INCORPORATING GENERALIZATION HEURISTICS

INTO VERIFICATION OF PROLOG PROGRAMS

Hirohisa SEKI

Mitsubishi Electric Corporation
Central Research Laboratory
Tsukaguchi-Honmachi 8-1-1
Amagasaki,Hyogo,JAPAN 661

ABSTRACT

This paper is concerned with the problem of generalization heuristics, which are incorporated into our verification system for Prolog programs. Two kinds of generalization are discussed, that is, a *mechanical generalisation* and an *intelligent generalization.* We show that the mechanical generalization used in Boyer-Moore*s theorem prover (BMTP) can be performed by the simplification rule of our verification system, as well as in the case of cross-fertilization. To the intelligent generalization heuristic, which is not employed in BMTP, we give a generalization scheme which is naturally incorporated into our inference system of the extended-execution style of the Prolog interpreter, and which proves to be effective also for flawed induction schemes.

I. INTRODUCTION

This paper is concerned with a verification system for Prolog programs which is currently under development, as one of the subprojects of the FGCS "Intelligent Programming System" [1].

Logic programming is often advocated as a desirable choice for the verification problem because of its clear semantics (e.g., [2]). In the design of our verification system, we have tried to take advantage of Prolog's characteristics and present first order inference in an extended execution style of a Prolog interpreter (3). Not only first order inference but induction is indispensable as a means of proving interesting properties of Prolog programs such as termination and correctness. In the case of functional language, Boyer-Moore's theorem prover (BMTP) [4] is famous for its automatic application of induction and many sophisticated heuristics constructing inductive proofs. Into our verification system, various kinds of heuristics have been integrated, most of which are inspired by BMTP and are developed to suit the verification of Prolog programs (e.g-,[5]).

In this paper, special attention is paid to "Generalization Heuristics" which are applied when a theorem to be proved is too weak and it is necessary and easier to prove a theorem that is stronger than the original weak one. We deal with two kinds of generalization heuristics. The first generalization heuristic (we call it *mechanical generalisation* in this paper) corresponds to the one used in BMTP, and the second (called intelligent *generalisation)* is one that is not employed in BMTP but is left to the user. Some work has been done with respect to functional language to mechanize the intelligent generalisation heuristic (e.g.,[6],[7]). Our main purpose lies in an attempt to clarify how these two kinds of generalization heuristics can be incorporated into the verification of Prolog programs.

After summarizing some preliminary materials in section II, we present our verification methods for Prolog programs in section III. In section IV, we explain two kinds of generalization heuristics ; first, we illustrate that the mechanical generalization is naturally incorporated by the simplification rule of our verification system ; secondly, we show some examples which cannot be proved by simply applying first order inference and induction, and therefore motivate us to incorporate the intelligent generalization heuristic. In section V, we discuss in detail how the intelligent generalization is performed in the framework of our verification system of Prolog programs. Lastly in section VI, we discuss the relation to other work and some implementation issues.

II. FORMULATION OF VERIFICATION

In this section, we give our formulation of the verification of Prolog programs. In the following, we follow the syntax of DEC-10 Prolog [8] and assume some familiarity to the terminology of Prolog and first order logic(see e.g.,[9], [10]).

In our verification system, we treat only pure Prolog, i.e., we impose the following 3 conditions on Prolog programs :

(i) no negative clauses and no "not" in programs.
(ii) no executable primitives.
(iii) no "cut" symbol (!).

On the other hand, our specification language is a subclass of first order logic formulas. That is, a specification is expressed by a closed formula which can be transformed into the following prenex form (we call it an *S-formula*) :

$$\forall X_1 \cdots \forall X_m \exists Y_1 \cdots \exists Y_n \; F \quad (m, n \geq 0),$$

where F contains no quantifiers.

Let S be a specification and P be a Prolog program and P^* be the completion of P in the sense of Clark [11]. Then we adopt a formulation that *verification* of S with respect to P proves that $P^* \vdash S$. This means that S is a logical consequence of P^* by first order inference and induction, which are explained in the next section.

We use the following definitions and notations. First, we use the notion of *polarity* (see [9], [10]). The positive and negative subformulas are defined as follows :

(i) F is a positive subformula of F.
(ii) When $\neg G$ is a positive (negative) subformula of F, then G is a negative (positive) subformula of F.

