数理解析研究所講究録
第511巻 1984年 242-258

242

The Temporal Semantics of Logic Programming

東工大（工） 米崎直樹 (Naoki Yonezaki)

ABSTRACT

Temporal semantics of Horn logic programming and how it can be applied to reasning about a logic program are presented. In the computational model, the concept of 'world' or 'state' corresponds to computational states of a program i.e. a set of substitutions and execution points. Temporal logic used in this paper is precisely defined and fundamental semantics for execution is given by a set of schemas of the logic. A general proof procedure for total correctness is also presented. Finally several extension to Horn logic programming are considered in our framework.

§1. Introduction

Programming languages, represented by Prolog, based on first order predicate logic provides simple declarative semantics. First order logical deduction is used for verifying, synthesizing and translating programs, since the model theortic semantics of Horn sentences of first order predicate logic is straightforward.[1]~[3] On the other hand, the theory of logic programming and their computations could be formalized in terms of the theory of resolution proof procedure. There exists various problems about executions of

logic programs, such as termination problem and total correctness.[4],[5]

To characterize such properties, model theoritic semantics of a logic program execution is desired. Temporal logic has been shown to be adequate for expressing a wide variety of properties of execution sequence of programs.[6]~[13]

In this paper we presents a temporal semantics of Horn logic programming. This paper is organized as follows. In section 2, we define a modal logic, a version of temporal logic used in this formalism. In section 3, we define semantics of Horn logic programming. Section 4 presents a general verificaton method for logic programming. Section 5 discusses semantics of special constructs such as cut operator and pseudo-parallel execution. Section 6 gives conculuding remarks.

§2. Modal logic

The logic employed in this paper is a version of Modal logic(ML) which uses temporal operators such as $\Box$ (necessity operator), $\circ$ (next time operator) and until (until operator).

2.1. Syntax

Types: Let e and b be any two objects. The set T of types of ML is defined as follows,

(i) $e, b, (b) \in T$,

(ii) $a_0=e, a_1=e, \ldots, a_{n-1}=e$ imply $(a_0, a_1, \ldots, a_{n-1}) \in T$,

(iii) $a_0=e, a_1=e, \ldots, a_n=e$ imply $[a_0, a_1, \ldots, a_n] \in T$,

As will emerge, objects of type e will be possible entities on individuals and objects of type b will be labels, which will be used for indicating program locations. Objects of type (b) will be

predicates of labels. Objects of type $(a_0,a_1,\ldots,a_{n-1})$ will be predicates of n arguments which are objects of type e. Objects of type $[a_0, a_1,\ldots,a_n]$ will be functions of n arguments which are objects of type e and whose value is also object of type e.

Primitive Symbols: For each $a \in T$ we have a denumerable list of variables and non-logical constants together with the improper symbols (,), =, $\supset$, $\sim$, $\forall$, $\Box$, $\circ$ and until.

Terms: We characterize the set of Tm_a of terms of ML of type a, as follows,

(i) Every variable of type $a \in T$ belongs to Tm_a respectively,

(ii) Every constant of type $a \in T$ belongs to Tm_a respectively,

(iii) $A \in Tm_{[a_0,a_1,\ldots,a_n]}$, $B^0 \in Tm_{a_0}$, $B^1 \in Tm_{a_1}$, $\ldots$, $B^{n-1} \in Tm_{a_{n-1}}$ imply $A(B^0,B^1,\ldots,B^{n-1}) \in Tm_{a_n}$,

Atomic formula: An atomic formula of ML is an expression of one of the form

$$A(B^0, B^1,\ldots, B^{n-1})$$

,where A is of type $a=(a_0, a_1,\ldots, a_{n-1})$ and B^k is a term of type a_k for $k<n$: or the form

$$A = B$$

where A, B are terms of type e.

Formulas: Formulas of ML are generated from the atomic formulas by the connectives $\sim$,$\supset$, the quantifier $\forall x_a$, where x_a is an arbitrary variable, and modal operators $\Box$, $\circ$, until.

It is important to note that the sequence $\emptyset$ belongs to T, so that a symbol A of type $\emptyset$ standing alone is an atomic formula. The sentential connectives $\wedge$, $\vee$, $\leftrightarrow$ and the quantifier $\exists$ are defined as

usual.

