Synthesis of Conversion Rules by Expanding Logic Programs

Hiroshi Mabuchi[1], Kiyoshi Akama[2], and Hidekatsu Koike[2]

[1] Iwate Prefectural University, 152-52 Sugo Takizawa Iwate 020-0173 Japan
mabu@soft.iwate-pu.ac.jp
[2] Hokkaido University, Kita 11, Nishi 5, Kita-ku, Sapporo, 060-0811, Japan
{akama, koke}@cims.hokudai.ac.jp

Abstract. In some cases a problem can not be solved effectively and automatically in only the space of a certain knowledge representation. We propose in this paper a natural and efficient method for solving a problem by expanding the space using the characteristics of the space in which the problem can not be solved. We deal with synthesis of conversion rules that simplify logical circuits as a concrete problem, and by comparing the synthesis of conversion rules in the space before expansion (space of logic programs) with that in the space after expansion, we show that expansion of logic programs is efficient for synthesis of conversion rules.

1 Introduction

In the field of artificial intelligence, various knowledge representations, such as semantic networks, production systems, logic, and frames, have been used, and each of these knowledge representations has been shown to be superior for certain types of problems.

However, in some cases a problem can not be solved effectively and automatically in only the space of a certain knowledge representation. We therefore consider how to solve such problems effectively and automatically. As a solution, we propose in this paper a natural and efficient method for solving a problem by expanding the space using the characteristics of the space in which the problem can not be solved.

We deal with synthesis of conversion rules that simplify logical circuits as a concrete problem, and by comparing the synthesis of conversion rules in the space before expansion (space of logic programs [8]) with that in the space after expansion, we show that expansion of logic programs is efficient for synthesis of conversion rules. Synthesis of conversion rules requires a new conversion rule (i.e., synthesis rule), which is obtained by combining two or more existing successive conversion rules to achieve a new conversion from one state to another [1–4, 6]. In the space after expansion, various data structures, including strings and multisets as well as terms, can be treated.

Synthesis of conversion rules is performed by equivalent transformation [7]. A program is converted into another equivalent program by an equivalent transformation. The expansion of logic programs is performed by applying an equivalent transformation rule and enables a useful synthesis rule for problem solving to be obtained efficiently and automatically and at a low cost.

2 Outline of Synthesis by Expanding Logic Programs

In problem solving, a state is converted into another state by a conversion rule. The ability to solve a problem thus depends on the conversion rules. In general the more applicable conversion rules there are, the higher the possibility of efficient conversion will be. Therefore, when useful synthesis rules are added to existing conversion rules, the ability to solve a problem can be enhanced. Thus, obtaining useful synthesis rules is important for problem solving.

The concept in synthesis of conversion rules by expanding logic programs is shown in Fig.1. Let Γ_1 be the space of logic programs and Γ_2 the space after expansion. A is a program before synthesizing conversion rules. Program B including a synthesis rule is obtained from A. However, if this synthesis rule is not useful for problem solving, program D including a useful synthesis rule must be obtained in Γ_1. When automatically obtaining D from B is difficult, we must consider other methods to obtain a useful synthesis rule. The problem, however, is the cost of change. Methods such as not changing the way to synthesize or not changing the knowledge representation could be considered to reduce this cost. We therefore propose a method for solving a problem by expanding Γ_1 using the characteristics of Γ_1.

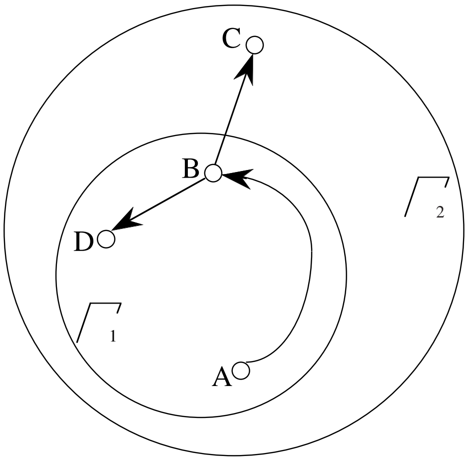

Fig. 1. Synthesis by Expanding Logic Programs

Data structures of an object domain are characterized by a mathematical structure called a *specialization system* [11], and a program is defined on a specialization system. A specialization system prescribes the space of programs and determines the objects that are treated in each space. By prescribing a specialization system, Γ_1 and Γ_2 can be made.