(iii) When $G \wedge H$ or $G \vee H$ is a positive (negative) subformula of F, then G and H are positive (negative) subformulas of F.
(iv) When $G \supset H$ is a positive (negative) subformula of F, then G is a negative (positive) subformula of F and H it a positive (negative) subformula of F.

Next, variables which appear in a specification are distinguished in the following way. Let F be a closed first order formula. When $\forall X G$ is a positive subformula or $\exists X G$ is a negative subformula of F, then X is called a *free variable* of F. On the other hand, when $\forall Y H$ is a negative subformula or $\exists Y H$ is a positive subformula of F, then Y is called an *undecided variable* of F. In other words, when F is transformed into prenex normal form, free variables are variables quantified universally, and undecided variables are those quantified existentially.

A *goal formula* is a formula which is obtained from an 5-formula by replacing each undecided variable Y with ?Y and deleting all quantifications. Furthermore, a substitution o for a goal formula G is called a *deciding substitution* when a instantiates no free variable in G.

Example : An S-formula :
$\forall X(list(X) \supset \forall Y \exists Z\ append(X, Y, Z))$
is represented by a goal formula :
$list(X) \supset append(X, Y, ?Z)$,
where $list(X)$ is negative and $append(X, Y, ?Z)$ is positive.

A replacement of an occurrence of a term t in a formula F by s is denoted by $F_t[s]$, and a replacement of all occurrences of a term t in a formula F by s is denoted by $F_t(s)$. Formula F is said to be in a reduced form [10] with respect to logical constants *true* and *false*, if F is either (i) *true* or (ii) *false*, or (iii) if neither *true* nor *false* occur in F. The reduced form of a formula F is denoted by F i.

III. INFERENCE RULES

In this section, we give a brief description of our verification procedure for a specification (for a detailed explanation, see [3]).

A. Extended Execution

Since a specification is not restricted in a Horn clause but is expressed in an 5-formula, we need some extension of the usual Prolog interpreter. For this purpose, our verification system uses the following four inference rules.

(1) Case Splitting

Let G be a goal formula and H either (i) an outermost positive subformula of the form $H_1 \wedge \cdots \wedge H_k$ or (ii) an outermost negative subformula of the form $H_1 \vee \cdots \vee H_k (k > 1)$. Then, if each undecided variable appearing in H_i appears only in $H_i (1 \leq i \leq k)$, we generate new k AND-goals $G_H[H_1], \cdots, G_H[H_k]$.

The following two rules are an extension of a Prolog interpreter using polarity.

(2) Definite Clause Inference (DCI)

Let A be a positive atom in a goal formula G and let "$B :- B_1, \cdots, B_m$" be any definite clause in P. When A is unifiable with B by a deciding m.g.u.(most general unifier) σ, we generate new OR-goals for all such definite clauses : $\sigma(G_A[B_1 \wedge \cdots \wedge B_m]) \downarrow (\sigma(G_A[true]) \downarrow$ when $m=0$). All newly introduced variables are treated as fresh undecided variables.

(3) Negation as Failure Inference(NFI)

Let A be a negative atom in a goal formula G and let "$B :- B_1, \cdots, B_m$" be any definite clause in P. When A is not unifiable with a head of any definite clause in P, we generate a goal : $G_A[false] \downarrow$. When A is unifiable with B of a definite clause by an m.g.u. σ, we generate new AND-goals for all such definite clauses : $\sigma(G_A[B_1 \wedge \cdots \wedge B_m]) \downarrow (\sigma(G_A[true]) \downarrow$ when $m=0$). All newly introduced variables are treated as fresh free variables.

(4) Simplification

Let G be a goal formula. When $A_1, \cdots, A_m$ are positive atoms and $A_{m+1}, \cdots, A_n$ are negative atoms unifiable to A by a deciding m.g.u. σ, we generate new AND-goals : $\sigma(G)_A(true) \downarrow$ and $\sigma(G)_A(false) \downarrow$.

These four inference rules are repeatedly applied to a given goal formula until it is reduced to *true* or *false*. If these rules cannot be applied any more, then we appeal to the following induction.

B. Induction

Our verification system utilizes inductive proofs which are based on structural induction schemes. Those induction schemes are also used in [12],[13],[14] for the verification of Prolog programs. For example, the following is an induction scheme for list X :

$$\frac{Q([\,]) \qquad \forall A, X\ (Q(X) \supset Q([A|X]))}{\forall X : list\ Q(X)}$$

where $Q(X)$ is a theorem to be proved, $Q([\,])$ is a base case, and $\forall A, X\ (Q(X) \supset Q([A|X]))$ is an induction step.