2.2. Semantics

For an arbitrary set X, we denote the power set, or a set of all subsets of X by P(X). For sets $X_0, \ldots, X_{n-1}$, we let $X_0 \times \ldots \times X_{n-1}$ denote their Cartesian product.

Frame: Let D_a and D_b be non-empty sets. By a frame for ML based on D_a and D_b, we understand an indexed family (M_a) $a\epsilon T$ of sets, where

(i) $M_e=D_e$, $M_b=D_b$

(ii) For each type $a = (b_0,\ldots,b_{n-1})$,

$$M_a = P(M_{b_0} \times \ldots \times M_{b_{n-1}})$$

(iii) For each type $a = [b_0,b_1,\ldots,b_n]$,

$$M_a = M_{b_n}{}^{M_{b_0} \times \ldots \times M_{b_{n-1}}}$$

Model: A model of ML based on D_a, D_b and I is a system

$$M = (M_a , m)\ a\epsilon T, \text{ where}$$

(i) M_a $a\epsilon T$ is a frame based on D_a and D_b,

(ii) m (the meaning function) is a mapping which assigns to each constant C_a a function from I into M_a, where I be the linearly ordered set of denumerable states.

We denote by As(M) the set of all assignments over the model M, i.e. all functions on the set of variables such that $a(x_a)\epsilon M_a$ for each variable x_a. We define the value $V_{i,a}^{M}(A_a) \epsilon M_a$ of the term A_a with respect to the state i and the assignment a by the following recursion on the term $A_a\epsilon Tm_a$:

(i) $V_{i,a}^{M}(x_a) = a(x_a)$

(ii) $V_{i,a}^{M}(C_a) = m(C_a)(i)$

(iii) $V_{i,a}^{M}(A_a(B_{a_0},B_{a_1},\ldots,B_{a_{n-1}}))$

$$= V_{i,a}^{M}(A_a)(V_{i,a}^{M}(B_{a_0}),\ldots,V_{i,a}^{M}(B_{a_{n-1}}))$$

,where $a=[a_0,a_1,\ldots,a_n]$

We define the notion

M,i,a **sat** A

by recursion on the formula A of ML as follows:

(i) M,i,a **sat** $A(A_0,\ldots,A_{n-1})$

if and only if

$(V_{i,a}^{M}(A_0),V_{i,a}^{M}(A_1),\ \ldots\ ,V_{i,a}^{M}(A_{n-1}))\in V_{i,a}^{M}(A)$

(ii) M,i,a **sat** $[A=B]$ if and only if

$V_{i,a}^{M}(A)=V_{i,a}^{M}(B)$

,where A and B are terms of type e.

(iii) Usual satisfaction clauses for $\sim$, $\supset$, $\forall x_a$

(iv) M,i,a **sat** $\Box A$ if and only if $\forall j\ j\geq i$, M,j,a **sat** A

(v) M,i,a **sat** A_1 <u>until</u> A_2 if and only if

$\forall j\ j\geq i$, M,j,a **sat** A_1 or

$\exists j\ j\geq i$, $(M,j,a$ **sat** $A_2 \wedge \forall k(i\leq k\leq j \rightarrow M,k,a$ **sat** $A_1))$

(vi) M,i,a **sat** $\circ A$ if and only if

$\forall j((i<j \wedge \forall k(i<k \rightarrow j\leq k)) \rightarrow M,j,a$ **sat** $A)$.

Notice the until operator we have defined does not have an eventuality component, and A_1 and A_2 can be satisfied simultaniously, in contrast with the one defined in [10]. This involves no difference of expressive power since until operator in [10] can be expressed in our system using our until operator, and vice versa. This is just for the convenience of having compact logical expressions in our application.

