edges have equal chance to provide correction terms. (Steps 2 and 3 are exemplified in Fig. 9.)

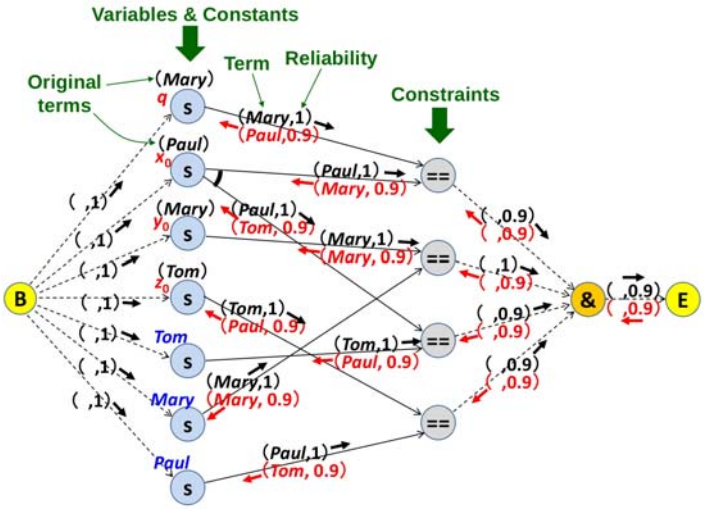

Fig. 9. Artificial DFG for the binding conditions $q = x_0$, Mary = y_0, x_0 = Tom, and z_0 = Paul. The conditions were extracted from the depth-two Exp-DFG of Fig. 7. (Here, 'depth' is the number of times that tokens circulate around the loop.) Tokens propagated in Steps 2 and 3 are depicted in black and red, respectively. At each '==' node, the correction term for an operand is calculated by keeping the other operands intact. For example, the top '==' node which received Mary and Paul in the forward propagation produces the correction terms, Paul and Mary. Since the variable node x_0 is an AND node, the node chooses the correction term randomly out of the two arriving tokens, before going to Step 4.

(4) [Selection (Evolution)] From a correction token arriving at a variable node, the consistency factor is calculated between the original and correction terms by Eq. (3), and the reliability is multiplied by the factor. Using this reliability, tokens are selected globally or locally (see below), and the variable nodes' original terms are updated. If the reliability of the final tokens is sufficiently high, namely, the variables have self-consistent terms, stop the procedure and answer the query node's term. Otherwise, go to Step 2.

In a typical Exp-DFG, two consistency factors are evaluated during one return propagation − one at a constraint node '==' at the end of the forward propagation (see r''s formula in Table 1 and another at a variable node at the end of the backward propagation – and they are accumulated (multiplied) at reliability r. ELISE which selects tokens with higher reliability gradually gets rid of the conflict between token terms and finally makes the variables have a self-consistent solution. In the paper, we prepare two different selection schemes (global and local) and evaluate their performance.

When we take the global selection scheme, we calculate the 'correction probability'

$$p_i = \hat{r}'_i / (\sum_{i'} \hat{r}'_{i'}) \tag{5}$$

from the final token reliability $\{\hat{r}'_i\}$ (where i is the variable number) after one return propagation, and make the original term of the ith variable node be replaced with the correction term $\hat{t}'_i$ in the probability of p_i. Eq. (5)'s normalization sum is taken for all the variables in the network. In this way, in the global selection scheme, a single variable correction happens at a time in the entire network on the average. This helps avoid the conflict between correction terms and makes ELISE approach slowly but steadily to a solution.

Definition 4: We call an Exp-DFG's branch whose reliability is low a **false branch**, and a branch whose reliability is not yet determined an **uncertain branches**.

In Section 3.1, we argued that the truth or falsehood of the Exp-DFG or subgraph G is determined by the logical values of its leaves. Likewise, the reliability of the Exp-DFG or a branch is calculated from the reliability values of its leaves, which is accomplished by the token propagation in ELISE. ELISE calculates rs of leaves at the end of Step 2, carries them backward during Step 3, and finally evaluates the reliability of its root. During this process, we are able to identify false or uncertain branches: a false branch always produces low-reliability tokens backward, and an uncertain branch does not produce tokens backward. By cutting false branches and growing the Exp-DFG at the tip of uncertain branches, we can minimize the search of Exp-DFGs and accelerate deduction. A similar idea is also found in [5], where the search space is pruned based on probabilities.