When a synthesis rule in B obtained in Γ_1 is not useful for problem solving and automatically obtaining a useful synthesis rule that is included in D of Γ_1 is difficult, Γ_2 must be established so that a program C including a useful synthesis rule from B can be obtained efficiently and automatically. To achieve this, we expand Γ_1 to allow treatment of various data structures including multisets and constraints.

Synthesis of conversion rules is performed by equivalent transformation. The transformation from Γ_1 into Γ_2 is performed by an equivalent transformation rule that changes representation of Γ_1 into that of Γ_2.

As mentioned above, a program A can be transformed efficiently and automatically into C through B by equivalent transformations.

3 Conversion Program and Synthesis of Conversion Rules

3.1 Conversion Program

We show the conversion program.

C_1 : `path([Rule|Path],A,C)`$\leftarrow$
```
    arc(Rule,A,B),
    path(Path,B,C).
```
C_2 : `path([],A,A)`$\leftarrow$.

Hereafter, C_1,C_2, $\cdots$ are running names of clauses.

The predicate **path** has the form of

path (sequence of conversion rules p, state 1, state 2),

and this means that "state 1 is converted into state 2 by p".

The predicate **arc** has the form of

arc (conversion rule q, state 1, state 2),

and this means that "state 1 is converted into state 2 by q".

The conversion program consists of C_1, C_2 and several arc clauses (conversion rules), which have a predicate arc in the head.

3.2 Synthesis of Conversion Rules by Equivalent Transformation

Assume that two conversion rules, r_1 and r_2, are defined as

C_a : arc $(r_1 \ , S_1 \ , S_2) \leftarrow body_1$.

C_b : arc $(r_2 \ , S_3 \ , S_4) \leftarrow body_2$.

For example, C_a means that a state S_1 is converted into another state S_2 by a conversion rule r_1 under the condition of $body_1$.

The synthesis of r_1 and r_2 requires the following new conversion rule, r_3, by which the state that can be reached by successively utilizing the two conversion rules r_1 and r_2 can be reached by one conversion.

C_c : arc $(r_3 \ , S_5 \ , S_6) \leftarrow body$.

Then, due to the influence of unification of S_2 and S_3, S_1 and S_4 are changed into S_5 and S_6, respectively. $body_1$ and $body_2$ are also changed due to the influence of this unification. *body* is a sequence of changed $body_1$ and $body_2$.

We introduce the following clause to check the effect when r_1 and r_2 are used in order.

C_0 : reach$(x\ ,\ y) \leftarrow$ path$([r_1\ , r_2]\ , x\ , y)$.

From the above, a synthesis rule can be obtained as follows.

(1) Transform equivalently $P_a = \{C_0,\ C_1,\ C_2,\ C_a,\ C_b\}$.

(2) First transform equivalently C_0, and by further equivalent transformations,
$P_b = \{d,\ C_1,\ C_2,\ C_a,\ C_b\}$

can be obtained, where d is a clause that does not include arc predicates in the body as

d : reach $(S_5\ ,\ S_6) \leftarrow$ *body*.

(3) Using S_5, S_6, and *body*, a synthesis rule C_c can be obtained.

4 Synthesis in the Space of Logic Programs

4.1 Conversion Rules

In this paper, we deal with synthesis of conversion rules that simplify logical circuits as a concrete problem. From a characteristic of logical circuit that there exist a lot of elements that have n inputs having no order, the notion of multiset is required. A logical circuit consists of many multisets.

However, a logic program has no multisets. Lists are therefore used instead of multisets. A difference between list and multiset is whether the order is thought or not. A predicate member_rest (described later in this section) is used to realize multisets with lists in logic programs.

In this section, we describe two conversion rules (an "andAnd rule" and a "noConnection rule"), from which a new rule is synthesized later, from among many conversion rules in logic programs.

The andAnd rule means that "when an output terminal of an *and element* is an input terminal of another *and element*, one new *and element* can be obtained by connecting these two *and elements*.

C_3 :
```
arc(andAnd,Circuit1,Circuit2)←
    member_rest([and,E,IN],Circuit1,RestCircuit),
    member_rest([and,D,R1],RestCircuit,RR),
    member_rest(D,IN,R2),
    member_rest([and,E,R3],Circuit2,RestCircuit),
    union(R1,R2,R3).
```

The first argument of the predicate arc is the name of the conversion rule, the second argument is a logical circuit before conversion, and the third argument is a logical circuit after conversion. For example, [and,E,IN] is an *and element*, and this element has n inputs (IN) and one output (E).

