International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

Generating *Speq* Rules based on Automatic Proof of Logical Equivalence

Katsunori Miura, Kiyoshi Akama, and Hiroshi Mabuchi

Abstract**—** In the Equivalent Transformation (ET) computation model, a program is constructed by the successive accumulation of ET rules. A method by meta-computation by which a correct ET rule is generated has been proposed. Although the method covers a broad range in the generation of ET rules, all important ET rules are not necessarily generated. Generation of more ET rules can be achieved by supplementing generation methods which are specialized for important ET rules. A *Specialization-by-Equation (Speq) rule* is one of those important rules. A *Speq* rule describes a procedure in which two variables included in an atom conjunction are equalized due to predicate constraints. In this paper, we propose an algorithm that systematically and recursively generate *Speq* rules and discuss its effectiveness in the synthesis of ET programs. A *Speq* rule is generated based on proof of a logical formula consisting of given atom set and dis-equality. The proof is carried out by utilizing some ET rules and the ultimately obtained rules in generating *Speq* rules.

Keywords**—** Equivalent transformation, ET rule, Equation of two variables, Rule generation, Specialization-by-Equation rule

I. Introduction

GENERATING efficient programs which are correct with respect to their specifications is very important. Many studies generate a low-level language program from a specification (Refer to Fig. 1). In their studies, a specification is strictly defined by a formal description approach to efficiently generate a program. An algebraic specification [4] is one of formal description approaches, where a specification is defined by an equality. A method for generating C or Java programs from algebraic specifications is proposed in [7], [11]. A method [3] for generating C++ programs from specifications defined by UML is also proposed.

A theory of program generation based on the Equivalent Transformation (ET) computation model [1] has also been proposed by our research group, where a specification is defined by a set of definite clauses, and a low-level language program is generated from their specification (Refer to Fig. 1). A crucial difference between our approach and others is that we have a space of ET rules for searching algorithms that is correct with respect to the given specification, and generate a program from a set of ET rules that is found by the search. A framework for generating C programs from ET rule sets has been proposed in [18].

An *ET rule* is a rewriting rule that describes a procedure to replace a clause set with another clause set preserving the equivalent declarative meaning. ET rules have abundant expressive powers that can describe various procedures, and each ET rule is completely independent of other ET rules. An algorithm is constructed by successively accumulating ET rules, and it is guaranteed that an algorithm constructed by correct ET rules is correct with respect to their specifications. ET rules have such good features for constructing algorithms. Generation of ET rules is the most important task in making programs.

A general method for generating correct ET rules has been proposed in [9]. In this method, an ET rule is generated by a finite synthesis of basic equivalent transformations using meta-computation. Since a method using meta-computation covers a broad range in the generation of ET rules, a system [14] based on this method can generate various ET rules. However, in the method using meta-computation, 1. it is not necessarily the case that all ET rules are generated, and 2. there are ET rules which cannot be generated in a short period of time.

In this paper, we consider that ET rule generation focusing on the following items is important.

1) Even if a generation range of ET rules is narrowed, ET rules derived in the range are important.
2) ET rules cannot be generated by a finite synthesis of basic equivalent transformations.
3) An algorithm which efficiently generate ET rules can be found.

One of the rules that satisfy the items mentioned above is a *Specialization-by-Equation (Speq) rule*. A *Speq* rule describes a procedure in which two variables included in an atom conjunction are equalized due to predicate constraints. Generating a *Speq* rule is essential because the rule solves problems

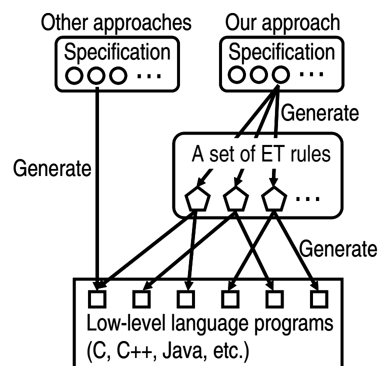

Fig. 1. Generating low-level language programs from their specification

Katsunori Miura is with the Faculty of Computer Science, Hokkaido University, Sapporo, Japan, email: kmiura@uva.cims.hokudai.ac.jp
Kiyoshi Akama is with the Information Initiative Center, Hokkaido University, Sapporo, Japan, email: akama@iic.hokudai.ac.jp
Hiroshi Mabuchi is with the Faculty of Software and Information Science, Iwate Prefectural University, Iwate, Japan, email: mabu@soft.iwate-pu.ac.jp

more efficiently and is absolutely necessary to solve a certain problem within a finite period of time. When a conjunction of atoms is given as an input, the algorithm proposed in this paper outputs all *Speq* rules obtained from the input. An ET rule set is called a program in the rest of this paper.

This paper describes the syntax of ET rules, specifications, programs and computation on the ET computation model in Section 2 and discusses the importance of *Speq* rules in Section 3. It also proposes an algorithm that generates a *Speq* rule in Section 4 and presents the process of generating the rules by the proposed algorithm in Section 5. Furthermore, the effectiveness of the algorithm is discussed in Section 6.

The following notations will be used. Taking $\mathbb{A}$ as an atom, let $Var(\mathbb{A})$ be the set of variables on $\mathbb{A}$. Since an atom conjunction is treated as an atom set in this paper, it is called an atom set in the rest of this paper.

II. Program construction in the ET computation model

A. Definition of ET rules

In the ET computation model, a procedure is described by a set of declarative meaning-preserving rewriting rules, each of which is called an *ET rule*. An ET rule replaces a clause set Cls with another clause set Cls' which has an equivalent declarative meaning as Cls.

[Syntax of ET rules]

An ET rule is composed of a head part (Hs), a condition part (C), an execution part (E_n) and a replacement part (Bs_n); where n is a nonnegative integer.

$$\begin{aligned} Hs,\ \{C\} &\Rightarrow \{E_1\},\ Bs_1\ ; \\ &\Rightarrow \{E_2\},\ Bs_2\ ; \\ &\ \ \vdots \\ &\Rightarrow \{E_n\},\ Bs_n\ . \end{aligned}$$

A head part consists of one or more atoms while all other parts consist of zero or more atoms. When a head is transformed into two or more bodies as shown above, ";" is used for dividing those bodies. If a head is transformed to only one body, such an ET rule is described as ($Hs,\ \{C\} \Rightarrow \{E_1\},\ Bs_1.$). The proposed algorithm in this paper generates ET rules in which the number of bodies is one. However, ET rules with multiple bodies are used when generating ET rules by the proposed algorithm (See Section V).