3.3 Token Pool

When we take the local selection scheme, we no longer use Eq. (5) to select correction tokens among variables. We prepare 'token pools' at the variable nodes instead, put all the correction tokens from Step 3 into the pools, and make evolution happen in the pools. This makes the operations of ELISE completely local, and at the same time, accelerates its convergence.

Concrete operations of a token pool is as follows:

(1) [Preparation] Prepare an empty token pool at a variable node whose size is limited to N.

(2) [Push & Reproduction] If a correction token $(\hat{t}'_i, \hat{r}'_i)$, arrives at the variable node by Step 3 of the previous subsection, it is pushed onto the pool together with $s\hat{r}'_i$ copies of it. Here, s is a constant named 'selection coefficient'.

(3) [Selection] Tokens are chosen in inverse proportion to r^{α} and are eliminated until the pool size becomes less than N.

(4) [Mutation] At a constant rate u, tokens are randomly chosen and their terms are replaced with arbitrary constants.

(5) [Pop] A token is randomly chosen (popped) and eliminated out of the pool, and its term is substituted for the node's next original term. Go to Step 2.

4. CONVERGENCE EXPERIMENTS OF ELISE

4.1 Problems

To examine the convergence properties of ELISE, numerical experiments are conducted using the following logic program:

Example 3 (Arithmetic operations):

$$Add(0, x, x). \tag{6a}$$
$$Mul(0, y, 0). \tag{6b}$$
$$Add(S(u), v, S(w)) \leftarrow Add(u, v, w). \tag{6c}$$
$$Mul(S(e), f, h) \leftarrow Mul(e, f, g), Add(g, f, h). \tag{6d}$$

$S(\ )$ is a successor function.

In addition to these facts and rules, we prepare a goal clause in the form of

$$\leftarrow Mul(S(S(\ldots S(0)\ldots)), S(0), q), \tag{6e}$$

extract the binding conditions in the depth-D Exp-DFG of the whole program, and construct an artificial DFG with the same structure as that of Fig. 9 from the binding conditions. Hereafter, with M (defined as the number of Ss included in the first argument of Eq. (6e)), we designate a logic program represented by Eqs. 6 (a)-(e) with M Ss as 'Problem-M'. If $D \leq M$, the extracted binding conditions are nonunifiable, but if $D > M$, the conditions are unifiable. From preliminary studies, we obtained the optimal parameters as follows: $\kappa = 0.1$, $N = 3$, $s = 5$, $\alpha = 1$, and $u = 0$.

4.2 Convergence Ability

Fig. 10 shows two example artificial DFGs (nonunifiable and unifiable) for Problem-2, and Fig. 11 shows the results for them. We see from Figs. 11 (a1) and (a2) that with the nonunifiable constraint set, the number of unsatisfied constraints stays at one at larger iteration numbers, meaning that a self-consistent solution cannot be found no matter whether we may use global or local selection scheme. In this case, the token's final reliability is less than one (typically, $\hat{r} = 0.8$~0.9).