A predicate member_rest(X, Y,Z) means that "Z is a list that is obtained by removing an element X from a list Y", and its program is as follows.

C_4 : `member_rest(A,[A|R],R)←.`
C_5 : `member_rest(A,[X|R],[X|RR])←`
`member_rest(A,R,RR).`

A predicate union(X,Y,Z) – union of multisets – means that "$X \cup Y = Z$", and its program is as follows.
C_6 : `union([X|U],Y,[X|V])←union(U,Y,V).`
C_7 : `union([],Y,Y)←.`

Since IN and R1 are multisets, R2 obtained by removing D from IN is a multiset. R3 – a union of R1 and R2 – is also a multiset.

The noConnection rule means that "when a terminal of an element is not connected to a terminal of another element, the element can be removed from the logical circuit" and is represented in the space of logic programs as follows.

C_8 : `arc(noConnection,Circuit,RR1)←`
```
    member_rest([ELEMENT,AA,P],Circuit,RR1),
    notExist(AA,RR1),
    free(AA).
```

A predicate notExist(AA,RR1) means that AA is not used as a terminal in logical circuit RR1, and free(AA) means that AA is not connected to another terminal. Let C_9 be a clause of notExist and C_{10} a clause of free. Since C_9 and C_{10} are not unfolded, these programs are omitted.

4.2 Example of Synthesis by Equivalent Transformation

In this section, we show an example of synthesis by equivalent transformation in the space of logic programs. We first introduce the following clause to call a predicate newarc, which is a predicate of the head in a clause that is obtained by synthesis.

C'_1 : `path([Rule|Path],A,C)←`
```
    newarc(Rule,A,B),
    path(Path,B,C).
```

A predicate **newarc** has the form of
newarc (synthesis rule r, state 1, state 2),
and this means that "state 1 is converted into state 2 by r".

In the following, the synthesis of the andAnd rule and the noConnection rule by equivalent transformation is shown.

A program P_0 is defined as follows.
$P_0 = \{C_1, C'_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}\}$

Then, a program P_1 is obtained by adding the following clause C_{11}; i.e., $P_1 = P_0 + \{C_{11}\}$. P_1 corresponds to A in Fig.1.

C_{11} : `newarc(andAndnoConnection,A,C)←`
```
        path([andAnd,noConnection],A,C).
```

P_1 is transformed as $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6$. This process is shown in Fig.2. P_6 corresponds to B in Fig.1.

Since P_0 is left unchanged and C_{11} is transformed by only unfolding as $C_{11} \to C_{12} \to C_{13} \to C_{14} \to C_{15} \to C_{16}$, each transformation from P_1 to P_6 is equivalent. C_{16} is a synthesis rule. C_{12},C_{13},C_{14},C_{15},C_{16} in Fig.2 are as follows.

selected clause	matching head	generated clause	Program
			$P_1 = P_0 \cup \{C_{11}\}$
body of C_{11}	head of C_1	C_{12}	equivalence
			$P_2 = P_0 \cup \{C_{12}\}$
path of body of C_{12}	head of C_1	C_{13}	equivalence
			$P_3 = P_0 \cup \{C_{13}\}$
path of body of C_{13}	head of C_2	C_{14}	equivalence
			$P_4 = P_0 \cup \{C_{14}\}$
first arc of body of C_{14}	head of C_3	C_{15}	equivalence
			$P_5 = P_0 \cup \{C_{15}\}$
arc of body of C_{15}	head of C_8	C_{16}	equivalence
			$P_6 = P_0 \cup \{C_{16}\}$

Fig. 2. Synthesis in the Space of Logic Programs

C_{12} :
```
newarc(andAndnoConnection,A,Z)←
    arc(andAnd,A,B),
    path([noConnection],B,Z).
```

C_{13} :
```
newarc(andAndnoConnection,A,Z)←
    arc(andAnd,A,B),
    arc(noConnection,B,C),
    path([],C,Z).
```

C_{14} :
```
newarc(andAndnoConnection,A,C)←
    arc(andAnd,A,B),
    arc(noConnection,B,C).
```

C_{15} :
```
newarc(andAndnoConnection,Circuit1,C)←
    member_rest([and,E,IN],Circuit1,RestCircuit),
    member_rest([and,D,R1],RestCircuit,RR),
```

```
        member_rest(D,IN,R2),
        member_rest([and,E,R3],Circuit2,RestCircuit),
        union(R1,R2,R3),
        arc(noConnection,B,C).
```