IV. GENERALIZATION HEURISTICS

In the following, we discuss two kinds of generalization heuristics incorporated into our verification system.

A. Mechanical Generalization

The generalization employed in BMTP is a heuristic by which a term in a formula is replaced by a variable under an appropriate condition [4]. For example, when the generalization heuristic is applied to a formula in functional language :

$reverse(reverse(L))=L$
$\supset reverse(append(reverse(L), [X]))=[X|L]$,

then the following formula is obtained :

$reverse(N)=L \supset reverse(append(N, [X]))=[X|L]$,

where term *reverse(L)* in the first formula it replaced by a new variable *N* and thus a more general formula is obtained. On the other hand, let *G* be a goal formula :

$$(reverse(L, M) \supset reverse(M, L))$$
$$\supset (\underline{reverse(L, M)} \wedge append(M, [X], N) \supset reverse(N, [X|L]))$$

When we apply the simplification rule to the above underlined positive and negative *rever$t(L, M)* in *G,* then we generate the following AND-goals :

$$[(true \supset reverse(M, L))$$
$$\supset (true \wedge append(M, [X], N) \supset reverse(N, [X|L]))] \downarrow,$$
$$[(false \supset reverse(M, L))$$
$$\supset (false \wedge append(M, [X], N) \supset reverse(N, [X|L]))] \downarrow,$$

which are immediately reduced to :

$$reverse(M, L) \supset (append(M, [X], N) \supset reverse(N, [X|L]))$$

and *true,* respectively. This inference exactly corresponds to the above-mentioned generalization heuristic employed in BMTP.

Likewise, the inference of cross-fertilization in BMTP also corresponds to the one performed by simplification [3]. In this way, generalization and cross-fertilization, which are treated as different heuristics in BMTP, are performed in a unified way by the simplification rule in our verification system.

Furthermore, the heuristic of eliminating destructors in BMTP can be considered as a kind of generalization heuristic. For example, selectors for data structure like *ear(L)* and c*dr(L)* appearing in the goals of BMTP are eliminated and replaced by variables. In Prolog programs, we usually don't use such selectors explicitly ; we do without them by using unification. Hence, Prolog programming style sometimes makes unnecessary such a generalization heuristic as eliminating destructors.

B Intelligent Generalization

The second generalization heuristic differs from the above one, and BMTP intentionally does not employ it because it requires "creative* insight (chap.XII in [4]). In the verification of Prolog programs, however, there are also some cases where, in proving an induction step goal, we cannot use its induction hypothesis because of mismatching with its conclusion. As an example of such a case, consider the proof of the following theorem.

theorem (*rever se-rever se).*
$\forall XY\ reverse(X, [\,], Y) \supset reverse(Y, [\,], X) \quad \cdots (G_0)$
end.

where program *reverse* is defined as follows :

```
reverse([ ], X, X).
reverse([A|X], Y, Z) :- reverse(X, [A|Y], Z).
```

First, we try to prove the above theorem by induction and the induction scheme in section III - B is generated for *list*: *X,* where $Q(X)$ is $\forall Y\ reverse(X, [\,], Y) \supset reverse(Y, [\,], X)$.

The proof of its base case, Q([ ]), is trivial. The proof of the induction step goal, however, is not easily performed because of those underlined mismatched literals shown below :

[induction step] $(Q(X) \supset Q([A|X]))$

$$(reverse(X, [\,], ?Y_0) \supset reverse(?Y_0, [\,], X))$$
$$\supset (reverse([A|X], [\,], Y) \supset reverse(Y, [\,], [A|X]))$$
$$\Downarrow \text{NFI for } reverse([A|X], [\,], Y)$$
$$(reverse(X, [\,], ?Y_0) \supset reverse(?Y_0, [\,], X))$$
$$\supset (\underline{reverse(X, [A], Y)} \supset reverse(Y, [\,], [A|X]))$$

Here, since we cannot apply simplification because of mismatching between *[ ]" and "*[A]*" in the above underlined parts, there is no way to use the induction hypothesis. These "phenomena" often happen to the verification of Prolog programs containing a "accumulator" [6] like the second argument of *reverse,* which, though they make the computation linear order, cause at the same time mismatching between an induction hypothesis and its conclusion. Hence, in order to solve these kinds of mismatching, it is necessary to incorporate some heuristics in order to generate a generalized goal. Our verification system generates the following goal mechanically :

theorem (*generalized reverse-reverse*).
$\forall XSMT\ reverse(X, S, M) \wedge reverse(S, X, T)$
$\supset reverse(M, [\,], T) \quad \cdots (G_{gen})$
end.

