International Journal of Innovative Computing, Information and Control
Volume **18**, Number **2**, April **2022**

 ISSN 1349-4198
pp. 361–376

GENERATION OF EQUIVALENT TRANSFORMATION RULES BY META COMPUTATION

Kiyoshi Akama[1] and Ekawit Nantajeewarawat[2,*]

[1]Information Initiative Center
Hokkaido University
Kita 11, Nishi 5, Kita-ku, Sapporo, Hokkaido 060-0811, Japan
akama@iic.hokudai.ac.jp

[2]School of Information, Computer and Communication Technology
Sirindhorn International Institute of Technology
Thammasat University
99 Moo 18, Km. 41 on Paholyothin Highway, Khlong Luang, Pathum Thani 12120, Thailand
*Corresponding author: ekawit@siit.tu.ac.th

Received September 2021; revised January 2022

Abstract. *In the "Equivalent Transformation model" (ET model), computation is regarded as equivalent transformation (ET) of declarative descriptions, and a program consists of ET rules and a control description. Programs may be improved by repeated adoption of new ET rules, and finally an efficient program may be obtained, while the correctness of computation is guaranteed. We have introduced a class of rewriting rules consisting of meta atoms. Meta clauses and meta descriptions consist of meta atoms. A meta rule is a rule for transforming one meta description into another meta description. A meta computation consists of repeated application of meta rules to a meta description, by which we obtain a sequence of meta descriptions. A meta computation determines a correct rewriting rule, i.e., an ET rule. This defines a method of generation of ET rules by meta computation. Both splitting and non-splitting rewriting rules are obtained. We can generate rules to solve problems on any background knowledge represented by a set of arbitrary clauses, not only definite clauses but also non-definite clauses.*
Keywords: Query-answering problem, Model-intersection problem, Equivalent transformation rule, Rule generation, Meta computation

1. Introduction.

1.1. ET structures and rule generation.

Formalization of problems as model intersection problems (MI problems) is of fundamental importance for establishing a general solution method for solving all deductive problems on first-order formulas. To solve an MI problem, equivalent transformation rules (ET rules) are used. The ET-based theory has been successfully applied to proof problems and query-answering problems (QA problems) [1, 2, 3, 4, 5]. In the ET model, computation is regarded as equivalent transformation of declarative descriptions, and a program consists of ET rules and a control description.

1.2. Previous research.

A theory of meta computation was proposed on ET-based computation of QA problems [6, 7, 8]. A typical program synthesis method, called the squeeze method [9], has been developed, where program transformation and rule generation are combined into one computation.

DOI: 10.24507/ijicic.18.02.361

ET rule generation from a logical formula, called a logical equivalence, has been introduced [10, 11], and a correctness proof of a class of logical equivalences by ET-based bidirectional search was developed for rule generation [12].

1.3. **Objectives.** We have proposed a method of rule generation based on meta computation by repeated application of meta rules. This theory is an extension of the previous rule generation theory, i.e., the problem class is extended from the one on definite clauses to the one on arbitrary clauses, where rewriting rules cover from single-head rules to multi-head rules.

1.4. **Outline of the theory.** An outline of the theory is as follows. We extend the previous theory by changing the underlying logic and generalize the concepts around meta descriptions and meta rules. A meta rule is a rule for transforming one meta description into another meta description. A meta computation consists of repeated application of meta rules to a meta description, by which we obtain a sequence of meta descriptions. The first and the last meta descriptions of the obtained sequence give a pair of meta descriptions. The pair of meta descriptions determines a partial mapping to transform a description into a description. A rewriting rule is obtained as a partial mapping to transform a description into a description.

1.5. **Rule generation for increasing solvability.** Rule generation is fundamental for solving problems since it produces programs from a given problem. SLD resolution for QA problems can be regarded as a solver that uses only the general unfolding rules. By generating various, less general and more efficient rules other than the general unfolding rules, we can solve more problems in shorter time. Rule generation increases solvability even if a small problem is considered [13]. Greater increase of solvability is expected for larger and practical problems.

1.6. **Organization.** Section 2 explains problem solving by equivalent transformation. Section 3 defines a class of declarative descriptions and introduces rule generation problems. Section 4 defines meta atoms, meta clauses, and meta descriptions, which are instantiated into atoms, clauses, and descriptions, respectively. Section 5 gives a syntax of rewriting rules by using meta atoms, where two kinds of variables play an important role. Section 6 defines a syntax of meta rules, and the instantiation of meta rules. Section 7 discusses meta computation, and generates splitting/non-splitting rewriting rules by meta computation. Section 8 shows that the current theory is an extension of the previous theory of meta computation on definite clauses. Section 9 concludes the paper.

2. **Problem Solving by ET Rules.** Problem solving by equivalent transformation is explained.

2.1. **ET computation.** Many problems can be solved by transforming problem descriptions equivalently into simpler forms. For example, let D be a set consisting of the following definite clauses, where *eq* means equality, and *app* and *rev* mean *append* and *reverse*, respectively:

$eq(X, X) \leftarrow.$
$rev([\,], [\,]) \leftarrow .$
$rev([A|X], Y) \leftarrow rev(X, Z), app(Z, [A], Y).$
$app([\,], Y, Y) \leftarrow.$
$app([A|X], Y, [A|Z]) \leftarrow app(X, Y, Z).$

We consider two problems, $\langle D, q_1\rangle$ and $\langle D, q_2\rangle$, where the first query q_1 is $rev([1,2], X)$ and the second one q_2 is $rev(X, [1,2])$. When given the first query $q_1 = rev([1,2], X)$, we first make a set of definite clauses:

$$Cs_1 = D \cup \{ans(X) \leftarrow rev([1,2], X)\}.$$

We use the following rules. Let R_1 be the set of these rules. The syntax of rewriting rules is explained in [13].

$eq(X, Y) \Rightarrow \{X = Y\}$.
$rev([\,], Y) \Rightarrow eq(Y, [\,])$.
$rev([A|X], Y) \Rightarrow rev(X, Z), app(Z, [A], Y)$.
$app([\,], Y, Z) \Rightarrow eq(Y, Z)$.
$app([A|X], Y, W) \Rightarrow eq(W, [A|Z]), app(X, Y, Z)$.

Then, Cs_1 is transformed equivalently by R_1 as follows. Only the change of the ans clause in Cs_1 is shown since other clauses in Cs_1 are kept unchanged. Selected atoms are underlined in the computation sequence.