C_{16} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
        member_rest([and,E,IN],Circuit1,RestCircuit),
        member_rest([and,D,R1],RestCircuit,RR),
        member_rest(D,IN,R2),
        member_rest([and,E,R3],Circuit2,RestCircuit),
        union(R1,R2,R3),
        member_rest([ELEMENT,AA,P],Circuit2,RR1),
        notExist(AA,RR1),
        free(AA).
```

This synthesis process corresponds to the conversion from A to B in Fig.1, and C_{16} is contained in B. If C_{16} is useful, the logical circuit will be efficiently simplified by C_{16}. However, C_{16} is simply a combination of the body of C_3 and that of C_8. By calling one clause C_{16} instead of calling two clauses C_3 and C_8, execution time is slightly decreased, but there are no other merits of the synthesis. Therefore, C_{16} is not useful.

4.3 Improvement of Synthesis Rule

Since C_{16} is not useful for simplifying a logical circuit, a useful synthesis rule (C_{17}) that is included in program D of Fig.1 should be required.

C_{17} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
        member_rest([and,E,IN],Circuit1,RestCircuit),
        member_rest(D,IN,R2),
        union(R1,R2,R3),
        member_rest([and,D,R1],RestCircuit,RR),
        member_rest([and,E,R3],RR1,RR),
        notExist(D,RR1),
        free(D).
```

In this connection, C_{17} is the same as C_{19}, which is a useful synthesis rule corresponding to a given example in section 5.1.

In the following, we consider to look for C_{17}. To save space, we change the names of variables in C_{16} as follows.

C'_{16} :
```
newarc(andAndnoConnection,X14,X13)←
        member_rest([and,X4,X7],X14,X15), ...(1)
        member_rest([and,X2,X1],X15,X5),  ...(2)
        member_rest(X2,X7,X3),            ...(3)
        member_rest([and,X4,X8],X12,X15), ...(4)
        union(X1,X3,X8),
        member_rest([X9,X10,X11],X12,X13),...(5)
```

```
notExist(X10,X13),
free(X10).
```

Since there are five member_rest literals in the body, we consider to reduce the number. We therefore consider a transformation in the conjunction of member_rest literals in the body. For example, suppose that the following possible transformation is selected.

T_1 :

```
{member_rest(Y1,Y2,Y3),    ...(a)
 member_rest(Y4,Y5,Y2),    ...(b)
 member_rest(Y1,Y5,Yans)} ...(c)

              ↓

{member_rest(Y1,Y2,Y3),    ...(d)
 member_rest(Y4,Yans,Y3)} ...(e)
```

T_1 means to transform {(a),(b),(c)} into {(d),(e)}.

In order to apply T_1 to the body of C'_{16}, we look for combinations of member_rest literals matching {(a),(b),(c)} from the body of C'_{16}. There exist five member_rest literals in the body of C'_{16}. Accordingly, we search for combinations of three member_rest literals from the five member_rest literals. There are ten possible combinations. From among these combinations, the only combination matching {(a),(b),(c)} is (a) = (2),(b) = (4),(c) = (5). By this combination, the first argument [X9,X10,X11] of (5) is changed to [and,X2,X1] as follows.

```
member_rest([and,X2,X1],X12,X13) ...(5')
```

Therefore, the following transformation is performed.

```
{member_rest([and,X2,X1],X15,X5),
 member_rest([and,X4,X8],X12,X15),
 member_rest([and,X2,X1],X12,X13)}

                   ↓

{member_rest([and,X2,X1],X15,X5),
 member_rest([and,X4,X8],X13,X5)}
```

By applying this transformation to C'_{16} and rewriting to the original variables, C_{17} can be successfully obtained.

4.4 Correctness of Transformation

We try to prove the correctness of T_1 theoretically. Let (a),(b),(c),(d),(e) be the following relations, respectively.

$$
\begin{aligned}
\{\mathrm{Y1}\} \cup \mathrm{Y3} &= \mathrm{Y2} \quad ...(a)\\
\{\mathrm{Y4}\} \cup \mathrm{Y2} &= \mathrm{Y5} \quad ...(b)\\
\{\mathrm{Y1}\} \cup \mathrm{Yans} &= \mathrm{Y5} \quad ...(c)\\
\{\mathrm{Y1}\} \cup \mathrm{Y3} &= \mathrm{Y2} \quad ...(d)\\
\{\mathrm{Y4}\} \cup \mathrm{Y3} &= \mathrm{Yans} \quad ...(e)
\end{aligned}
$$

When (a) is substituted for the left side of (b), (b) changes as follows.

$$\{\mathrm{Y4}\} \cup \{\mathrm{Y1}\} \cup \mathrm{Y3} = \mathrm{Y5} \quad ...(f)$$

When (c) is substituted for the right side of (f), (f) changes as follows.

$$\{\mathrm{Y4}\} \cup \{\mathrm{Y1}\} \cup \mathrm{Y3} = \{\mathrm{Y1}\} \cup \mathrm{Yans} \quad ...(g)$$

Then, under the condition that Y1 is contained neither in {Y4} nor in Y3 or Yans, (e) is obtained by removing Y1 from both sides of (g). (e) is therefore correct under this condition.