We call such kind of generalization an *'intelligent* generalisation." It is easily known that the above theorem is actually a generalised goal of G_o and its proof can be rather straightforwardly performed.

V. INTELLIGENT GENERALIZATION HEURISTIC

In this section, we state how the intelligent generalization heuristic is applied to the proof of an induction step goal and give the intelligent generalization scheme which mechanically generates its generalised goal. We then go on to show that its scheme is also effective for flawed induction schemes [4].

A Intelligent Generalisation Scheme

At first, for ease of understanding, we illustrate the intelligent generalization heuristic by using the previous example.

$$(\underline{reverse(X, [\,], ?Y_0)} \supset reverse(?Y_0, [\,], X))$$
$$\supset (\underline{reverse(X, [A], Y)} \supset reverse(Y, [\,], [A|X]))$$

The first step of intelligent generalization is to find those mismatching arguments which make it impossible to use the induction hypothesis. In the above example, those mismatching arguments are "[ ]" and "(A)" in *rever$e(X,{),*?Yo) and *reverse(X, [A], Y),* respectively.

Next, we replace the mismatching arguments in the induction conclusion by new variables. In the above, "[A]" is replaced by a new variable, say, *T.* We call those variables contained in the mismatching arguments "*mismatching* variab/ei," and those arguments in the induction conclusion which contain mismatching variables are called "arguments relating to mismatching." In this case, "A" is a mismatching variable and "[A|X]" is an argument relating to mismatching. These arguments relating to mismatching are also replaced by new distinct variables. In this case, we replace "[A|Xj" by a new

variable, say, U, and the induction conclusion becomes the following formula :

$$\forall XYTU\ reverse(X,T,Y) \supset reverse(Y,[\,],U) \quad \cdots(G_1).$$

Clearly, the above goal G_1 is not a correct specification but an "over-generalized" goal of the original theorem G_0, and an appropriate constraint condition should be imposed on G_1 in order to obtain a correct generalised goal. Hence, we assume some "*constraint relation*" between freshly introduced variables and those variables which appear in arguments relating to mismatching. In this case, a constraint relation, say, $\mathcal{R}(T,X,U)$, is imposed, because T and U are newly introduced variables and X is a variable which appears in the argument relating to mismatching, i.e., $[A|X]$. We assume the following goal as a generalized goal of the original one :

$$reverse(X,T,Y) \wedge \mathcal{R}(T,X,U) \supset reverse(Y,[\,],U) \cdots(G_2),$$

where $\mathcal{R}(T,X,U)$ is some relation whose precise form is not determined yet ; and we call such a goal that contains an unspecified relation a "*temporary goal.*"

The third step is to infer the constraint relation mentioned above. Our current system infers such a relation from the following two conditions. The first inferring condition is called a "*generalisation condition,*" which is a necessary condition for a temporary goal to be actually a generalised goal of the original one. In this case, by comparing G_0 and G_2, it follows that X equals U when T is $[\,]$, which imposes the following constraint on $\mathcal{R}(T,X,U)$:

$$\mathcal{R}([\,],X,X). \qquad \cdots(CR_1)$$

The second inferring condition is that derived from "*pseudo verification,*" which means that we apply the inference rule mentioned in section III to the temporary goal. For example, if we apply induction on G_2, we get the following verification process for the induction step : $\forall AX\ P(X) \supset P([A|X])$, where $P(X)$ is :
$\forall YTU\ reverse(X,T,Y) \wedge \mathcal{R}(T,X,U) \supset reverse(Y,[\,],U)$.