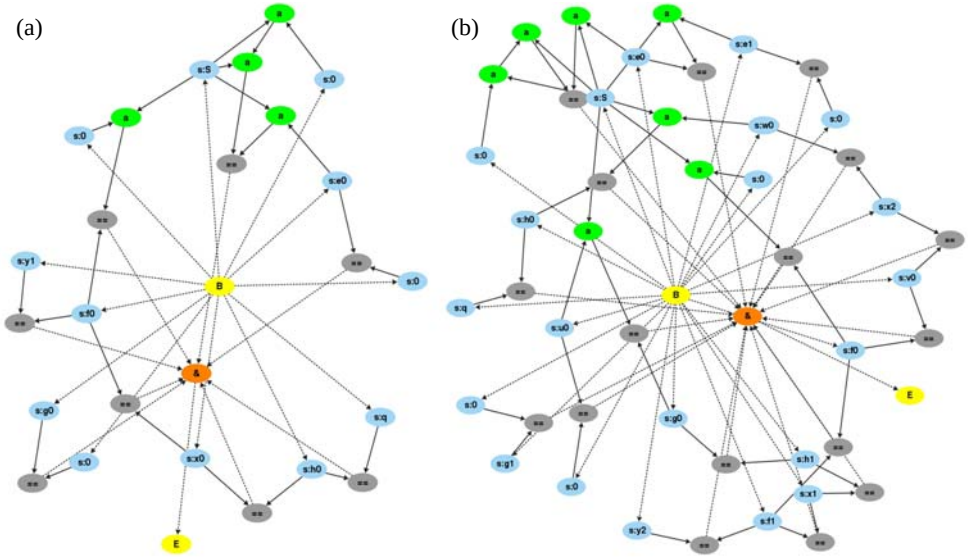

Fig. 10. Artificial DFGs created from the binding condition sets in the (a) depth-2 and (b) depth-3 Exp-DFGs (a) represents 8 nonunifiable binding conditions on 7 variables, and (b) represents 17 unifiable binding conditions on 15 variables. The symbol nodes (lightblue) are classified into function symbols, variables, or constants. An Apply node 'a' (lightgreen) always takes the 0th input from a function node.

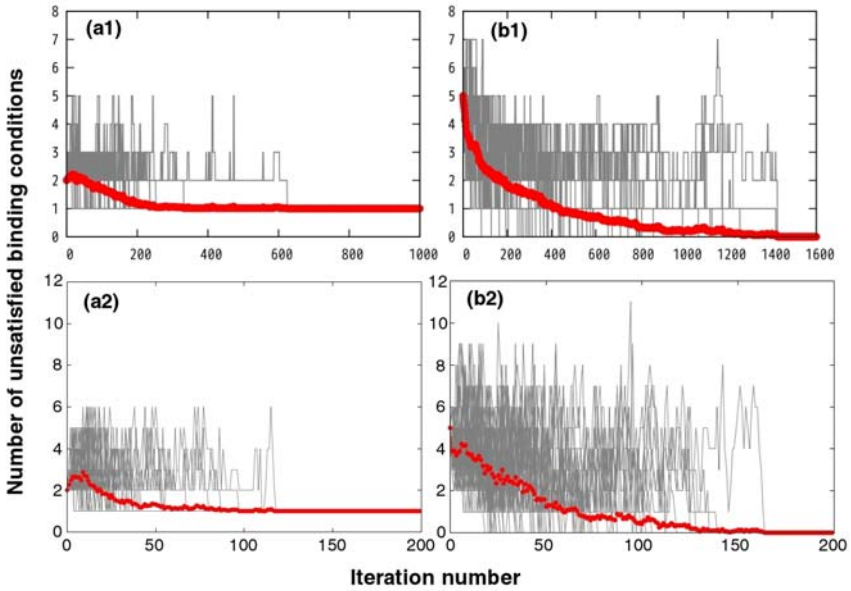

Fig. 11. ELISE's results for Figs. 10 (a1) and (b1) are the results by the global token selection scheme for Figs. 10 (a) and (b), respectively, and (a2) and (b2) are the results by the local selection scheme for Figs. 10 (a) and (b), respectively. The thin gray lines are the results for fifty runs using different initial symbols and different random number sequences, and a thick red line is their average.

With the unifiable constraint set (Figs. 11 (b1) and (b2)), on the other hand, the number of unsatisfied constraints converges to zero for all the fifty runs, which means that ELISE succeeds in finding a solution regardless of the initial constant symbols substituted. The solution gives the right answer $q = S^2 0 = S(S(0))$ with the highest reliability $r = 1.0$ in this case.

In both nonunifiable and unifiable cases, the average of fifty runs monotonously decreases over time, from which it is concluded that ELISE has a tendency to minimize conflict between token variables and make their terms converge.

4.3 Scaling Properties

We prepare artificial DFGs for unifiable binding conditions for Problems-0 to –7 and apply ELISE to them. According to the results shown in Table 3 and Fig. 12, we can say that the convergence speed of ELISE with the local selection scheme is much faster than that with the global selection scheme. From the regression lines in Fig. 12, we see

Table 3. Average convergence time.

M	*D*(depth)	Num. of variables	Num. of constraints	Convergence time	
				global	local
0	1	2	2	3.16	2.26
1	2	7	8	139.44	28.10
2	3	15	17	576.22	60.00
3	4	26	29	1760.88	116.66
4	5	40	44	5564.18	186.52
5	6	57	62	9626.60	237.66
6	7	77	83	20418.92	354.04
7	8	100	107	35928.52	496.42

M is the number of *S*s included in the first argument of Eq. 6 (e), and *D* is the depth of an Exp-DFG for a program represented by Eqs. 6 (a)-(e). The third and fourth columns are the variable and constraint number included in the binding conditions in the Exp-DFGs, respectively, and the fifth and sixth columns represent the average iteration number of fifty runs until the convergence of ELISE using the global and local selection schemes, respectively.