$ans(X) \leftarrow \underline{rev([1,2], X)}$.
$ans(X) \leftarrow \underline{rev([2], Y)}, app(Y, [1], X)$.
$ans(X) \leftarrow \underline{rev([\,], Y)}, app(Y, [2], Z), app(Z, [1], X)$.
$ans(X) \leftarrow \underline{eq(Y, [\,])}, app(Y, [2], Z), app(Z, [1], X)$.
$ans(X) \leftarrow \underline{app([\,], [2], Z)}, app(Z, [1], X)$.
$ans(X) \leftarrow \underline{eq([2], Z)}, app(Z, [1], X)$.
$ans(X) \leftarrow \underline{app([2], [1], X)}$.
$ans(X) \leftarrow \underline{eq(X, [A|Y])}, app([\,], [1], Y)$.
$ans([2|Y]) \leftarrow \underline{app([\,], [1], Y)}$.
$ans([2|Y]) \leftarrow \underline{eq([1], Y)}$.
$ans([2, 1]) \leftarrow$.

It follows that by the rules in R_1, this problem $\langle D, q_1\rangle$ is easily solved.

2.2. **Problem solving by new ET rules.** The rule set R_1 cannot solve the second problem $\langle D, q_2\rangle$ since no rule is applicable to the ans clause in Cs_2, where

$$Cs_2 = D \cup \{ans(X) \leftarrow rev(X, [1,2])\}.$$

To solve the second problem, we consider the following rule set, which is named R_2:

$rev(X, [\,]) \Rightarrow eq(X, [\,])$.
$rev(X, [A|Y]) \Rightarrow eq(X, [C|W]),\ rev(W, Z), app(Z, [C], [A|Y])$.
$app(X, [E], [A]) \Rightarrow eq(X, [\,]), eq(E, A)$.
$app(X, [E], [A, B|Z]) \Rightarrow eq(X, [A|W]),\ app(W, [E], [B|Z])$.

Then the second problem can be solved by $R_1 \cup R_2$. The process of answering the query $q_2 = rev(X, [1,2])$ is

1 $ans(X) \leftarrow \underline{rev(X, [1,2])}$.
2 $ans(X) \leftarrow \underline{eq(X, [A|B])}, rev(B, C), app(C, [A], [1,2])$.
3 $ans([A|B]) \leftarrow rev(B, C), \underline{app(C, [A], [1,2])}$.
4 $ans([A|B]) \leftarrow rev(B, C), \underline{eq(C, [1|D])}, app(D, [A], [2])$.
5 $ans([A|B]) \leftarrow rev(B, [1|D]), \underline{app(D, [A], [2])}$.
6 $ans([A|B]) \leftarrow rev(B, [1|D]), \underline{eq(D, [\,])}, eq(A, 2)$.
7 $ans([A|B]) \leftarrow rev(B, [1]), \underline{eq(A, 2)}$.
8 $ans([2|B]) \leftarrow \underline{rev(B, [1])}$.
9 $ans([2|B]) \leftarrow \underline{eq(B, [F|G])}, rev(G, H), app(H, [F], [1])$.

10 $ans([2,F|G]) \leftarrow rev(G,H), \underline{app(H,[F],[1])}$.
11 $ans([2,F|G]) \leftarrow rev(G,H), \underline{eq(H,[\,])}, eq(F,1)$.
12 $ans([2,F|G]) \leftarrow rev(G,[\,]), \underline{eq(F,1)}$.
13 $ans([2,1|G]) \leftarrow \underline{rev(G,[\,])}$.
14 $ans([2,1|G]) \leftarrow \underline{eq(G,[\,])}$.
15 $ans([2,1]) \leftarrow$.

2.3. **ET rule generation at each computation step.** Since the second problem cannot be solved by the rule set R_1, we need to generate new rules such as the ones in R_2. We consider a method of rule generation at each computation step. To obtain the computation process above, each rule in R_2 should be generated at the following steps:

- at the 1st step with an atom $rev(X,[1,2])$ in the body
  $rev(X,[A|Y]) \Rightarrow eq(X,[C|W]), rev(W,Z), app(Z,[C],[A|Y])$;
- at the 3rd step with an atom $app(X,[A],[1,2])$ in the body
  $app(X,[E],[A,B|Z]) \Rightarrow eq(X,[A|W]), app(W,[E],[B|Z])$;
- at the 5th step with an atom $app(X,[A],[2])$ in the body
  $app(X,[E],[A]) \Rightarrow eq(X,[\,]), eq(E,A)$; and
- at the 13th step with an atom $rev(X,[\,])$ in the body
  $rev(X,[\,]) \Rightarrow eq(X,[\,])$.

Each of these rules can be obtained by the meta computation that will be proposed in Section 7.

2.4. **Generation of rewriting rules by meta computation.** We will propose a method of generating ET rules by meta computation as follows.

- Rewriting rules are introduced conceptually (Section 3.2) and syntactically (Section 5). Rewriting rule (ET rule) generation is defined based on the concept of declarative description (Section 3).
- The concept of meta description is introduced (Section 4). The concept of meta rule is introduced (Section 6). A meta rule is a rule for transforming one meta description into another meta description.
- A meta computation (Section 7) consists of repeated application of meta rules to a meta description, by which we obtain a sequence of meta descriptions. The first and the last meta descriptions of the obtained sequence give a pair of meta descriptions. A rewriting rule is obtained from a pair of meta descriptions.

3. **Declarative Descriptions and Generation of ET Rules.** We introduce a class of separated descriptions, and define rule generation.

3.1. **Declarative descriptions and separated descriptions.** A set of clauses is called a *declarative description* or simply a *description*. Let $\mathcal{A}$ be the set of all user-defined atoms. Let $\mathcal{S}$ be the set of all specializations (substitutions). Let A be a subset of $\mathcal{A}$. A is closed iff $a\theta \in A$ for any atom $a \in A$ and any specialization $\theta \in \mathcal{S}$.

Definition 3.1. *Assume that A_1 and A_2 are closed subsets of $\mathcal{A}$. C is a clause from A_1 to A_2 iff*

1) C is a clause,
2) all atoms in the right-hand side of C are elements of A_1, and
3) all atoms in the left-hand side of C are elements of A_2.

The set of all declarative descriptions consisting only of clauses from A_1 to A_2 is denoted by $Dscr(A_1, A_2)$.

Definition 3.2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be mutually disjoint closed subsets of $\mathcal{A}$. A declarative description Cs is called a separated description iff it satisfies the following conditions:*

1) $Cs = D \cup Q$,
2) D is a clause set in $Dscr(\mathcal{A}_1, \mathcal{A}_1)$, and
3) Q is a clause set in $Dscr(\mathcal{A}_1, \mathcal{A}_2)$.

D is called background knowledge, and Q a set of query clauses. Given D in $Dscr(\mathcal{A}_1, \mathcal{A}_1)$, the set of all separated descriptions that have D as background knowledge is denoted by $Dscr(D, \mathcal{A}_1, \mathcal{A}_2)$.