$(reverse(X,?T_0,?Y_0) \wedge \mathcal{R}(?T_0,X,?U_0) \supset reverse(?Y_0,[\,],?U_0))$
$\supset (reverse([A|X],T,Y) \wedge \mathcal{R}(T,[A|X],U) \supset reverse(Y,[\,],U))$
$\Downarrow$ NFI for $reverse([A|X],T,Y)$
$(reverse(X,?T_0,?Y_0) \wedge \mathcal{R}(?T_0,X,?U_0) \supset reverse(?Y_0,[\,],?U_0))$
$\supset (reverse(X,[A|T],Y) \wedge \mathcal{R}(T,[A|X],U) \supset reverse(Y,[\,],U))$
$\Downarrow$ simplification w.r.t
$\Downarrow$ $reverse(X,?T_0,?Y_0)$ and $reverse(X,[A|T],Y)$
$(\mathcal{R}([A|T],X,?U_0) \supset reverse(Y,[\,],?U_0))$
$\supset (\mathcal{R}(T,[A|X],U) \supset reverse(Y,[\,],U))$
$\Downarrow$ simplification w.r.t
$\Downarrow$ $reverse(Y,[\,],?U_0)$ and $reverse(Y,[\,],U)$
$\mathcal{R}(T,[A|X],U) \supset \mathcal{R}([A|T],X,U). \qquad \cdots(CR_2)$

Similarly, as for the base case of the induction scheme for $P(X)$, we obtain the following constraint relation :

$$\mathcal{R}(Y,[\,],U) \supset reverse(Y,[\,],U). \qquad \cdots(CR_3)$$

Then, from these constraint relations, (CR_1), (CR_2) and (CR_3), we try to identify the unknown relation $\mathcal{R}(T,X,U)$. Our verification system searches for an already defined Prolog program which satisfies those constraint relations. Actually, we rewrite those constraint relations into formulas in Horn clauses ;

$\mathcal{R}([\,],X,X).$ $\cdots(CR_1^*)$
$\mathcal{R}([A|T],X,U) :- \mathcal{R}(T,[A|X],U).$ $\cdots(CR_2^*)$
$reverse(Y,[\,],U) :- \mathcal{R}(Y,[\,],U).$ $\cdots(CR_3^*),$

and then find a definite clause which can be "unified" with those (CR_i^*)'s whose heads are the unknown constraint relation. If there are no such clauses defined in the system, we consider all the permutations of the arguments of the constraint relation. That is, for example in this case, we consider all the permutations of $\{T,X,U\}$ and check each case for $\mathcal{R}(T,U,X)$, $\mathcal{R}(U,T,X)$, $\mathcal{R}(U,X,T)$, etc. In the above example, from (CR_1^*) and (CR_2^*), we can find that

$$\mathcal{R}(T,X,U) \equiv reverse(T,X,U), \qquad \cdots(G_3)$$

and it is easily confirmed to satisfy (CR_3). From G_2 and G_3, we finally obtain the generalised goal G_{gen} mentioned in section IV-B.

To sum up, the intelligent generalization scheme consists of the following 4 steps :

(1) Find out mismatching arguments in literals to which the induction has been applied.
(2) Replace those arguments relating to the mismatching by new variables in the induction conclusion (and obtain an "over-generalised goal").
(3) On the over-generalized goal, impose an appropriate constraint relation by generalization condition and by the pseudo verification.
(4) From the restrictions derived in (3), infer the constraint relation.

As for the inference method in step (4) above, our current implementation deals only with those constraint relations which can be reduced into Horn clauses as mentioned in the examples.

B Application to Flawed Induction Schemes

The application of the intelligent generalization scheme is not restricted to the above-mentioned proofs, but it is sometimes effective also for a "flawed" induction scheme [4]. For example, consider the following example which is a corollary of the associativity of *Append*.

theorem(corollary-of-append-associativity)
VXDR *append{X, X, D)Aappend(X, D, R) D append(D, X, R)*
end.

From the definition of *append,* we note that the predicate *append{X,Y,Z)* recursively changes its first and third argument and leaves its second argument fixed. If we use the terminology of BMTP, the first and third arguments are *changing* variables and the second argument is an *unchanging variable.* The above example shows a case where an induction scheme suggested by an atom and another induction scheme suggested by a different atom, are mutually flawed. That is, the induction scheme suggested by *appendix, D,R)* recursively changes *X,* which is an unchanging variable in

append(D, X, R), while the induction scheme suggested by *append*(D, X, R) recursively changes D, which is an unchanging variable in *append*(X, D, R) (and note that the induction scheme suggested by *append*(X, X, D) is flawed by itself).