B. Program and specification in the ET computation model

In the theory of program construction based on the ET computation model, a *program* is a set of ET rules. The computation of the program is represented as a finite or infinite sequence $com = [s_0, s_1, s_2, \cdots]$. For any two successive problems s_i and s_{i+1} in *com*, the program transforms s_i into s_{i+1} by one step. The transformation is carried out as long as there exists an ET rule that can be applied to a problem in *com*.

In the ET computation model, a specification is strictly defined by a set of definite clauses. Such a method where a set of definite clauses is taken as a specification is an essential paradigm used in Logic Programming (LP) [10], [15] and Constraint Logic Programming (CLP) [8], [17]. Strictly, a *specification* is a pair $\langle \mathbb{D},\ \mathbb{Q} \rangle$, where $\mathbb{D}$ is background knowledge (predicate definition) and $\mathbb{Q}$ is a set of queries. Since ET rules have *partial correctness* with respect to $\mathbb{D}$, it is guaranteed that a program consisting of only correct ET rules has partial correctness with respect to its specification.

C. Importance of ET rules

Since ET rules have complete independence with regard to partial correctness, adding new ET rules to an existing ET rule set does not impair the correctness of the existing ET rule set. This enables the function of a program to be flexibly expanded through modification of each ET rule or addition of new ET rules. In addition, the squeeze method [2] where a program is created by successively accumulating ET rules has been proposed.

Furthermore, since software systems become larger and more complex, the need for the development of cost-effective and reliable software in a short period of time increases accordingly. As a result, there has been increasing attention to component-wise program generation [5], [6], [16]. This programming method generates a program by accumulating program components one by one, each of which has passable independence. Since ET rules can be considered as components constituting a program, also when viewed from the perspective of component-wise program generation, they can confer considerable advantage [12].

III. Approaches for generating ET rules

A. Generating ET rules based on the meta-computation method

A general method for generating ET rules has been proposed in [9]. The method replaces an atom set with another atom set using a basic equivalent transformation and generates ET rules by a finite synthesis of replacements. This replacement is carried out using meta-computation, which replaces an atom set with another atom set using general ET rules (unfolding rules and build-in rules) that are obtained from the definition of predicates. Since the method using meta-computation can generate various ET rules, a system for constructing programs based on this method has achieved many successful results [14].

In the meta-computation method, paths for replacing an atom set are important. There exist many paths because of the following two items (Refer to Fig. 2).

1) There exist various ET rules applied to an atom set.
2) Number of rule applications

An ET rule is generated based on a certain evaluation from among the derived paths. Because there is a possibility that a search area may increase tremendously, the meta-computation method uses the following controls.

1) Limit the number of rule applications
2) Specify an application rule

In the meta-computation method, if the number of unfolding rule applications increases, then a search area grows correspondingly, increasing the cost of ET rule generation.

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

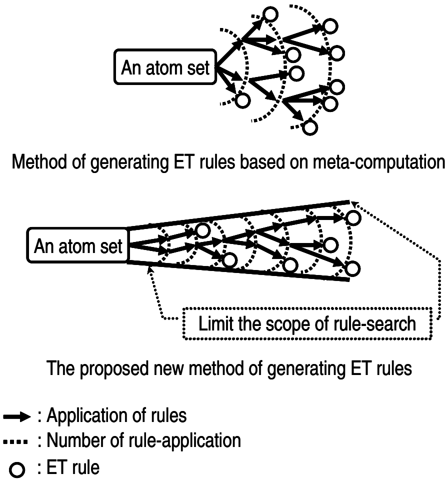

Fig. 2. Conventional rule generation method and the proposed method

B. Proposing a new method focusing on specialized rule classes

In this paper, we consider that ET rule generation focusing on the following items is important.

1) Even if a search area of ET rules is narrowed, ET rules derived in the area are important (Refer to Fig. 2).
2) ET rules cannot be generated by a finite synthesis of basic equivalent transformations.
3) An algorithm which efficiently generate ET rules can be found.

We are not dismissing the meta-computation method but, instead, taking an approach where several generation methods are combined as shown in Fig. 3. In this paper, we propose an algorithm that generates ET rules with respect to a meshed area of Fig. 3. (The area includes ET rules which cannot be generated by meta-computation method.)

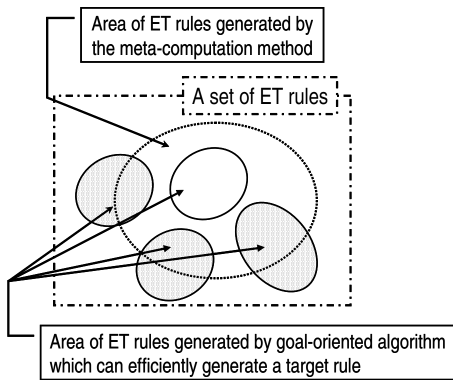

Fig. 3. Adding a new rule generation method to meta-computation method

C. The target ET rules

A Specialization-by-Equation (*Speq*) rule is one of ET rules, each of which describes a procedure in which two variables included in an atom set are equalized due to predicate constraints. For example, we will now consider a *Speq* rule for an atom set $\{copy(Vx,\ Vy),\ copy(Vx,\ Vy)\}$. A predicate *copy* is defined by the following two clauses.

$$cl1\ :\ copy([\ ],\ [\ ]) \leftarrow$$
$$cl2\ :\ copy([A \mid B],\ [A \mid C]) \leftarrow copy(B,\ C)$$

Since variables Vy and Vz are copies of a variable Vx, it is not difficult to deduce that $Vy = Vz$. This equation is defined by a *Speq* rule, and the following formula is a *Speq* rule with respect to $\{copy(Vx,\ Vy),\ copy(Vx,\ Vy)\}$. $eq(Arg1,\ Arg2)$ means that $Arg1$ is equal to $Arg2$.

$$r_{copy}\ :\ copy(Vx,\ Vy),\ copy(Vx,\ Vz) \Rightarrow eq(Vy,\ Vz),\ copy(Vx,\ Vy),\ copy(Vx,\ Vz).$$

[Importance of *Speq* rules]

A *Speq* rule has the following features:

1) an ET rule with single body, i.e., a *Speq* rule replaces a head with one body,
2) an ET rule where a *eq* atom is included in the body part.

In the ET program, primarily applying the following makes the computation efficient: 1. ET rules with single body and 2. ET rules which are useful for specializing clauses. A *eq* atom in the body part of a *Speq* rule promotes the transformation of clauses by specializing a clause efficiently. Many *Speq* rules cannot be generated by meta-computation method; therefore, an algorithm for generating *Speq* rules is essential.