However, since it is possible that Y1 is contained either in any one or in any two or all three of {Y4}, Y3, and Yans, T_1 is not correct; i.e., T_1 is not an equivalent transformation. It is difficult to consider other transformations which replace T_1.

5 Example of Synthesis by Expanding Logic Programs

5.1 Synthesis by Expanding Logic Programs

Since a logic program has no multisets, lists are used instead of multisets. A predicate member_rest is then used to realize multisets with lists.

In contrast, the space after expansion allows use of multisets, the following rule can be therefore described.

T_2 :

```
member_rest(X,Y,Z)
        ↓
equal(Y,{X|Z})
```

where equal means Y = {X | Z}, and {X | Z} is a multiset.

We look for a useful synthesis rule in the space after expansion (Γ_2) by applying an equivalent transformation rule (T_2) to C_{16}. This process corresponds to the transformation from B to C in Fig.1.

By applying T_2 to C_{16}, the following clause C''_{16} is obtained. Then, in the space after expansion, a logical circuit can be represented as a set of multisets.

C''_{16} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
    equal(Circuit1,{[and,E,IN]|RestCircuit}),
    equal(RestCircuit,{[and,D,R1]|RR}),
    equal(IN,{D|R2}),
    equal(Circuit2,{[and,E,R3]|RestCircuit}),
```

```
        union(R1,R2,R3),
        equal(Circuit2,{[ELEMENT,AA,P]|RR1}),
        notExist(AA,RR1),
        free(AA).
```

C''_{16} is further transformed as follows.

C'''_{16} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
        equal(Circuit1,{[and,E,{D|R2}],[and,D,R1]|RR}),
        equal(Circuit2,{[and,E,R3],[and,D,R1]|RR}),
        union(R1,R2,R3),
        equal(Circuit2,{[ELEMENT,AA,P]|RR1}),
        notExist(AA,RR1),
        free(AA).
```

Here, {[and,E,R3],[and,D,R1] | RR} is unified with {[ELEMENT,AA,P] | RR1}. Then, the unifications could be as follows:

(1) `[and,E,R3]`$\theta =$ `[ELEMENT,AA,P]`σ
 `{[and,D,R1]|RR}`$\theta =$ `RR1`σ
(2) `[and,D,R1]`$\theta =$ `[ELEMENT,AA,P]`σ
 `{[and,E,R3]|RR}`$\theta =$ `RR1`σ
(3) `RR`$\theta =$ `{[ELEMENT,AA,P]|RR'}`σ
 `{[and,D,R1],[and,E,R3]|RR'}`$\theta =$ `RR1`σ

Here, RR' is the circuit that can be obtained by removing [ELEMENT,AA,P] from RR. From the unifications of (1),(2), and (3), the following three synthesis rules can be obtained.

C_{18} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
      union(R1,R2,R3),
      notExist(E,{[and,D,R1]|RR}),
      free(E).
```
where
```
Circuit1={[and,E,{D|R2}],[and,D,R1]|RR},
RR1={[and,D,R1]|RR}.
```

C_{19} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
      union(R1,R2,R3),
      notExist(D,{[and,E,R3]|RR}),
      free(D).
```
where
```
Circuit1={[and,E,{D|R2}],[and,D,R1]|RR},
RR1={[and,E,R3]|RR}.
```

C_{20} :
```
newarc(andAndnoConnection,Circuit1,RR1)←
```

```
union(R1,R2,R3),
notExist(AA,{[and,E,R3],[and,D,R1]|RR'}),
free(AA).
```

where

```
Circuit1={[and,E,{D|R2}],[and,D,R1],{[ELEMENT,AA,P]|RR'}},
RR1={[and,E,R3],[and,D,R1]|RR'}.
```

5.2 Synthesis Rule Corresponding to Given Examples

By giving an example, one useful synthesis rule corresponding to the given example is further obtained automatically from C_{18}, C_{19}, and C_{20} [9, 10].