3.2. **Rewriting rules and ET rules.** Assume that X is the power set of the set of all clauses. A rewriting rule r on the set X is a partial mapping from X to X, i.e.,

$$r \in PartialMap(X, X).$$

The set of all models of a formula F is denoted by $Models(F)$. The intersection of all models of F is denoted by $\bigcap Models(F)$.

Definition 3.3. *A rewriting rule r is model-preserving iff if $\langle Cs, Cs' \rangle \in r$, then $Models(Cs) = Models(Cs')$.*

Definition 3.4. *A rewriting rule r is model-intersection preserving iff if $\langle Cs, Cs' \rangle \in r$, then $\bigcap Models(Cs) = \bigcap Models(Cs')$.*

Obviously, if r is model-preserving, then r is model-intersection preserving.

Definition 3.5. *A rewriting rule r is an ET rule iff r is model-intersection preserving.*

3.3. **Rule generation problem on separated descriptions.** Assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ are mutually disjoint closed subsets of $\mathcal{A}$. Assume that D is a clause set in $Dscr(\mathcal{A}_1, \mathcal{A}_1)$. The objective of the paper is to develop a method of rule generation with the background knowledge D of general clauses. We want to generate rules to transform $Cs_1 = D \cup Q \cup \{C_1\}$ into $Cs_2 = D \cup Q \cup \{C'_1, C'_2, \ldots, C'_n\}$ equivalently for any declarative description $Q \in Dscr(\mathcal{A}_1, \mathcal{A}_2)$, where equivalence means that $\bigcap Models(Cs_1) = \bigcap Models(Cs_2)$. Rules considered in this paper do not change D and Q, and transform a clause C_1 from $\mathcal{A}_1$ to $\mathcal{A}_2$ to a finite clause set $\{C'_1, C'_2, \ldots, C'_n\}$ in $Dscr(\mathcal{A}_1, \mathcal{A}_2)$. A rule that may transform a clause into more than one clause is called a splitting rule, while it is called non-splitting rule otherwise. An ET-rule generation problem on $\langle D, \mathcal{A}_1, \mathcal{A}_2 \rangle$ is defined to be a problem of finding rules that transform a clause into a finite number of clauses preserving model-intersection of the whole descriptions. Given a clause C_1 from $\mathcal{A}_1$ to $\mathcal{A}_2$, a rule generation problem with respect to C_1 is a rule generation problem on $\langle D, \mathcal{A}_1, \mathcal{A}_2 \rangle$ that finds rules that is applicable to C_1. When $\mathcal{A}_1$ and $\mathcal{A}_2$ are known from the context, they are omitted for simplicity and an ET-rule generation problem on $\langle D, \mathcal{A}_1, \mathcal{A}_2 \rangle$ (with respect to C_1) is simply called an ET-rule generation problem on D (with respect to C_1).

3.4. **Examples.** Let $\mathcal{A}_1$ be the set of all atoms that have the predicates in $\{rev, app, eq\}$. Let $\mathcal{A}_2$ be the set of all atoms that have the predicate ans. $\mathcal{A}_1$ and $\mathcal{A}_2$ are closed subset of $\mathcal{A}$. Let D be the set of definite clauses defined in Section 2.1. Then each rule in Section 2.2 is an ET rule that transforms declarative descriptions in $Dscr(D, \mathcal{A}_1, \mathcal{A}_2)$ and is a solution to the ET-rule generation problem on $\langle D, \mathcal{A}_1, \mathcal{A}_2 \rangle$ with respect to some clause in the ET computation in Section 2.2. More precisely, by the rule generation

- with respect to $rev(X, [\,])$, we may have a rule:
  $rev(X, [\,]) \Rightarrow eq(X, [\,])$;
- with respect to $rev(X, [A|Y])$, we may have a rule:
  $rev(X, [A|Y]) \Rightarrow eq(X, [C|W]), rev(W, Z), app(Z, [C], [A|Y])$;

- with respect to $app(X, [E], [A])$, we may have a rule: $app(X, [E], [A]) \Rightarrow eq(X, [\,]), eq(E, A)$; and
- with respect to $app(X, [E], [A, B|Z])$, we may have a rule: $app(X, [E], [A, B|Z]) \Rightarrow eq(X, [A|W])$, $app(W, [E], [B|Z])$.

These rules can be obtained by the meta computation that will be proposed in Section 7.

4. **Meta Atoms and Meta Descriptions.** We define meta atoms, meta clauses, and meta descriptions, which are instantiated into atoms, clauses, and descriptions.

4.1. **Meta terms and meta atoms.** Meta terms are of the same form as usual terms (simple terms and compound terms) in logic programming, but, instead of usual variables, &-variables and #-variables are used. For example, "terms" such as $f(\&A, a, \#Z)$, a, $\&B$, and $\#X$ are meta terms. An &-variable is a variable that begins with & (such as $\&X$) and can be replaced with an arbitrary usual term. A #-variable is a variable that begins with # (such as $\#Z$) and can be replaced with an arbitrary usual variable. &-variables and #-variables are called meta variables. For simplicity, we use the usual abbreviated notation for lists. For instance, $[\&A|\#Z]$ is an abbreviation of $cons(\&A, \#Z)$, and $[\&A, b, \#C]$ is an abbreviation of $cons(\&A, cons(b, cons(\#C, nil)))$. nil is also denoted by $[\,]$.

A meta atom is an expression consisting of a predicate and a sequence of (possibly zero) meta terms. For instance, expressions such as $eq(\&X, [\&A|\#Z])$ and $app(\#Z, [a|\&Y], \#Z)$ are meta atoms.

4.2. **Instantiation of meta atoms.** Let Θ be the set of all mappings θ, from the set of all meta variables to the set of all usual terms, that satisfy the following conditions:

1) An &-variable is mapped into a usual term.
2) A #-variable is mapped into a usual variable.

Assume that $\hat{\mathcal{A}}_1$ is the set of all meta atoms. Let ϕ be a mapping from $\hat{\mathcal{A}}_1 \times \Theta$ to $\mathcal{A}_1$ such that $\phi(A, \theta)$ is the atom that is obtained by substituting all meta variables v in A with $\theta(v)$.

4.3. **Meta clauses and meta descriptions.** A *meta clause* is an expression of the same form as a definite clause except that it is constructed from meta atoms instead of atoms. A meta clause is defined as follows:

1) The head is always h (called a dummy head).
2) The body is a sequence of more than zero meta atom.

A *meta description* is a set of meta clauses. The set of all meta descriptions is denoted by MD.

4.4. **Instantiation of meta clauses (ϕ).** Let $\mathcal{B}$ be the set of all $p = \langle \theta, \alpha, \beta, \gamma \rangle$ in $\Theta \times \mathcal{A}_2^* \times \mathcal{A}_1^* \times \mathcal{A}_1^*$ such that for any #-variable v, $\theta(v)$

1) is different from variables substituted for other #-variables,
2) does not appear in the terms substituted for &-variables, and
3) does not appear in α, β, and γ.