[Example of computation using *Speq* rule]

Speq rules promote the transformation of definite clauses. Therefore, by using a *Speq* rule, a set of unit clauses can be obtained from a query $\mathbb{Q}$ with fewer transformations than without the use of the *Speq* rule. We now show an example of replacing a clause using r_{copy} mentioned above. Let a definite clause set $\{ans(Vc,\ Vd) \leftarrow copy([Va,\ Vb],\ Vc),\ copy([Va,\ Vb],\ Vd)\}$ be a query $\mathbb{Q}$. When r_{copy} is applied to $\mathbb{Q}$, $\mathbb{Q}$ is replaced as follows:

$$\{ans(Vc,\ Vd) \leftarrow eq(Vc,\ Vd),\ copy([Va,\ Vb],\ Vc),\ copy([Va,\ Vb],\ Vd)\}$$

When the *eq* atom is executed, the above clause set is replaced to $\{ans(Vc,\ Vc) \leftarrow copy([Va,\ Vb],\ Vc), copy([Va,\ Vb],\ Vc)\}$. Since two atoms in the body part of the above clause are the same, the above clause is replaced to $\{ans(Vc,\ Vc) \leftarrow copy([Va,\ Vb],\ Vc)\}$. Finally, when a unfolding rule is applied to $copy([Va,\ Vb],\ Vc)$, $\{ans([Va,\ Vb],\ [Va,\ Vb]) \leftarrow\}$ is obtained from $\{ans(Vc,\ Vc) \leftarrow copy([Va,\ Vb],\ Vc)\}$.

When the above computation is carried out without using *Speq* rules, unfolding rules are generally used. Because unfolding rules are applied separately to $copy([Va,\ Vb],\ Vc)$ and $copy([Va,\ Vb],\ Vd)$, they, as a result, require more transformations than *Speq* rules use.

IV. Algorithm for generating *Speq* rules

A. Syntax of Speq rules

A *Speq* rule describes a procedure in which two variables included in an atom set are equalized due to predicate constraints (Refer to Section III-C). In this section, we show the

syntax of *Speq* rules. Let A be a set of atoms. Let $a \in A$. Then, arguments of A take a variable or a constant. Let $X \in Var(A)$ and $Y \in Var(A)$, where $X \neq Y$. A *Speq* rule is composed of an atom set A and an atom $eq(X, Y)$. The syntax of *Speq* rules is defined by the following formula.

$$A \Rightarrow eq(X, Y), A.$$

$\{app(Vx, [\,], Vy)\}$ is one of examples including variables and a constant in an atom a. A *Speq* rule for $\{app(Vx, [\,], Vy)\}$ has a procedure $Vx = Vy$ as follows:

$$r_{app} : app(Vx, [\,], Vy) \Rightarrow eq(Vx, Vy), app(Vx, [\,], Vy).$$

B. Outline of the proposed algorithm and its input and output

1) Outline of the algorithm: The proposed algorithm generates ET rules consisting of a head part and a replacement part. Let A be an atom set. Let X and Y be variables on $Var(A)$, where $X \neq Y$. The head part consists of an atom set A and the replacement part an atom set $A \cup \{eq(X, Y)\}$. That an ET rule contains $eq(X, Y)$ in its body part is important in embodying variables. If a logical formula $\forall(A \rightarrow eq(X, Y))$ is true, the following logical equivalence between atom sets is guaranteed.

$$\forall(A \leftrightarrow eq(X, Y), A)$$

Therefore, the following rewriting rule derived by the proposed algorithm is an ET rule [13].

$$A \Rightarrow eq(X, Y), A.$$

Given A, the proposed algorithm proves $X = Y$, and consequently an ET rule is generated. The algorithm then derives the empty set by successively transforming a clause set $\{yes \leftarrow A, noteq(X, Y)\}$ using a rule set.

2) Input and output : We show the definition of an input and an output of the proposed algorithm. An input is A, and an output is S, which is a set of all *Speq* rules obtained from the input. A *Speq* rule is composed of A and $eq(X, Y)$ and is defined as follows:

$$S = \{(A \Rightarrow eq(X, Y), A) \mid (X \in Var(A)) \ \& \ (Y \in Var(A)) \ \& \ (Y \neq X) \ \& \ (\forall(A \rightarrow eq(X, Y)))\}$$

If all conditions for S are not satisfied, then S is the empty set.

C. The proposed algorithm

When A is given, the proposed algorithm outputs S obtained by from A (Refer to Fig. 4). The algorithm is composed of the following eight processes.

[Algorithm for generating *Speq* rules]

1) derive A as an input (See Section IV-B.2).
2) Select variables X and Y, where $X \in Var(A)$, $Y \in Var(A)$ and $X \neq Y$.
3) Create a clause set cls as an initial state. cls is composed of A and $noteq(X, Y)$.
   $noteq(Arg1, Arg2)$ means that $Arg1$ is not equal to $Arg2$.
   $$cls = \{yes \leftarrow A, noteq(X, Y)\}$$
4) Replace cls using a rule r in a rule set $\mathbb{R}_t$.
   $\mathbb{R}_t$ consists of some ET rules and one Inductively-used rule (Refer to Section IV-D). Typical ET rules used by the proposed algorithm are general unfolding rules and specialization rules. Only ET rules can be applied to cls while Inductively-used rules can be applied to atoms obtained through replacement by unfolding rules. $trans(cls, r \in \mathbb{R}_t)$ used in Fig. 4 is defined as follows:
   $$trans(cls, r \in \mathbb{R}_t) = \mathbb{A} \cup \mathbb{B}$$
   $$\mathbb{A} = \{cl_a \mid (cl \in cls) \ \& \ (cl \text{ is applicable to } r) \ \& \ (cl_a \text{ is obtained from } cl \text{ applied to } r)\}$$
   $$\mathbb{B} = \{cl_b \mid (cl_b \in cls) \ \& \ (cl_b \text{ is not applicable to } r)\}$$
5) Add a *Speq* rule *rule* to a *Speq* rule set S.
   If cls is an empty set, then add *rule* to S and go to 7, else go to 6.
6) Select a rule $r \in \mathbb{R}_t$.
   If there exist rules applicable to a body atom set of a clause $cl(\in cls)$, then select one of them and go to 4. Otherwise, generate a new ET rule r, and then add it to $\mathbb{R}_t$. select r and go to 4.
7) Select other variables X and Y.
   If X and Y are selected, then go to 3, else go to 8.
8) Output S.

D. Rule set $\mathbb{R}_t$ used in generation process

$\mathbb{R}_t$ consists of some ET rules and one Inductively-used rule. While ET rules can be applied to any atoms, Inductively-used rules can be applied to only atoms that satisfy a certain condition. The atoms to which Inductively-used rules can be applied are a set of atoms obtained following the application of general unfolding rules. The control of rule application in the proposed algorithm is shown below.