For example, we give the following examples:

(1) Example for Circuit1:

$\{[\mathrm{and}\,,\mathrm{o}\,,\{\mathrm{mp1}\,,\mathrm{mp2}\}]\,,[\mathrm{and}\,,\mathrm{mp2}\,,\{\mathrm{mq1}\,,\mathrm{mq2}\}]\}$.

(2) Example for RR1:

$\{[\mathrm{and}\,,\mathrm{o}\,,\{\mathrm{mq1}\,,\mathrm{mq2}\,,\mathrm{mp1}\}]\}$.

Here, o, mp1, mq1, etc. are input or output terminals.

The examples of (1) and (2) are substituted for C_{19}. Then, there exists a substitution

$$\{\mathrm{D}\,/\,\{\mathrm{mp2}\},\mathrm{R1}\,/\,\{\mathrm{mq1}\,,\mathrm{mq2}\},\mathrm{E}\,/\,\mathrm{o},\quad \mathrm{R2}\,/\,\{\mathrm{mp1}\},\mathrm{RR}\,/\,(),\mathrm{R3}\,/\,\{\mathrm{mq1}\,,\mathrm{mq2}\,,\mathrm{mp1}\}\},$$

and this substitution is also true concerning the body. As for C_{18} and C_{20}, there exists no such substitution. Therefore, C_{19} is a useful synthesis rule corresponding to the given example.

5.3 Effect of Synthesis by Expanding Logic Programs

In the synthesis by expanding logic programs, three synthesis rules were automatically obtained. Therefore, a synthesis rule C_{16} in the space of logic programs tacitly contains information on these three synthesis rules. However, as described in sections 4.3 and 4.4, it is not easy to even take out one synthesis rule C_{17} from C_{16}.

In contrast, in the space after expansion, by giving an example, a useful synthesis rule (C_{19}) corresponding to a given example can be obtained efficiently and automatically. In this connection, C_{19} is the same as C_{17}.

As for the representation of a rule, in the space of logic programs, since the majority of a logical circuit is represented in the body, the body is long and it is difficult to understand connection of the circuit. Moreover, since the body is unfolded by equivalent transformation, the processing cost is high when the body is long. In contrast, in the space after expansion, since the majority of a logical circuit is represented in the head as multisets, the body is short and it is easy to understand connection of the circuit.

6 Conclusions

We proposed a method for obtaining a useful synthesis rule by expanding logic programs.

In the space of logic programs, it is difficult to automatically obtain a useful synthesis rule. However, several synthesis rules can be obtained by expanding logic programs, and by giving an example, one useful synthesis rule corresponding to the given example can be obtained from these synthesis rules.

In the present method, since knowledge representation is expanded using the characteristics of the space of logic programs, a useful synthesis rule can be efficiently obtained at a low cost.

References

1. Dix, J. and Brass, S. : Semantics of Logic Programs Based on Partial Evaluation. Journal of Logic Programming 40(1) (1999) 1–46
2. Pettorossi, A. and Proietti, M. : Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs. Journal of Logic Programming Vol.41, N.2&3 (1999) 197–230
3. Vere, S.: Relational Production Systems. Artificial Intelligence 8 (1977) 47–68
4. Buchberger, B.: History and Basic Features of the Critical-Pair / Completion Procedure. J. Symbolic Computation, Vol.3 (1987) 3–38
5. Hashida, K. and Ishizaki, S.: Dependency Propagation: A Unified Theory of Sentence Comprehension and Generation. Proceedings of the 10th IJCAI (1987) 664–670
6. Takeuchi, A. and Fujita, H.: Competitive Partial Evaluation. Workshop on Patial Evaluation and Mixed Computation (1987) 317–326
7. Tamaki, H. and Sato, T.: Unfold/fold Transformation of Logic Programs. Proc. of 2nd ILPC (1984) 127–138
8. Lloyd, J. W.: Foundations of Logic Programming. Second Edition, Springer-Verlag (1987)
9. Dejong, G. and Mooney, R.: Explanation-Based Learning: An Alternative View. Machine Learning, vol.1. No.2 (1986) 145–176
10. Mitchell, T. M., Keller, R., and Kedar-Cabelli, S.: Explanation-Based Generalization: A unifying view. Machine Learning, vol.1. No.1 (1986) 47–80
11. Akama, K.: Declarative Computation Model. Information Processing, SIG-programming, 12-10-PRG (1993) 87–94