Assume that a specialization $p = \langle \theta, \alpha, \beta, \gamma \rangle$ in $\mathcal{B}$ is given. For an arbitrary meta clause

$$\hat{C} = \left(h \leftarrow \hat{B}_1, \hat{B}_2, \ldots, \hat{B}_n\right),$$

we define $\psi\left(\hat{C}, p\right)$ as a clause

$$\alpha \leftarrow \beta, \phi\left(\hat{B}_1, \theta\right), \phi\left(\hat{B}_2, \theta\right), \ldots, \phi\left(\hat{B}_n, \theta\right), \gamma$$

i.e., $\psi\left(\hat{C}, p\right)$ is constructed from $\hat{C}$ by the following steps.

1) The body of $\hat{C}$, i.e., $\left(\hat{B}_1, \hat{B}_2, \ldots, \hat{B}_n\right)$, is instantiated by θ into a sequence δ of atoms on $\mathcal{A}_1$, i.e., $\delta = \left(\phi\left(\hat{B}_1, \theta\right), \phi\left(\hat{B}_2, \theta\right), \ldots, \phi\left(\hat{B}_n, \theta\right)\right)$.
2) The head of $\psi\left(\hat{C}, p\right)$ is α, which is the second element of p.
3) The body of $\psi\left(\hat{C}, p\right)$ is the atom sequence obtained by appending β, δ, and γ in that order.

$\psi\left(\hat{C}, p\right)$ belongs to $Dscr(\mathcal{A}_1, \mathcal{A}_2)$ since

1) α is a sequence of elements in $\mathcal{A}_2$,
2) β and γ are sequences of elements in $\mathcal{A}_1$, and
3) $\phi\left(\hat{B}_i, \theta\right)$ is an element of $\mathcal{A}_1$.

The operation for obtaining a clause $\psi\left(\hat{C}, p\right)$ from a meta clause $\hat{C}$ is called *instantiation* by p.

4.5. **Instantiation of meta descriptions (ψ).** A meta description M is instantiated into a declarative description by p, denoted $\psi(M, p)$, defined by

$$\psi(M, p) = \left\{C \mid \left(C = \psi\left(\hat{C}, p\right)\right) \ \& \ \left(\hat{C} \in M\right)\right\}.$$

This operation for obtaining a declarative description from a meta description M is called *instantiation* by p.

5. **Rewriting Rules Consisting of Meta Atoms.** We give the syntax of rewriting rules by using meta atoms, where meta variables play an important role.

5.1. **Syntax of rewriting rules by using meta atoms.** To discuss rule generation, we represent rewriting rules by using meta atoms. The syntax of a rewriting rule is as follows:

$$\begin{aligned}
&\alpha, \{cond\} \\
&\quad \Rightarrow \{exec_1\}, \beta_1; \\
&\quad \Rightarrow \{exec_2\}, \beta_2; \\
&\quad \vdots \\
&\quad \Rightarrow \{exec_n\}, \beta_n,
\end{aligned}$$

where $n \in \{1, 2, \ldots\}$, *cond* and $exec_i$ are optional sequence of built-in meta atoms, and each of $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ is a sequence of meta atoms. α may have a sequence *cond* of built-in atoms, which is called a condition part. β_i $(i \leq n)$ may have a sequence $exec_i$ of built-in atoms, which is called an execution part. The rule is applied if α matches a sequence of atoms in the left-hand side of a target clause, and the condition part *cond* succeeds. The sequence that is matched by α is replaced with β_i after execution of $exec_i$. If execution of $exec_i$ succeeds with an instantiation θ, the new clause is specialized by θ. If execution of $exec_i$ fails, the new clause is removed. Since the *false* atom fails, the rule $(\alpha \Rightarrow \{false\})$ in Section 7.1 removes the target clause to which the rule is applicable.

The next three rules are simple rewriting rules.

(a) $initial(\&A, \&B) \Rightarrow app(\&A, \#Y, \&B)$.

(b)
$$\begin{aligned}
&app(\&X, \&Y, \&Z) \\
&\quad \Rightarrow eq(\&X, [\,]),\ eq(\&Y, \&Z); \\
&\quad \Rightarrow eq(\&X, [\#A|\#V]),\ eq(\&Z, [\#A|\#W]),\ app(\#V, \&Y, \#W).
\end{aligned}$$

(c) $app(\&X, \#Y, \&Z) \Rightarrow initial(\&X, \&Z)$.

Given a target clause, the above rewriting rules work as follows.

- By the rewriting rule (a), a target atom that $initial(\&A, \&B)$ matches is rewritten into a new atom of the form $app(\&A, \#Y, \&B)$. In this case, the values of $\&A$ and $\&B$ are determined uniquely. A new variable, which does not appear in the target clause, is introduced and substituted for $\#Y$.
- By the rewriting rule (b), a target atom that $app(\&X, \&Y, \&Z)$ matches is rewritten into two clauses. In making the first clause, the target atom is changed into a sequence of $eq(\&X, [\,])$ and $eq(\&Y, \&Z)$. Since the values of $\&X$, $\&Y$, and $\&Z$ are determined uniquely, the first clause is determined uniquely. In making the second clause, the target atom is changed by the rule (b) into three atoms of the form $eq(\&X, [\#A|\#V])$, $eq(\&Z, [\#A|\#W])$, and $app(\#V, \&Y, \#W)$. Three new variables, which do not appear in the target clause, are introduced and substituted for $\#A$, $\#V$, and $\#W$.
- By the rewriting rule (c), an atom that $app(\&X, \#Y, \&Z)$ matches is rewritten into an atom of the form $initial(\&X, \&Z)$. If the variable that is matched by $\#Y$ has an occurrence in other parts of the target clause, the matching fails, and the rule cannot be applied.

5.2. **Rich representational power by two kinds of variables.** Recall the set R_2 of rewriting rules in Section 2. Usual atoms are used in these rules.

$rev(X, [\,]) \Rightarrow eq(X, [\,])$.
$rev(X, [A|Y]) \Rightarrow eq(X, [C|W]), rev(W, Z), app(Z, [C], [A|Y])$.
$app(X, [E], [A]) \Rightarrow eq(X, [\,]), eq(E, A)$.
$app(X, [E], [A, B|Z]) \Rightarrow eq(X, [A|W]), app(W, [E], [B|Z])$.

These rules are represented using meta atoms as follows.