1) Inductively-used rules should primarily be applied to a set of atoms to which general unfolding rules were applied previously.
2) For the other atoms, ET rules are applied.

1) ET rules : Typical ET rules used by the proposed algorithm are general unfolding rules and specialization rules. A general unfolding rule can be created by simple transformation of a clause which is the definition of a predicate. For example, we will show a general unfolding rule with respect to app predicate. app predicate is defined by the following two clauses.

$$app([\,], A, A) \leftarrow$$
$$app([A \mid B], C, [A \mid D]) \leftarrow app(B, C, D)$$

The following general unfolding rule is created based on the definition of app predicate.

$$app(Vx, Vy, Vz) \Rightarrow \{Vx = [\,], Vy = Vz\}; \Rightarrow \{Vx = [Vp \mid Vq], Vz = [Vp \mid Vw]\}, app(Vq, Vy, Vw).$$

This rule replaces a clause to which this rule can be applied with two clauses, one of which executes the computation of $Vx = [\,]$ and $Vy = Vz$, and the other replaces $app(Vx, Vy, Vz)$ with $app(Vq, Vy, Vw)$ after executing $Vx = [Vp \mid Vq]$ and $Vz = [Vq \mid Vw]$.

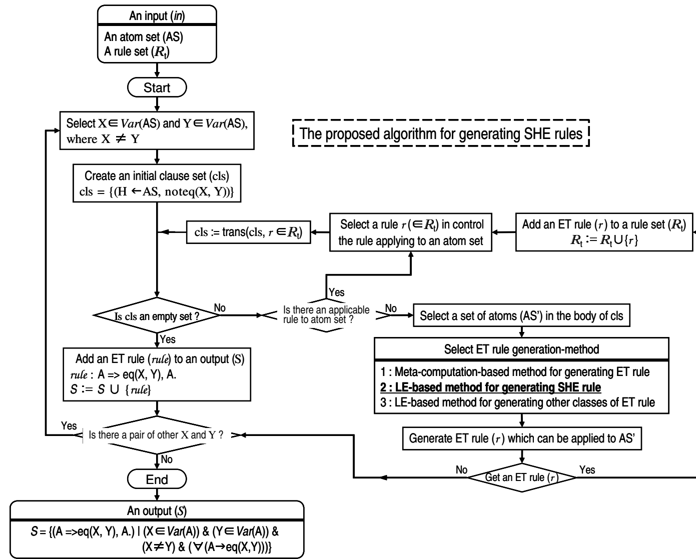

Fig. 4. Algorithm for generating *Speq* rules

A specialization rule describes a unification of arguments or the definition of a false condition. For example, basic ET rules on *eq* which describes a unification of arguments are as follows:

$$eq(Vx,\ [\ ]) \Rightarrow \{Vx = [\ ]\}.$$
$$eq([Vx \,|\, Vy],\ [Vz \,|\, Vw]) \Rightarrow eq(Vx,\ Vz),\ eq(Vy,\ Vw).$$

$$noteq(Vx,\ Vx) \Rightarrow \{false\}.$$
$$noteq([Vx \,|\, Vy],\ [Vx \,|\, Vz]) \Rightarrow noteq(Vy,\ Vz).$$

$false$ is a special built-in atom with the meaning that "a head atom of a rule does not satisfy a predicate constraint".

2) Inductively-used rules : An Inductively-used rule is created based on *Speq* rules generated by the proposed algorithm. By using the proposed algorithm, the following *Speq* rule is created.

$$A \Rightarrow eq(X,\ Y),\ A.$$

Since only the logical equivalence of this rule is guaranteed, using it as it is at this stage may cause an infinite loop. To avoid such a risk, a transformation of the rule based on the following terms is employed.

1) When a variable X is not equal to a variable Y, this ET rule is applicable to an atom set ($not(X == Y)$).
2) Move *eq* atom to the execution part, and define *eq* atom using a built-in description ($X = Y$).

A rule derived by such a transformation is Inductively-used rule, which is defined as follows.

$$A,\ \{not(X == Y)\} \Rightarrow \{X = Y\},\ A.$$

Let $app(Vx,\ [\ ],\ Vy)$ be an input atom set. Let $X(\in Var(app(Vx,\ [\ ],\ Vy)))$ be a variable Vx and $Y(\in Var(app(Vx,\ [\ ],\ Vy)))$ a variable Vy. Then, the following Inductively-used rule is created.

$$app(Vx,\ [\ ],\ Vy)\ \{not(Vx == Vy)\} \Rightarrow \{Vx = Vy\},\ app(Vx,\ [\ ],\ Vy).$$

V. Examples for generating *Speq* rules by the proposed algorithm

A. Generation of Speq rule on $\{app(Vx,\ [\ ],\ Vy)\}$

Based on the algorithm shown in Fig. 4, let us generate *Speq* rules for an atom set $app(Vx,\ [\ ],\ Vy)$. An input is $\{app(Vx,\ [\ ],\ Vy)\}$. $app(Arg1,\ Arg2,\ Arg3)$ means that the concatenation of a list $Arg1$ and a list $Arg2$ is a list $Arg3$. Since the second argument of $app(Vx,\ [\ ],\ Vy)$ is an empty list, it is not difficult to deduce that Vx is equivalent to Vy. In this example, we will present the process in which a set S, taking the following *Speq* rule r_{app} as its element, is output.

$$r_{app}\ :\ app(Vx,\ [\ ],\ Vy) \Rightarrow eq(Vx,\ Vy),\ app(Vx,\ [\ ],\ Vy).$$

[Selection of X and Y and creation of initial clause set cls]

First, select $X(\in Var(app(Vx,\ [\ ],\ Vy)))$ and $Y(\in Var(app(Vx,\ [\ ],\ Vy)))$, and then the initial clause set cls using A and $noteq(X,\ Y)$. Let X and Y be $X = Vx$ and $Y = Vy$, respectively. cls is created based on $app(Vx,\ [\ ],\ Vy)$ and $noteq(Vx,\ Vy)$.

$$cls = \{yes \leftarrow app(Vx,\ [\ ],\ Vy),\ noteq(Vx,\ Vy)\}$$

If an empty set is obtained by replacing cls using a rule set $\mathbb{R}_t$, then $\forall(app(Vx,\ [\ ],\ Vy) \rightarrow eq(Vx,\ Vy))$ can be proven.

[Initial rule set $\mathbb{R}_t$]

$\mathbb{R}_t$ consists of four rules, which are r_1, r_2 and r_3 representing ET rules and r_4 representing an Inductively-used rule, shown in Table II.