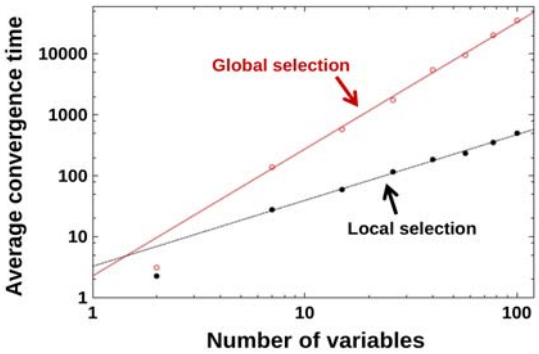

Fig. 12. The plot of Table 3. The convergence time by the global and local selection (fifth and sixth columns) is plotted as a function of the number of variables (third column) in red circles and black dotts, respectively. The red and black dashed lines are the regression lines which represent 2.3 × [Number of variables]$^{2.08}$ and 3.3 × [Number of variables]$^{1.08}$, respectively.

that the convergence time by the local token selection scheme increases almost linearly with the number of variables. This suggests that ELISE using token pools is applicable to larger inference problems without the risk of convergence time explosion.

5. CONCLUSION

A general transformation scheme from a Horn logic program to a data-flow graph (KTN) was presented. Using an appropriate set of node operations, the KTN describes constants, variables, and function symbols explicitly, and the binding conditions for the variables are specified by topological structure of the network. After arguing a sematic aspect of the KTN, a graph-based method to solve unification, ELISE, was presented, and its semantic meanings and convergence properties were described with some numerical results.

Future research agendas of the KTN are as follows:

- Full implementation of the KTN's deduction scheme and experimental verification of ELISE on the KTN.
- Incorporating a learning scheme that revises the KTN's ground terms or topological structure.
- Devising a method to transform the modified structure of the KTN back into predicate logic. This would be useful for humans to interpret the KTN's knowledge accumulated through the learning.

ACKNOWLEDGEMENTS

We appreciate valuable comments by Hiromu Hayashi, Auditor of NICT, that led us to invent the KTN.

REFERENCES

1. C. Baral, M. Gelfond, and N. Rushton, "Probabilistic reasoning with answer sets," *Theory and Practice of Logic Programming*, Vol. 9, 2009, pp. 57-144.
2. I. Bratko, *Prolog Programming for Artificial Intelligence*, 3rd ed., Addison-Wesley, MA, 2001.
3. R. D. S. Braz, E. Amir, and D. Roth, "Lifted first-order probabilistic inference," in *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 2005, pp. 1319-1325.
4. J. B. Dennis, "Data flow supercomputer," *Computer*, Vol. 13, 1980, pp. 48-56.
5. L. de Raedt, A. Kimmig, and H. Toivonen, "ProbLog: a probabilistic prolog and its application in link discovery," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007, pp. 2462-2467.
6. M. Jackman and C. Pavelin, "Conceptual graphs," in G. A. Ringland and D. A. Duce eds., *Knowledge Representation – An Introduction*, Research Studies Press, Ltd., Chapter 7, 1988, pp. 161-174.
7. J. Jeffrey, J. Lobo, and T. Murata, "A high-level Petri net for goal-directed semantics of Horn clause logic," *IEEE Transactions on Knowledge and Data Engineering*,