$rev(\&X, [\,]) \Rightarrow eq(\&X, [\,])$.
$rev(\&X, [\&A|\&Y]) \Rightarrow eq(\&X, [\#C|\#W]), rev(\#W, \#Z), app(\#Z, [\#C], [\&A|\&Y])$.
$app(\&X, [\&E], [\&A]) \Rightarrow eq(\&X, [\,]), eq(\&E, \&A)$.
$app(\&X, [\&E], [\&A, \&B|\&Z]) \Rightarrow eq(\&X, [\&A|\&W]), app(\&W, [\&E], [\&B|\&Z])$.

The former is called one-kind variable notation, and the latter two-kind variable notation. We consider translation from one-kind variable notation to two-kind variable notation as follows.

1) Usual variables that appear in the left-hand side of a rule are transformed into &-variables.
2) Usual variables that do not appear in the left-hand side of a rule are transformed into #-variables.

Obviously, this translation method is not one-to-one. One-kind-variable rules have less expressive power than two-kind-variable rules. For example,

$$app(\&X, \#Y, \&Z) \Rightarrow initial(\&X, \&Z)$$

cannot be represented by one-kind-variable rules since $app(\&X, \#Y, \&Z)$ includes a #-variable in the left-hand side of the rule. According to the correspondence above, the rule $(app(X, Y, Z) \Rightarrow initial(X, Z))$ is transformed into a rule:

$$app(\&X, \&Y, \&Z) \Rightarrow initial(\&X, \&Z),$$

which has not the intended meaning.

6. **Meta Meta Atoms and Meta Rules.** We define the concept of meta rules, a syntax of meta rules, and the correctness of meta rules.

6.1. **Meta meta terms and meta meta atoms.** An $*$-variable is a variable that begins with $*$ (such as $*X$). A %-variable is a variable that begins with % (such as $\%Z$). $*$-variables and %-variables are called meta meta variables. Meta meta terms are of the same form as usual terms (simple terms and compound terms) in logic programming, but, instead of usual variables, $*$-variables and %-variables are used. For example, "terms" such as $f(*A, a, \%Z)$, a, $*B$, and $\%X$ are meta meta terms. We use the usual abbreviated notation for lists.

A meta meta atom is an expression consisting of a predicate and a sequence of (possibly zero) meta meta terms. For instance, expressions such as $eq(*X, [*A|\%Z])$ and $app(\%Z, [a|*Y], \%Z)$ are meta meta atoms.

6.2. **Instantiation of meta meta atoms.** Let Θ be the set of all mappings θ, from the set of all $*$-variables and %-variables to the set of all meta terms, that satisfy the following conditions.

1) An $*$-variable is mapped into a meta term.
2) A %-variable is mapped into a #-variable.

Assume that $\hat{\hat{\mathcal{A}}}_1$ is the set of all meta meta atoms. Let $\hat{\phi}$ be a mapping from $\hat{\hat{\mathcal{A}}}_1 \times \Theta$ to $\hat{\mathcal{A}}_1$ such that $\hat{\phi}(A, \theta)$ is the atom that is obtained by substituting all meta meta variables v in A with $\theta(v)$.

6.3. **Syntax of meta rules.** Meta rules consist of meta meta atoms that may have $*$-variables and %-variables. The syntax of a meta rule is as follows:

$$
\begin{aligned}
&\alpha, \{cond\} \\
&\quad \Rightarrow \{exec_1\}, \beta_1; \\
&\quad \Rightarrow \{exec_2\}, \beta_2; \\
&\quad \vdots \\
&\quad \Rightarrow \{exec_n\}, \beta_n,
\end{aligned}
$$

where $n \in \{1, 2, \ldots\}$, *cond* and $exec_i$ are optional sequence of built-in meta meta atoms, and each of $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ are a sequence of meta meta atoms. α may have a sequence *cond* of built-in atoms, which is called a condition part. β_i $(i \leq n)$ may have a sequence $exec_i$ of built-in atoms, which is called an execution part. The rule is applied if α matches a sequence of meta atoms in the left-hand side of a target meta clause, and the condition part *cond* succeeds. Basically, the target clause is replaced with n clauses, which is modified by the execution of *cond* and $exec_i$ as follows: If execution of $exec_i$ succeeds with an instantiation θ, the sequence that is matched by α is replaced with β_i, and the new clause is specialized by θ. If execution of $exec_i$ fails, no clause is produced. Consider the rule $(eq(\%V, *Z) \Rightarrow \{bind(\%V, *Z)\})$ which will be shown in Section 7.2. If it is applicable to a target clause, it removes the atom that is matched by $eq(\%V, *Z)$, and specializes the target clause with $\{\%V/*Z\}$.

The next three rules are simple meta rules.

(a) $initial(*A, *B) \Rightarrow app(*A, \%Y, *B)$.

(b)
$$
\begin{aligned}
&app(*X, *Y, *Z) \\
&\quad \Rightarrow eq(*X, [\,]),\ eq(*Y, *Z); \\
&\quad \Rightarrow eq(*X, [\%A|\%V]),\ eq(*Z, [\%A|\%W]),\ app(\%V, *Y, \%W).
\end{aligned}
$$

(c) $app(*X, \%Y, *Z) \Rightarrow initial(*X, *Z)$.

We will explain meta rules using the above examples. Given a target meta clause, each of the above meta rules works as follows.

- By the meta rule (a), a target meta atom that $initial(*A, *B)$ matches is rewritten into a new meta atom of the form $app(*A, \%Y, *B)$. In this case, the values of $*A$

and $*B$ are determined uniquely by the matching. A new #-variable, which does not appear in the target meta clause, is introduced and substituted for $\%Y$.

- By the meta rule (b), a target meta atom that $app(*X, *Y, *Z)$ matches is rewritten. In making the first meta clause, the target meta atom is changed into a sequence of $eq(*X, [\,])$ and $eq(*Y, *Z)$. Since the values of $*X$, $*Y$, and $*Z$ are determined uniquely, the first meta clause is determined uniquely. In making the second meta clause, the target meta atom is changed by the meta rule (b) into three meta atoms of the form $eq(*X, [\%A|\%V])$, $eq(*Z, [\%A|\%W])$, and $app(\%V, *Y, \%W)$. Three new #-variables, which do not appear in the target meta clause, are introduced and substituted for $\%A$, $\%V$, and $\%W$.
- By the meta rule (c), a meta atom that $app(*X, \%Y, *Z)$ matches is rewritten into a meta atom of the form $initial(*X, *Z)$. If the #-variable that is matched by $\%Y$ has an occurrence in other parts of the target clause, the matching fails, and the rule cannot be applied.

6.4. **Correctness of meta rules for user-atoms.** We explain the correctness of the meta rules (a), (b), and (c) in Section 6.3. By the meta rule (a), an *initial* meta atom is replaced with an *app* meta atom. The meta meta variable $\%Y$ in (a) produces a # meta variable, which generates a new variable that does not appear in the target clause. It follows that the corresponding transformation in the level of usual declarative descriptions is unfolding with respect to the definition (A) of the *initial* predicate:

(A) $initial(A, B) \leftarrow app(A, Y, B)$.