The meaning of each rule is as follows. r_1 replaces a clause with two clauses. These are a clause that executes $Vx = [\ ]$ and $Vy = Vz$ and one that replaces $app(Vx,\ Vy,\ Vz)$ with $app(Vq,\ Vy, Vw)$ by executing $Vx = [Vp \mid Vq]$ and $Vz = [Vp \,|\, Vw]$. r_2 means that $noteq(Vx,\ Vx)$ does not satisfy the constraints of *noteq* predicate. r_3 means that $noteq([Vx \,|\, Vy],\ [Vx \,|\, Vz])$ is replaced with $not(Vy,\ Vz)$. r_4 means that if

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

TABLE I
TRANSFORMATION SEQUENCE OF DEFINITE CLAUSES FOR THE RULE ON $\{app(Vx,\ [\ ],\ Vy)\}$

Name	Definite clauses representation	Rule applied
s_0	$\{yes \leftarrow \underline{app(Va,\ [\ ],\ Vb)},\ noteq(Va,\ Vb)\}$	r_1
s_1	$\{yes \leftarrow \underline{noteq([\ ],\ [\ ])}$	r_2
	$yes \leftarrow noteq([Va \mid Vb],\ [Va \mid Vc]),\ app(Vb,\ [\ ],\ Vc)\}$	
s_2	$\{yes \leftarrow noteq([Va \mid Vb],\ [Va \mid Vc]),\ \underline{app(Vb,\ [\ ],\ Vc)}\}$	r_4
s_3	$\{yes \leftarrow \underline{noteq([Va \mid Vb],\ [Va \mid Vb])},\ app(Vb,\ [\ ],\ Vb)\}$	r_2
s_4	$\{\ \}$	–

TABLE II
$\mathbb{R}_t$ FOR GENERATING *Speq* RULE FOR $\{app(Vx,\ [\ ],\ Vy)\}$

$$
\begin{aligned}
r_1 &: app(Vx,\ Vy,\ Vz) \Rightarrow \{Vx = [\ ],\ Vy = Vz\}; \\
&\qquad \Rightarrow \{Vx = [Vp \mid Vq],\ Vz = [Vp \mid Vw]\}, \\
&\qquad\ \ app(Vq,\ Vy,\ Vw). \\
r_2 &: noteq(Vx,\ Vx) \Rightarrow \{false\}. \\
r_3 &: noteq([Vx \mid Vy],\ [Vx \mid Vz]) \Rightarrow noteq(Vy,\ Vz). \\
r_4 &: app(Vx,\ [\ ],\ Vy)\ \{not(Vx == Vy)\} \Rightarrow \{Vx = Vy\},\ app(Vx,\ [\ ],\ Vy).
\end{aligned}
$$

Vx is not equal to Vy, then $app(Vx,\ [\ ],\ Vy)$ is replaced with $app(Vx,\ [\ ],\ Vy)$ by executing $Vx = Vy$.

[Presentation of *Speq* rule r_{app} generation process]

Table I shows the transformation process in which a set of definite clauses is successively transformed through an application of qualified rules in $\mathbb{R}_t$. The underlined atoms in each step are the ones to which a rule has been applied. An initial clause set (s_0) is finally transformed to an empty set (s_4). An empty set signifies that no answer which satisfies the initial clause set exists. Let us see the transformation process of a definite clause set. By applying an ET rule, an initial clause set *cls* is replaced with a singleton s_2 with one definite clause.

$$s_2 = \{yes \leftarrow noteq([Va \mid Vb],\ [Va \mid Vc]),\ \underline{app(Vb,\ [\ ],\ Vc)}\}$$

An Inductively-used rule r_4 is applied to $app(Vb,\ [\ ],\ Vc)$ in s_2. Since a definite clause set s_3 is obtained as a result, an ET rule r_2 is applied to $noteq([Va \mid Vb],\ [Va \mid Vb])$ and consequently an empty set is obtained. An empty set is obtained from *cls*, therefore, $\forall(app(Vx,\ [\ ],\ Vy) \rightarrow eq(Vx,\ Vy))$ can be proven. Therefore, the following *Speq* rule is generated.

$$app(Vx,\ [\ ],\ Vy) \Rightarrow eq(Vx,\ Vy),\ app(Vx,\ [\ ],\ Vy).$$

Because differing variables substituted for X and Y other than $X = Vx$ and $Y = Vy$, respectively, do not exist, the following set S is obtained as an output, where $X \in Var(app(Vx,\ [\ ],\ Vy))$ and $Y \in Var(app(Vx,\ [\ ],\ Vy))$.

$$S = \{app(Vx,\ [\ ],\ Vy) \Rightarrow eq(Vx,\ Vy),\ app(Vx,\ [\ ],\ Vy).\}$$

B. Generation of Speq rule on $\{rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$

In Section V-A, S was obtained without requiring generation of new ET rules. In this section, we will present an example where generation of new ET rules using the proposed method or meta-computation method ("select ET rule generation-method" in Fig. 4) is needed. An input is an atom set $\{rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$. $rev(Arg1,\ Arg2)$ means that a list $Arg2$ is elements of a list $Arg1$ in reverse order. It is not difficult, therefore, to deduce that a variable Vx is equivalent to a variable Vz. In this example, we will present the process in which a set S, taking the following *Speq* rule r_{rev} as its element, is output.

$$
\begin{aligned}
r_{rev}\ :\ & rev(Vx,\ Vy),\ rev(Vy,\ Vz) \\
& \Rightarrow eq(Vx,\ Vz),\ rev(Vx,\ Vy),\ rev(Vy,\ Vz).
\end{aligned}
$$

[Selection of X and Y, Creation of initial clause set *cls*]

Let $X = Vx$ and $Y = Vz$, where $X(\in (Var(rev(Vx,\ Vy)) \cup Var(rev(Vy,\ Vz))))$ and $Y(\in (Var(rev(Vx,\ Vy)) \cup Var(rev(Vy,\ Vz))))$. Let *cls* be the following formula. *cls* is composed of $rev(Vx,\ Vy)$, $rev(Vy,\ Vz)$ and $noteq(Vx,\ Vz)$.

$$cls = \{yes \leftarrow rev(Vx,\ Vy),\ rev(Vy,\ Vz),\ noteq(Vx,\ Vz)\}$$

If an empty set is obtained by replacing *cls* using $\mathbb{R}_t$, then $\forall(rev(Vx,\ Vy),\ rev(Vy,\ Vz) \rightarrow eq(Vx,\ Vz))$ is proven.

[Initial rule set $\mathbb{R}_t$]

$\mathbb{R}_t$ consists of five rules, which are r_1, r_2, r_3 and r_4 representing ET rules and r_5 representing an Inductively-used rule, shown in Table IV.