Vol. 8, 1996, pp. 241-259.
8. S. Kok and P. Domingos, "Learning the structure of Markov logic networks," in *Proceedings of the 22nd International Conference on Machine Learning*, 2005, pp. 441-448.
9. R. Kowalski, *Logic for Problem Solving*, Elsevier Science Ltd., Amsterdam, The Netherlands, 1979.
10. V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, Vol. 10, 1966, p. 707.
11. V. Lifschitz, "Answer set programming and plan generation," *Artificial Intelligence*, Vol. 138, 2002, pp. 39-54.
12. T. Murata, "Petri nets: properties, analysis and applications," in *Proceedings of the IEEE*, Vol. 77, 1989, pp. 541-580.
13. M. L. Oarg, S. I. Abson, and P. Gupta, "Petri-nets for logic-based deductions," Technical Report, Electrical Engineering Department, Indian Institute of Technology, 1987.
14. G. Peterka and T. Murata, "Proof procedure and answer extraction in Petri net model of logic programs," *IEEE Transactions on Software Engineering*, Vol. 15, 1989, pp. 209-217.
15. D. Poole, "First-order probabilistic inference," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003, pp. 985-991.
16. M. Quillian, "Semantic memory," in M. Minsky (ed.), *Semantic Information Processing*, MIT Press, MA, 1968, pp. 216-270.
17. D. M. Randal, "Semantic networks," in G. A. Ringland and D. A. Duce, eds., *Knowledge Representation – An Introduction*, Research Studies Press, Ltd., 1988, Chapter 3, pp. 45-80.
18. M. Richardson and P. Domingos, "Markov logic networks," *Machine Learning*, Vol. 62, 2006, pp. 107-136.
19. D. D. Roberts, *The Existential Graphs of Charles S. Peirce (Approaches to Semiotics)*, Mouton de Gruyter, 1973.
20. T. Sato and Y. Kameya, "Parameter learning of logic programs for symbolic statistical modeling," *Journal of Artificial Intelligence Research*, Vol. 15, 2001, pp. 391-454.
21. J. A. Sharp, ed., *Data Flow Computing: Theory and Practice*, Ablex Publishing Corp., Norwood, NJ, 1992.
22. P. Singla and P. Domingos, "Lifted first-order belief propagation," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, 2008, pp. 1094-1099.
23. J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, MA, 1984.
24. L. Sterling and E. Shapiro, *The Art of Prolog, Advanced Programming Techniques*, MIT Press, Cambridge, 1986.
25. H. Suzuki, H. Ohsaki, and H. Sawai, "A network-based computational model with learning," in *Proceedings of the 9th International Conference on Unconventional Computation*, LNCS 6079, 2010, p. 193.
26. H. Suzuki, H. Ohsaki, and H. Sawai, "Algorithmically transitive network: a self-organizing data-flow network with learning," in *Proceedings of International Conference on Bio-Inspired Models of Network, Information, and Computing Systems* 2010, LNICST, Vol. 87, 2012, pp. 59-73.

27. H. Suzuki, M. Yoshida, and H. Sawai, "Knowledge transitive network: a data-flow network for backward deduction," in *Proceedings of the 8th International Conference on Complex Systems*, 2011, pp. 357-358.
28. H. Suzuki, M. Yoshida, and H. Sawai, "A proposal of data-flow network for deductive inference," in *The Special Interest Group Notes of the Japanese Society for Artificial Intelligence: Fundamental Problems of Artificial Intelligence, SIG-FPAI-B102*, 2011, pp. 1-7.
29. H. Suzuki, M. Yoshida, and H. Sawai, "A data-flow network that represents first-order logic for inference," in *Proceedings of Conference on Technologies and Applications of Artificial Intelligence*, 2012, pp. 211-218.
30. G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. de Raedt, "Lifted probabilistic inference by first-order knowledge compilation," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011, pp. 2178-2185.

Hideaki Suzuki (鈴木秀明) received his Ph.D. degree in Biological Science from Kyushu University, Japan in 1998, and in Informatics from Kyoto University, Japan in 2004. He is currently a Senior Researcher in Center for Information and Neural Networks (CiNet), National Institute of Information and Communications Technology (NICT), Japan. His main research interests are in brain-inspired computational/learning/reasoning systems with network architecture. He is a member of Society of Instrument and Control Engineers (SICE), Japan.

Mikio Yoshida (吉田幹) received his Master degree from Faculty of Engineering, Kyoto University, Japan in 1983. After working for IBM Japan and NS Solutions Corp., he established BBR Inc. and PIAX Inc. in 2002 and 2008, respectively. Now he is a Director and Chief Technology Officer of both companies. His main research interests are in logic programming and distributed computing. He is a member of the Information Processing Society of Japan (IPSJ) and Japanese Society for Artificial Intelligence (JSAI).

Hidefumi Sawai (澤井秀文) is currently a Managing Director at International Affairs Department, National Institute of Information and Communications Technology, Japan. He was an invited researcher of Carnegie Mellon University, USA in 1989 and 1990, and a Professor of Graduate School of Kobe University (joint appointment) from 1999 to 2010. His research interests include intelligent information processing inspired by brain function and biological evolution: neural networks, evolutionary computation and complex networks.