By the meta rule (b), an *app* meta atom is replaced with two meta atom sequences. The meta meta variables $\%A$, $\%V$, and $\%W$ in (b) produce mutually different # meta variables, each of which generates a new variable that does not appear in other part of the target clause. It follows that the corresponding transformation in the level of usual clauses is unfolding with respect to the definitions (B1) and (B2) of the *app* predicate:

(B1) $app(X, Y, Z) \leftarrow eq(X, [\,]), eq(Y, Z)$;
(B2) $app(X, Y, Z) \leftarrow eq(X, [A|V]), eq(Z, [A|W]), app(V, Y, W)$.

By the meta rule (c), an *app* meta atom is replaced with an *initial* meta atom. The meta meta variable $\%Y$ in (c) matches a # meta variables, which does not appear in the other part of the target meta atom. The corresponding transformation in the level of usual clauses is reverse of unfolding with respect to the definition (A) of the *initial* predicate. Hence it preserves model-intersection.

6.5. **Correctness of meta rules for equality.** We also use meta rules for *eq* atoms.

(Ea) $eq([*A|*X], [*B|*Y]) \Rightarrow eq(*A, *B), eq(*X, *Y)$.
(Eb) $eq(*X, *X) \Rightarrow .$
(Ec) $eq([\,], [*X|*Y]) \Rightarrow \{false\}$.
$eq([*X|*Y], [\,]) \Rightarrow \{false\}$.
(Ed) $eq(\%V, *Z) \Rightarrow \{bind(\%V, *Z)\}$.
$eq(*Z, \%V) \Rightarrow \{bind(\%V, *Z)\}$.

We explain the correctness of the meta rules (Ea), (Eb), (Ec), and (Ed). By the meta rule (Ea), an *eq* meta atom is replaced with two meta *eq* atoms. The corresponding transformation in the level of usual clauses is decomposition of lists, which is obviously correct transformation. The correctness of meta rules (Eb) and (Ec) is similarly proved.

By the meta rule (Ed), equality is solved. Since the meta meta variable $\%Y$ in (Ed) matches a # meta variables, which does not appear in the other part of the target meta atom. The equality meta atom in (Ed) is instantiated to equality atoms with variables that do not appear in the other part of the target clause. Hence the variables can be bound

without affecting other part of the target clause, and the variable binding obviously yields correct transformation.

7. **Meta Computation for Rule Generation.** We discuss meta computation and rule generation by meta computation.

7.1. **Rule generation by meta computation.** Let D be a clause set. We propose a method of generating ET rules by repeated application of meta rules to meta clauses, which is called meta computation.

1) Assume that a non-empty sequence α of meta atoms, called an input meta atom sequence, is given.
2) Prepare a set R of correct meta rules with respect to D.
3) Let M_1 be a meta description $\left\{\hat{C}\right\}$, which is a singleton set of a meta clause $\hat{C} = (h \leftarrow \alpha)$.
4) Obtain a meta description M_k by transforming M_1 by repeated application of meta rules in R: $M_1 \to M_2 \to \cdots \to M_k$.
5) If there is $n \geq 1$ such that $M_k = \{(h \leftarrow \beta_1), (h \leftarrow \beta_2), \ldots, (h \leftarrow \beta_n)\}$, then obtain a rule
$$\begin{aligned} \alpha &\Rightarrow \beta_1; \\ &\Rightarrow \beta_2; \\ &\vdots \\ &\Rightarrow \beta_n. \end{aligned}$$
6) If $M_k = \{\}$, then obtain a rule
$$\alpha \Rightarrow \{\mathit{false}\}.$$

7.2. **Generating non-splitting rules.** Using the meta rules in Section 6 together with an input meta atom sequence $[\mathit{initial}(\&X, [\,])]$, we can construct meta computation as follows:

(1) $h \leftarrow \underline{\mathit{initial}(\&X, [\,])}$.
(2) $h \leftarrow \underline{\mathit{app}(\&X, \#Y, [\,])}$.
(3) $h \leftarrow \mathit{eq}(\&X, [\,]), \mathit{eq}(\#Y, [\,])$.
$h \leftarrow \mathit{eq}(\&X, [\#A|\#V]), \underline{\mathit{eq}([\,], [\#A|\#V])}, \mathit{app}(\#V, \#Y, \#W)$.
(4) $h \leftarrow \mathit{eq}(\&X, [\,]), \underline{\mathit{eq}(\#Y, [\,])}$.
(5) $h \leftarrow \mathit{eq}(\&X, [\,])$.

Application of meta rules in this meta computation is listed:

(1) $\Rightarrow$ (2) by application of the meta rule (a)
(2) $\Rightarrow$ (3) by application of the meta rule (b)
(3) $\Rightarrow$ (4) by application of the meta rule (Ec)
(4) $\Rightarrow$ (5) by application of the meta rule (Ed)

Since meta clause (1) is transformed into (5), we obtain a rule:

$$\mathit{initial}(\&X, [\,]) \Rightarrow \mathit{eq}(\&X, [\,]).$$

An explanation of meta computation from (1) to (5) is given as follows. (1) is transformed into (2) by the meta rule (a), where the variable $\%Y$ introduced a new $\#Y$ variable, which does not appear elsewhere. Using the meta rule (b), (2) is transformed into two meta clauses in (3). The variables $\%A$, $\%V$, and $\%W$ produce three new variables, $\#A$, $\#V$, and $\#W$. Then the second meta clause in (3) was removed by the meta rule (Ec). The second body atom in (4) was removed by the meta rule (Ed) to have the final meta clause (5). There is no meta rule that is applicable to (5).

7.3. **Generating splitting rules.** We use the same meta rules as given in Section 7.2, and make meta computation starting with an input meta atom sequence $[initial(\&X, [\&A|\&Z])]$, as follows.