TABLE IV
$\mathbb{R}_t$ FOR GENERATING *Speq* RULE FOR $\{rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$

$$
\begin{aligned}
r_1 &: rev(Vx,\ Vy) \Rightarrow \{Vx = [\ ],\ Vy = [\ ]\}; \\
&\qquad \Rightarrow \{Vx = [Vp \mid Vq]\},\ rev(Vq,\ Vw),\ app(Vw,\ [Vp],\ Vy). \\
r_2 &: app(Vx,\ Vy,\ Vz) \Rightarrow \{Vx = [\ ],\ Vy = Vz\}; \\
&\qquad \Rightarrow \{Vx = [Vp \mid Vq],\ Vz = [Vp \mid Vw]\}, \\
&\qquad\ \ app(Vq,\ Vy,\ Vw). \\
r_3 &: noteq([Vx \mid Vy],\ [Vx \mid Vz]) \Rightarrow noteq(Vy,\ Vz). \\
r_4 &: noteq(Vx,\ Vx) \Rightarrow \{false\}. \\
r_5 &: rev(Vx,\ Vy),\ rev(Vy,\ Vz),\ \{not(Vx == Vz)\} \\
&\qquad \Rightarrow \{Vx = Vz\},\ rev(Vx,\ Vy),\ rev(Vy,\ Vz).
\end{aligned}
$$

The meaning of each rule is as follows. r_1 replaces a clause with two clauses. These are a clause that executes $Vx = [\ ]$ and $Vy = [\ ]$ and a clause that replaces $rev(Vx,\ Vy)$ with $rev(Vq,\ Vw)$ and $app(Vw,\ [Vp],\ Vy)$ by executing $Vx = [Vp \mid Vq]$. The meaning of r_2, r_3 and r_4 is the same as that of respective ET rules described in Section V-A. r_5 means that if Vx is not equal to Vz, then a set of $rev(Vx,\ Vy)$ and $rev(Vy,\ Vz)$ is replaced with a set of $rev(Vx,\ Vy)$ and $rev(Vy,\ Vz)$ by executing $Vx = Vz$.

TABLE III
TRANSFORMATION SEQUENCE OF DEFINITE CLAUSES FOR THE RULE ON $\{rev(Vx, Vy), rev(Vy, Vz)\}$

Name	Definite clauses representation	Rule applied
s_0	$\{yes \leftarrow rev(Va, Vb), rev(Vb, Vc), noteq(Va, Vc)\}$	r_1
s_1	$\{yes \leftarrow rev([\,], Va), noteq([\,], Va)$ $yes \leftarrow rev(Vc, Vd), noteq([Va \mid Vb], Vd), rev(Vb, Ve), app(Ve, [Va], Vc)\}$	r_1
s_2	$\{yes \leftarrow noteq([\,], [\,])$ $yes \leftarrow rev(Vc, Vd), noteq([Va \mid Vb], Vd), rev(Vb, Ve), app(Ve, [Va], Vc)\}$	r_4
s_3	$\{yes \leftarrow rev(Vc, Vd), noteq([Va \mid Vb], Vd), rev(Vb, Ve), app(Ve, [Va], Vc)\}$	r_1
s_4	$\{yes \leftarrow noteq([Va \mid Vb], [\,]), rev(Vb, Vc), app(Vc, [Va], [\,])$ $yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, Vf), app(Vf, [Va], [Vc \mid Vd]), rev(Vd, Vg), app(Vg, [Vc], Ve)\}$	r_2
s_5	$\{yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, Vf), app(Vf, [Va], [Vc \mid Vd]), rev(Vd, Vg), app(Vg, [Vc], Ve)\}$	r_2
s_6	$\{yes \leftarrow noteq([Va \mid Vb], Vc), rev(Vb, [\,]), rev([\,], Vd), app(Vd, [Va], Vc)$ $yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$	r_6
s_7	$\{yes \leftarrow noteq([Va], Vb), rev([\,], Vc), app(Vc, [Va], Vb)$ $yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$	r_1
s_8	$\{yes \leftarrow noteq([Va], Vb), app([\,], [Va], Vb)$ $yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$	r_2
s_9	$\{yes \leftarrow noteq([Va], [Va])$ $yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$	r_4
s_{10}	$\{yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$	r_2
s_{11}	$\{yes \leftarrow noteq([Va \mid Vb], [Vc]), rev(Vb\ [Vc \mid Ve]), rev(Vd, [\,]), app(Ve, [Va], Vd)$ $yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), rev(Vd, [Ve \mid Vh]), app(Vg, [Va], Vd), app(Vh, [Vc], Vf)\}$	r_6
s_{12}	$\{yes \leftarrow noteq([Va \mid Vb], [Vc]), rev(Vb, [Vc \mid Vd]), app(Vd, [Va], [\,])$ $yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), rev(Vd, [Ve \mid Vh]), app(Vg, [Va], Vd), app(Vh, [Vc], Vf)\}$	r_2
s_{13}	$\{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), rev(Vd, [Ve \mid Vh]), app(Vg, [Va], Vd), app(Vh, [Vc], Vf)\}$	r_7
s_{14}	$\{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), app(Vh, [Vc], Vf), rev(Vg, Vi), rev([Va], Vj),$ $app(Vj, Vi, [Ve \mid Vh])\}$	r_1
s_{15}	$\{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), app(Vh, [Vc], Vf), rev(Vg, Vi), app(Vj, Vi, [Ve \mid Vh]),$ $rev([\,], Vk), app(Vk, [Va], Vj)\}$	r_6
s_{16}	$\{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), app(Vh, [Vc], Vf), rev(Vg, Vi), app(Vj, Vi, [Ve \mid Vh]),$ $app([\,], [Va], Vj)\}$	r_2
s_{17}	$\{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), app(Vh, [Vc], Vf), rev(Vg, Vi), app([Va], Vi, [Ve \mid Vh])\}$	r_8
s_{18}	$\{yes \leftarrow noteq([Va \mid Vb], [Va \mid Ve]), rev(Vb, [Vc \mid Vf]), rev([Vc \mid Vf], Ve)\}$	r_3
s_{19}	$\{yes \leftarrow noteq(Vb, Ve), rev(Vb, [Vc \mid Vf]), rev([Vc \mid Vf], Ve)\}$	r_5
s_{20}	$\{yes \leftarrow noteq(Vb, Vb), rev(Vb, [Vc \mid Ve]), rev([Vc \mid Ve], Vb)\}$	r_4
s_{21}	$\{\,\}$	–

[Presentation of *Speq* rule r_{rev} generation process]

Table III shows the transformation process in which a set of definite clauses is successively transformed through an application of qualified rules in $\mathbb{R}_t$. Let us see the transformation process. By repeatedly applying ET rules one by one, *cls* is replaced by a singleton s_6 with two definite clauses.