In this case, our intelligent generalization scheme generates the following generalized goal :

$$append(X, V, D) \wedge append(X, U, R) \wedge append(V, V, U) \supset append(D, V, R),$$

where *append*(V, V, U) is a constraint relation obtained ; we know that the above goal is easily proved by an "unflawed" induction.

VI. CONCLUDING REMARKS & RELATED WORK

This paper has shown how generalization heuristics are incorporated into the verification system of Prolog programs. 'Two kinds of generalization, mechanical generalization and intelligent generalization, are discussed. We have shown that the mechanical generalization used in BMTP can be performed by simplification in our verification system, as well as in the case of cross-fertilization. To the intelligent generalization heuristic we have given a generalization scheme which is naturally incorporated into our inference system of the extended-execution style, and which has proved to be effective also for flawed induction schemes.

We owe our heuristics to BMTP and other related work by Moore [6) and Aubin [7], although there is a difference in target languages and deduction methods. As for the inferring constraint relation mentioned in section V, we find some relevance to Shapiro's [15] Model Inference System which is a general theory to infer a Prolog predicate satisfying given facts. Our intelligent generalization scheme differs in that it gives a Prolog predicate some constraint relations to be satisfied.

The generalization heuristics presented here have been examined by numerous hand proofs and their first versions have been implemented in DEC-10 Prolog on DEC-2060 as one of the heuristics of our verification system for Prolog programs. We intend to implement our verification system on PSI [16] as a basis for "Intelligent Programming Environment- for Prolog programming on which we can perform various kinds of experiments such as program transformation and synthesis.

Acknowledgments

The author appreciates K. Fuchi(Director of ICOT) , K. Furukawa (Chief of ICOT 2nd Laboratory) and T. Yokoi(Chief of ICOT 3rd Laboratory) for the chance of doing this research. He is greatly indebted to T. Kanamori for valuable comments, especially on the extended execution. He would also like to thank A. Fusaoka, H. Fujita and K. Suzuki for their useful discussions and help.

References

[1] Furukawa, K. and T. Yokoi, "Basic Software System" In Proc FGCS-84. Tokyo, Japan, November, 1984, pp. 37-57.

[2] Bowen, K. A. "Programming with Full First-Order Logic" In Machine Intelligence 10 (1982) 421-440.

[3] Kanamori, T. "Verification of Prolog programs Using an Extension of Execution," ICOT Technical Report, TR-096, 1984.

[4] Boyer, R. S. and J. S. Moore. Computational Logic, New York: Academic Press, 1979.

[5] Kanamori, T. et al. "Formulation of Induction Formulas in Verification of Prolog Programs," ICOT Technical Report, TR-094,1984.

[6] Moore, J. S., "Introducing iteration into the pure LISP theorem prover." IEEE Trans. Software Eng. 1.3 (1975) 328-338 .

[7] Aubin, R. "Some Generalisation Heuristics in Proofs by Induction" In Proc. IRIA Collog. on Proving and Improving Programs, Arc et Senans, France, July, 1975, pp. 197-208.

[8] Pereira, L. M., F. C. Pereira and D. H. Warren "User's Guide to DECsystem-10 PROLOG," Techical Report, Univ. of Edinburgh, September 1978.

[9] Prawitz, D. Natural Deduction, A Proof-Theoreticai-Study Stockholm: Almqvist & Wiksell, 1965.

[10] Murray, N. V., "Completely Non-Clausal Theorem Proving", Artificial Intelligence, 18:1 (1982) 67-85.

[11] Clark, K. L. "Negation as Failure" In Logic and Database. Gallaire. H and J. Minker. Eds., (1978) 293-322.

[12] Clark, K. L. and S-A. Tarnlund "A First Order Theory of Data and Programs" In IFIP-77. Toronto, Canada, August, 1977, pp. 939-944.

[13] Clark, K. L. "Predicate Logic as a Computational Formalism," Technical Report 59, Imperial College, December 1979.

[14] Stering, L. and A. Bundy "Meta-Level Inference and Program Verification" In 6th Conf. on Automated Deduction. Lecture Notes in Computer Science 138, 1982, pp. 144-150.

[15] Shapiro, E. Y. "An Algorithm that Infers Theories from Facts" In Proc. 1JCAI-81. Vancouver, Canada, August, 1981, pp. 446-451.

[16] Yokoi, T., S. Uchida, et al. "Sequential Inference Machine :SIM" In Proc. FGCS-84. Tokyo. Japan, November, 1984, pp. 70-81.