(1) $h \leftarrow \underline{initial(\&X, [\&A|\&Z])}$
by the meta rule (a)

(2) $h \leftarrow \underline{app(\&X, \#Y, [\&A|\&Z])}$
by the meta rule (b)

(3) $h \leftarrow eq(\&X, [\,]), \underline{eq(\#Y, [\&A|\&Z])}.$
$h \leftarrow eq(\&X, [\#B|\#V]), eq([\&A|\&Z], [\#B|\#W]), app(\#V, \#Y, \#W).$
by the meta rule (Ed)

(4) $h \leftarrow eq(\&X, [\,]).$
$h \leftarrow eq(\&X, [\#B|\#V]), \underline{eq([\&A|\&Z], [\#B|\#W])}, app(\#V, \#Y, \#W).$
by the meta rule (Ea)

(5) $h \leftarrow eq(\&X, [\,]).$
$h \leftarrow eq(\&X, [\#B|\#V]), \underline{eq(\&A, \#B)}, eq(\&Z, \#W), app(\#V, \#Y, \#W).$
by the meta rule (Ed)

(6) $h \leftarrow eq(\&X, [\,]).$
$h \leftarrow eq(\&X, [\&A|\#V]), \underline{eq(\&Z, \#W)}, app(\#V, \#Y, \#W).$
by the meta rule (Ed)

(7) $h \leftarrow eq(\&X, [\,]).$
$h \leftarrow eq(\&X, [\&A|\#V]), \underline{app(\#V, \#Y, \&Z)}.$
by the meta rule (c)

(8) $h \leftarrow eq(\&X, [\,]).$
$h \leftarrow eq(\&X, [\&A|\#V]), initial(\#V, \&Z).$

Since meta clause (1) is transformed into two meta clauses in (8), we obtain a rule, which is represented by

$$\begin{aligned} &initial(\&X, [\&A|\&Z]) \\ &\quad \Rightarrow eq(\&X, [\,]); \\ &\quad \Rightarrow eq(\&X, [\&A|\#V]), initial(\#V, \&Z). \end{aligned}$$

7.4. **Making input meta atom sequence.** Meta computation starts with a meta clause consisting of an *input meta atom sequence*. Assume that we have no rule applicable to the separated description $Cs \cup Q$ at some computation state. To generate applicable rules to some clause in the set Q of clauses, we need to find input meta atom sequence for meta computation. We select a clause, called a target clause, in Q. We select a set of atoms, called target atoms, in the body of the target clause. One of the simplest methods of constructing an input meta atom sequence from selected target atoms is to make a set of simple meta atoms that can be instantiated into the given set of target atoms. For example, referring to the sequence of *ans* clauses in Section 2.2, the body atom $app(C, [A], [1, 2])$ in the third *ans* clause is subsumed by an atom $app(C, [A|R], [B, C|S])$, which is an instantiation of $app(\&C, [\&A|\&R], [\&B, \&C|\&S])$. Subsumption along with variable change from usual variables to meta variables is the basic method to construct an input meta atom sequence.

8. Linking with Previous Research.

8.1. **Minimal model semantics of definite clauses.** The previous theories for rule generation reviewed in Section 1.2 are constructed on the solution of QA problems on definite clauses, and the minimal model semantics of definite clauses is taken. Consider a QA problem $\langle D, q \rangle$, where D is a set of definite clauses, and q is an atom. D is called

background knowledge, and q is a query atom. The answer of the QA problem $\langle D, q\rangle$ is equal to the set $(instance(q) \cap \mathcal{M}(D))$, where $instance(q)$ is the set of all ground instances of q, and $\mathcal{M}(D)$ is the minimal model of D. By the model intersection property [14], the minimal model of D is equal to the intersection of all models of D, i.e., $\mathcal{M}(D) = \bigcap Models(D)$.

An ET-based solution of a QA problem is given as follows. We introduce a query clause $(ans(x_1, x_2, \ldots, x_n) \leftarrow q)$, where $x_1, x_2, \ldots, x_n$ are mutually different variables in q. Let Q be a singleton of the query clause above. We consider MI problems with $Cs = D \cup Q$, assuming D and Q are sets of definite clauses. ET computation consists of repeated application of ET rules to Q preserving $\mathcal{M}(D \cup Q)$.

8.2. **Extension of knowledge representation.** The previous meta computation theory cannot be applicable to a clause set that includes non-definite clauses. We have extended the theory of meta computation to non-definite clauses. The theory proposed in this paper can be applied to arbitrary clauses.

In logic programming, many specific models have been invented. The minimal model semantics is one of them. In our theory, all models are considered to be of equal value. No specific model is taken to be more important than others, which is common to first-order logic. The minimal model semantics that underlies the previous theory is the main hindrance to the extension from the theory of meta computation on definite clauses to the one on arbitrary clauses.

Extension to the meta computation theory on all-model semantics is realized as follows. We take the class of model-intersection problems (MI problems), which is an extension of the class of QA problems. Assume that D and Q are sets of arbitrary clauses. Taking D as background knowledge, and Q a set of query clauses, we consider an MI problem with $Cs = D \cup Q$, the answer to which is $\bigcap Models(D \cup Q)$. If $D \cup Q$ contains only definite clauses, the answer is represented by the minimal model of $D \cup Q$, i.e.,

$$\bigcap Models(D \cup Q) = \mathcal{M}(D \cup Q).$$

It follows that if we consider MI problems with all-model semantics in place of QA problems on definite clauses with minimal model semantics, the meta computation theory can be applied to all problems on definite/non-definite clauses.

8.3. **Theoretical framework for rule generation.** Production systems or expert systems [15, 16, 17, 18, 19] have rules. Rules are written by mimicing the reasoning of a human expert in solving a knowledge intensive problem. There is no theoretical framework for defining the correctness of various rules and rule generation based on the correctness.

Resolution-based computation in logic programming has no concept of utilizing various rules to improve solvability and efficiency [14, 20, 21, 22, 23, 24]. Again, there is no theoretical framework for defining the correctness of various rules and rule generation based on the correctness.

9. **Concluding Remarks.** In the ET model, computation is regarded as equivalent transformation of declarative descriptions, and a program consists of ET rules and a control description. Programs may be improved by adoption of new rules. Under the ET model, we have the description/instantiation structure. Atoms are instantiated into ground atoms. Descriptions consist of atoms, and are instantiated into ground descriptions.

A class of rewriting rules is introduced. A rewriting rule is a rule for transforming a description into a description, and is constructed from meta atoms. Meta atoms have two kinds of variables, &-variables and #-variables. An &-variable is instantiated into a

usual term, while a #-variable is instantiated into a usual variable that does not appear elsewhere in a target query clause. Meta atoms are instantiated into atoms.

A class of meta rules is introduced. A meta rule is a rule for transforming a meta description into a meta description, and is constructed from meta meta atoms. Meta meta atoms have two kinds of variables, i.e., $*$-variables and %-variables. A $*$-variable is instantiated into a meta term, while a %-variable is instantiated into a #-variable that does not appear elsewhere in a target meta clause. Meta meta atoms are instantiated into meta atoms.

A meta computation consists of repeated application of meta rules to a meta description, by which we obtain a sequence of meta descriptions. The first and the last meta descriptions of the obtained sequence give a pair of meta descriptions. The pair of meta descriptions determines a partial mapping for transforming a description into a description. A rewriting rule is obtained as a partial mapping for transforming a description into a description.

A theory of meta computation was proposed on the ET-based computation of MI problems. We can generate rewriting rules for any background knowledge that is represented by a set of arbitrary clauses, not only definite clauses but also non-definite clauses.