$$s_6 = \{yes \leftarrow noteq([Va \mid Vb], Vc), rev(Vb, [\,]), rev([\,], Vd), app(Vd, [Va], Vc)\quad yes \leftarrow noteq([Va \mid Vb], Ve), rev(Vb, [Vc \mid Vf]), rev(Vd, Vg), app(Vg, [Vc], Ve), app(Vf, [Va], Vd)\}$$

Next, the following ET rule which can be applied to $rev(Vb, [\,])$ is generated by meta-computation method.

$$r_6 \ : \ rev(Vx, [\,]) \Rightarrow \{Vx = [\,]\}.$$

Since r_6 is added to a rule set $\mathbb{R}_t$, r_6 can be applied to $rev(Vb, [\,])$ in s_6. Consequently, a definite clause set s_7 is obtained. By repeatedly applying ET rules one by one, s_7 is replaced by a singleton s_{13} with one definite clause.

$$s_{13} = \{yes \leftarrow noteq([Va \mid Vb], [Ve \mid Vf]), rev(Vb, [Vc \mid Vg]), rev(Vd, [Ve \mid Vh]), app(Vg, [Va], Vd), app(Vh, [Vc], Vf)\}$$

The following ET rule which can be applied to the set of $rev(Vd, [Ve \mid Vh])$ and $app(Vg, [Va], Vd)$ is generated using the third Logical Equivalence (LE)-based method (Refer to Fig. 4).

$$r_7 \ : \ app(Vx, Vy, Vz), rev(Vz, Vw) \Rightarrow rev(Vx, Vp), rev(Vy, Vq), app(Vq, Vp, Vw).$$

Since r_7 is added to $\mathbb{R}_t$, r_7 can be applied to the set of $rev(Vd, [Ve \mid Vh])$ and $app(Vg, [Va], Vd)$ in a definite clause set s_{13}. Consequently, a definite clause set s_{14} is obtained. By repeatedly applying ET rules one by one, s_{14} is replaced by a singleton s_{17} with one clause.

$$s_{17} = \{yes \leftarrow noteq([Va \mid Vb],[Ve \mid Vf]), rev(Vb,[Vc \mid Vg]), app(Vh, [Vc], Vf), rev(Vg, Vi), app([Va], Vi, [Ve \mid Vh])\}$$

The following folding rule for *rev* predicate which can be applied to the set of $rev(Vg, Vi)$ and $app([Va], Vi, [Ve \mid Vh])$ is generated by meta-computation method.

$$r_8 \ : \ rev(Vx, Vy), app(Vy, [Vz], Vw) \Rightarrow rev([Vz \mid Vx], Vw).$$

Since r_8 is added to $\mathbb{R}_t$, r_8 can be applied to the conjunction of $rev(Vg, Vi)$ and $app([Va], Vi, [Ve \mid Vh])$ in a definite clause

set s_{17}. Consequently, a definite clause set s_{18} is obtained. An Inductively-used rule r_5 is applied to the conjunction of $rev(Vb,\ [Vc\,|\,Vf])$ and $rev([Vc\,|\,Vf],\ Ve)$ in a definite clause set s_{19}. Consequently, a definite clause set s_{20} is obtained. Finally, r_4 is applied to $noteq(Vb,\ Vb)$ in a definite clause set s_{20}, and consequently an empty set is obtained.

Since an empty set is obtained from *cls*, $\forall(rev(Vx,Vy),\ rev(Vy,Vz) \rightarrow eq(Vx,Vz))$ can be proven. Consequently, the following *Speq* rule is generated.

$$rev(Vx,\ Vy),\ rev(Vy,\ Vz) \Rightarrow eq(Vx,\ Vz),\ rev(Vx,\ Vy),\ rev(Vy,\ Vz).$$

Since a *Speq* rule cannot be obtained from a pair of X and Y other than a pair of $X = Vx$ and $Y = Vz$, the following set S is obtained as an output.

$$S = \{rev(Vx,\ Vy),\ rev(Vy,\ Vz) \Rightarrow eq(Vx,\ Vz),\ rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$$

VI. Discussions

A. Comparison of the proposed algorithm and current meta-computation method

This section discusses the following two topics based on the comparison of the proposed algorithm and a current meta-computation method. 1. The proposed algorithm can generate more *Speq* rules than a meta-computation method can do. 2. When a *Speq* rule with respect to a certain input is generated by the proposed algorithm, it is important that the form of the *Speq* rule is understood before rule generation.

The proposed algorithm is a special method for generating *Speq* rules. By specializing *Speq* rules, the proposed algorithm can generate more *Speq* rules than a meta-computation method can do. For example, we show the following four *Speq* rules ($r1$, $r2$, $r3$ and $r4$). The proposed algorithm can generate all the *Speq* rules, but a meta-computation method can generate only $r2$ and $r4$.

$$\begin{aligned}
r1 &: rev(Vx,\ Vy),\ rev(Vy,\ Vz) \\
&\qquad \Rightarrow eq(Vx,\ Vz),\ rev(Vx,\ Vy),\ rev(Vy,\ Vz). \\
r2 &: rev([Vx,\ Vy],\ Vz),\ rev([Vx,\ Vy],\ Vw) \\
&\qquad \Rightarrow eq(Vz,\ Vw),\ rev([Vx,\ Vy],\ Vz), \\
&\qquad\quad rev([Vx,\ Vy],\ Vw). \\
r3 &: app(Vx,\ [\ ],\ Vy) \Rightarrow eq(Vx,\ Vy),\ app(Vx,\ [\ ],\ Vy). \\
r4 &: app([Va\,|\,Vx],\ Vy,\ [Vb\,|\,Vz]) \\
&\qquad \Rightarrow eq(Va,\ Vb),\ app([Va\,|\,Vx],\ Vy,\ [Vb\,|\,Vz]).
\end{aligned}$$

The head of $r1$ and $r2$ is the conjunction of two *rev* atoms. $r2$ can be generated by a meta-computation method, but $r1$ cannot be generated by it. We explain the reason why the proposed method can generate more *Speq* rules than a meta-computation method can do. When *Speq* rules are generated by the proposed algorithm, a form of the *Speq* rules obtained ultimately is understood before *Speq* rule generation; a meta-computation, however, does not provide such projection. This gives the proposed algorithm a big advantage in generating *Speq* rules, i.e., a *Speq* rule obtained ultimately can be inductively used during the process of *Speq* rule generation in the proposed algorithm.