Rule generation and meta computation in this paper will open up a new stage of computation and program synthesis. By rule generation, greater increase of solvability is expected for larger and practical problems. By generating various, less general and more efficient rules other than the general unfolding rules, we can solve many problems that cannot be solved without rule generation, and can improve efficiency of computation for many problems. Toy problems may even suffer from non-solvability due to shortage of rewriting rules [13]. It follows that the theory of rule generation improves the practicality of the solution method.

Rule generation is fundamental for solving problems since it produces programs from problems. A theory of program synthesis will be constructed for the largest class of logical problems on first-order formulas. Sequential and/or parallel programs to solve MI problems will be synthesized based on the technique of rule generation.

REFERENCES

[1] K. Akama and E. Nantajeewarawat, Solving query-answering problems with constraints for function variables, *Proc. of the 10th Asian Conference on Intelligent Information and Database Systems*, Dong Hoi City, Vietnam, pp.36-47, 2018.

[2] K. Akama and E. Nantajeewarawat, Model-intersection problems with existentially quantified function variables: Formalization and a solution schema, *Proc. of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2016)*, vol.2, Porto, Portugal, pp.52-63, 2016.

[3] K. Akama and E. Nantajeewarawat, Unfolding existentially quantified sets of extended clauses, *Proc. of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2016)*, vol.2, Porto, Portugal, pp.96-103, 2016.

[4] K. Akama, E. Nantajeewarawat and T. Akama, Computation control by prioritized ET rules, *Proc. of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018)*, vol.2, Seville, Spain, pp.84-95, 2018.

[5] K. Akama, E. Nantajeewarawat and T. Akama, Logical problem solving framework, *Proc. of the 11th Asian Conference on Intelligent Information and Database Systems*, Yogyakarta, Indonesia, pp.28-40, 2019.

[6] K. Akama, H. Koike and E. Miyamoto, A theoretical foundation for generation of equivalent transformation rules, *Program Transformation, Symbolic Computation and Algebraic Manipulation*, Research Institute for Mathematical Sciences, Kyoto University, Koukyuroku, no.1125, pp.44-58, 2000.

[7] K. Akama, H. Koike and H. Mabuchi, A theoretical foundation of program synthesis by equivalent transformation, *Perspectives of System Informatics, Lecture Notes in Computer Science*, vol.2244, pp.131-139, 2001.

[8] K. Akama, E. Nantajeewarawat and H. Koike, Program synthesis based on the equivalent transformation computation model, *Proc. of the 12th International Workshop on Logic Based Program Development and Transformation (LOPSTR 2002)*, Madrid, Spain, pp.285-304, 2002.
[9] K. Akama, E. Nantajeewarawat and H. Koike, Program generation in the equivalent transformation computation model using the squeeze method, *Lecture Notes in Computer Science*, vol.4378, pp.41-54, 2007.
[10] K. Miura, K. Akama and H. Mabuchi, Creation of ET rules from logical formulas representing equivalent relations, *International Journal of Innovative Computing, Information and Control*, vol.5, no.2, pp.263-277, 2009.
[11] K. Miura, K. Akama, H. Mabuchi and H. Koike, Theoretical basis for making equivalent transformation rules from logical equivalences for program synthesis, *International Journal of Innovative Computing, Information and Control*, vol.9, no.6, pp.2635-2650, 2013.
[12] K. Miura and K. Akama, ET-based bidirectional search for proving formulas in the class ES, *International Journal of Innovative Computing, Information and Control*, vol.10, no.6, pp.1999-2009, 2014.
[13] K. Akama and E. Nantajeewarawat, Computing logically with generation of equivalent transformation rules, *International Journal of Innovative Computing, Information and Control*, vol.18, no.1, pp.315-330, 2022.
[14] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.
[15] J. Durkin, *Expert Systems: Design and Development*, Prentice Hall, New York, 1994.
[16] S. Lindsay, *Practical Applications of Expert Systems*, John Wiley & Sons Inc., Chichester, 1988.
[17] J. C. Giarratano, *Expert Systems: Principles and Programming*, Brooks Cole, Pacific Grove, 1998.
[18] A. L. Kidd, *Knowledge Acquisition for Expert Systems: A Practical Handbook*, Plenum Publishing Corporation, New York, 1987.
[19] A. Alamoudi, A. Alomari, S. Alwarthan and A. Rahman, A rule-based information extraction approach for extracting metadata from PDF books, *ICIC Express Letters, Part B: Applications*, vol.12, no.2, pp.121-132, 2021.
[20] J. A. Robinson, A machine-oriented logic based on the resolution principle, *Journal of the ACM*, vol.12, pp.23-41, 1965.
[21] R. A. Kowalski, Predicate logic as a programming language, *Proc. of the 6th IFIP Congress 1974*, Stockholm, Sweden, pp.569-574, 1974.
[22] R. A. Kowalski, Algorithm = logic + control, *Communications of the ACM*, vol.22, pp.424-435, 1979.
[23] K. Doets, *From Logic to Logic Programming*, The MIT Press, 1994.
[24] M. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd Edition, Springer-Verlag, 1996.

Author Biography

Kiyoshi Akama received the B.Eng. and M.Eng. degrees in control engineering from Tokyo Institute Technology, Japan, in 1973 and 1975, respectively; and the D.Eng. degree in control engineering from Tokyo Institute Technology, Japan, in 1989. He was an assistant professor at Faculty of Engineering, Tokyo Institute Technology, Japan, 1979-1981; a lecturer at Faculty of Letters, Hokkaido University, Japan, 1981-1989; an associate professor at Faculty of Engineering, Hokkaido University, Japan, 1989-1999; a professor at Center for Multimedia Studies, Hokkaido University, Japan, 1999-2003; a professor at Information Initiative Center, Hokkaido University, Japan, 2003-2013; a specially-appointed professor at Information Initiative Center, Hokkaido University, 2013-2015; a professor at Graduate School of Information Science and Technology, Hokkaido University, Japan, 1999-2015.

Dr. Akama is currently an emeritus professor of Hokkaido University, Japan. His research interests include artificial intelligence, computer science, logic and computation, program generation and computation based on the equivalent transformation model, programming paradigms, and knowledge representation.

Ekawit Nantajeewarawat received the B.Eng. degree in computer engineering from Chulalongkorn University, Thailand, in 1987; and the M.Eng. and D.Eng. degrees in computer science from the Asian Institute of Technology, Thailand, in 1991 and 1997, respectively.

Dr. Nantajeewarawat is currently an associate professor of computer science at Sirindhorn International Institute of Technology, Thammasat University, Thailand. His research interests include knowledge representation, automated reasoning, rule-based equivalent transformation, program synthesis, formal ontologies, and object-oriented modelling.