We will show the effectiveness where rules to be inductively derived can be used. If $r1$ is generated by a meta-computation method, then an atom set $\{rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$ is replaced to another one by a basic equivalent transformation and $r1$ is generated by a finite synthesis of the transformations. The length of lists applicable to Vx and Vy, respectively, is arbitrary. If the number of lists is fixed, the number of basic equivalent transformations is finite. However, since lists lengthen infinitely when generating $r1$, $r1$ cannot be generated by a finite basic equivalent transformation. Since rules to be finally derived are inductively used in the proposed algorithm, computation that needs an infinite transformation can be replaced with one that needs a finite transformation. Therefore, the proposed algorithm can generate *Speq* rules which need an infinite transformation.

B. Importance of Speq rules for generating other ET rules

In this section, we discuss the proposed algorithm in detail that it not only generates *Speq* rules efficiently but also expands its capability of generating other ET rules.

Although our interest is in the generation of *Speq* rules, we are also intrigued by the generation of other ET rules and, therefore, studying those as well. Also in the generation of other ET rules, a logical formula is proven using a rule set in order to guarantee the correctness of a logical equivalence. When an ET rule is generated, a *Speq* rule plays an important role. Therefore, the generation of *Speq* rules is essential.

For example, a *Speq* rule is necessary when generating the following ET rule.

$$app(Vx,\ Vy,\ Vz),\ rev(Vz,\ Vw) \Rightarrow rev(Vx,\ Vp),\ rev(Vy,\ Vq),\ app(Vq,\ Vp,\ Vw).$$

This ET rule is used when generating a *Speq* rule with respect to an atom set $\{rev(Vx,\ Vy),\ rev(Vy,\ Vz)\}$ (Refer to Section V-B). The above ET rule can be generated from the following logical equivalence.

$$\forall(\exists\{Vz\}(\{app(Vx,\ Vy,\ Vz),\ rev(Vz,\ Vw)\}) \leftrightarrow \exists\{Vp,\ Vq\}(\{rev(Vx,\ Vp),\ rev(Vy,\ Vq),\ app(Vq,\ Vp,\ Vw)\}))$$

A *Speq* rule required in the above rule generation is one with respect to an atom set $\{app(Vx,\ [\ ],\ Vy)\}$ (Refer to Section V-A). This *Speq* rule cannot be generated by a meta-computation method as shown in the previous section.

Therefore, it can be suggested that we have expanded the capability of generating other ET rules by proposing an algorithm by which *Speq* rules are generated.

VII. Conclusions

We focused on goal-oriented rule generation where the rule to be eventually obtained is designated, and proposed an algorithm for generating *Speq* rules. The proposed algorithm made it possible to efficiently generate *Speq* rules difficult to generate with the conventional rule-generation method. When an atom set is given as an input, the proposed algorithm outputs all *Speq* rules relating to the input atom set based on proof of a logical formula. By changing the logical formula, a

variety of ET rules can be generated. A distinctive feature of this algorithm is that it includes a rule obtained ultimately in a rule set used for proving a logical formula. When generating a *Speq* rule, all ET rules necessary for generating the *Speq* rule are found and successively generated by recursively using the proposed algorithm. Future works include development of new methods for efficiently generating other ET rules that are important for solving many problems.

References

[1] Akama,K. , Y.Shigeta and E.Miyamoto, Solving logical problems by equivalent transformation - a theoretical foundation, *Journal of the Japanese Society for Artificial Intelligence (in Japanese)*, vol.13, pp.928-935, 1998.

[2] Akama, K., E.Nantajeewarawat and H.Koike, Program generation in the equivalent transformation computation model using the squeeze method, *Proc. of PSI2006*, LNCS4378, Springer-Verlag Berlin Heidelberg, pp.41-54, 2007.

[3] Harada,M., T.Mizuno and S.Hamada, Executable C++ Program Generation form the Structured Object-oriented Design Diagrams, *Transactions of Information Processing Society of Japan (in Japanese)*, vol.40, no.7, pp.2988-3000, 1999.

[4] Higashino,T, M.Mori, T.Sugiyama, K.Taniguchi and T.Kasami, An algebraic specification of HDLC procedures and its verification, *The IEICE Transactions on Information and Systems (in Japanese)*, vol.SE-10, no.6, pp.825-836, 1984.

[5] Hopkins,J., Component primer, *Communications of the ACM*, vol.43, no.10 pp.27-30, 2000.

[6] IBM San Francisce, *Concepts and facilities*, IBM Corporation, 1977 Project, Software Development, vol.6, no.2, 1998.

[7] Ikeda,M., T.Nakamura, Y.Takata and H.Seki, Algebraic Specification of User Interface and Its Automatic Implementation, *The Special Interest Group Notes of IPSJ (in Japanese)*, vol.2001, no.114, pp.9-16, 2001.

[8] Jaffar,J. and M.Maher, Constraint logic programming, A survey, *Journal of Logic Programming*, vol.19/20, pp.503-581, 1994.

[9] Koike,H. , K.Akama and E.Boyd, Program synthesis by generating equivalent transformation rules, *Proc. of the 2nd International Conference on Intelligent Technologies*, Bangkok, Thailand, pp.250-259, 2001.

[10] Lloyd,J.W., *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.

[11] Lu,Yiguang, H.Awaya, H.Seki, M.Fujii and K.Ninomiya, On a Translation from Algebraic Specifications of Abstract Sequential Machines into Programs, *The IEICE Transactions on Information and Systems (in Japanese)*, vol.J73-D-I, no.2, pp.201-213, 1990.

[12] Mabuchi,H. , K.Akama and T.Wakatsuki, Equivalent transformation rules as components of programs, *International Journal of Innovative Computing, Information & Control*, vol.3, no.3, pp.685-696, 2007.

[13] Miura,K. , K.Akama and H.Mabuchi, Creation of ET Rules from Logical Formulas Representing Equivalent Relations, *International Journal of Innovative Computing, Information & Control*, vol.5, (no.4 or no.5), 2009 (to appear).

[14] Nishida,Y. , K.Akama and H.Koike, An experimental program-generation system based on meta-computation, *Technical report of the Institute of Electronics, Information and Communication Engineers (in Japanese)*, Matsue, Japan, pp.31-36, 2007.

[15] Sterling,L. and E.shapiro, The Art of Prolog: Advanced Programming Techniques, *MIT Press, second edition*,

[16] Szyperski,C., *Component software: Beyond object-oriented programming*, Addison-Wesley, 1999.

[17] van Hentenryck,P., Constraint logic programming, *The Knowledge Engineering Review*, vol.6, no.3, pp.151-194, 1991.

[18] Wakatsuki,T., K.Akama and H.Mabuchi, A Framework for Synthesizing low-level Imperative Programs From Deterministic Abstract Programs, *Technical report of the Institute of Electronics, Information and Communication Engineers (in Japanese)*, vol.107, no.392, pp.37-42, 2007.