A is true in M, if M,i,a **sat** A for every state i and assingment a. A set F of formulas is satisfied in M by i and a, and we write M,i,a **sat** F, if M,i,a **sat** A for every $A \in F$. A formula A is a semantical consequence in ML of a set S of formulas, if M,i,a **sat** A whenever M,i,a **sat** S. We introduce (temporal) macro operators derived

from these primitives defined above to enhance the understandability of formulas in the specifications,

$$\Diamond A \stackrel{d}{=} \sim \Box \sim A$$

$$[A ==> B]\ \boxplus C \stackrel{d}{=} A \supset (C \ \underline{until}\ B)$$

$$[A ==> B]\ \diamondplus C \stackrel{d}{=} A \supset \sim(\sim C \ \underline{until}\ B)$$

$$[A ==> B]\ \boxplus(C \supset \diamondplus D) \stackrel{d}{=} [A ==> B]\ \boxplus\ ([C ==> B]\ \diamondplus D)$$

Intuitively speaking, $[A ==> B]\ \boxplus C$ means that C holds throughout the interval starting at the first state in which A holds and extending to the first state in which B holds. $[A ==> B]\ \diamondplus C$ requires that, if A holds in a future, and if the subsequent state in which B holds is found, this interval must sometime satisfy C. So that $[A ==> B]\ \boxplus(C \supset \diamondplus D)$ requires that, for the interval beginning with the state in which A holds and ending with the state in which B holds, if the state in which C holds is found then the state in which D holds should be found in the sub interval beginning with the state in which C holds and ending with the state in which B holds.

Now we present an axiomatic system in which proofs of the properties of program can be carried out.

Axioms of ML

A1: $\Box(A \supset B) \supset (\Box A \supset \Box B)$
A2: $\Box A \supset A$
A3: $\circ(\sim A) = \sim(\circ A)$
A4: $\circ(A \supset B) = (\circ A \supset \circ B)$
A5: $\Box A \supset \circ A$
A6: $\Box A \supset \circ\Box A$
A7: $\Box(A \supset \circ A) \supset (A \supset \Box A)$
A8: $(A \ \underline{until}\ B) = ((A \wedge B) \vee (A \wedge \circ(A \ \underline{until}\ B)))$
A9: $\Box A \supset (A \ \underline{until}\ B)$
A10: $\forall x\ A(x) \supset A(t)$,where t is free for x in A
A11: $\forall x\ \Box A \supset \Box \forall x\ A$
A12: $\forall x\ \circ A \supset \circ \forall x\ A$
A13: $\forall x(A \ \underline{until}\ B) \supset (\forall x A \ \underline{until}\ B)$, provided B does not contain free occurrence of x.

Inference Rules

R1: IF A is an instance of a tautology, then $\vdash A$.

R2: If $\vdash A$ and $\vdash A \supset B$, then $\vdash B$.

R3: If $\vdash A$, then $\vdash \Box A$.

R4: If $\vdash A \supset B$, then $\vdash A \supset \forall x B$, provided A does not contain free occurrence of x.

This system is certainly sound, and completeness of the system should be proved, however, it is beyond the scope of this paper.

§3. Semantics for logic programming in ML

A program is a set of clauses. A clause is a pair of sets of atomic formulas written as

$$A_1,\ldots,A_m \leftarrow B_1,\ldots,B_n \quad m\geq 0,\ n\geq 0.$$

The set $\{A_1,\ldots,A_m\}$ is the conclusion of the clause; $\{B_1,\ldots,B_n\}$ is the premise of the clause. A conditional clause is one where m=1, n>0. A negative clause is one where m=0 and n>0. A positive clause is one where m=1, n=0. An atomic formula is $P(t_1,\ldots,t_k)$ where P is a k-place predicate symbol, $t_1,\ldots,t_k$ are terms. A term is a variable or $f(t_1,\ldots,t_q)$ where f is a q-place function symbol, $t_1,\ldots,t_q$ are terms, and $q\geq 0$. A 0-place function symbol is a constant. Substitution is an operation, say θ, which replaces all occurrences of a variable throughout expression e by a term. The result is denoted by $e\circ\theta$. If there exists for expressions $e_1,\ldots,e_n$ a substitution θ such that $e_1\circ\theta=e_2\circ\theta=\ldots=e_n\circ\theta$, then θ is to be a unifier of $e_1,\ldots,e_n$.

A logic program can be seen as a set of clauses in first order predicate and hence it has a model theoretic semantics as usual. On the other hand, it has an operational semantics as process of resolution.

To formalize such a process, we use ML. Semantics of a program is defined by the set of formulas of ML generated for the program by rules associtated with each clause of Horn logic. We call those rules semantics of logic programming.

Before we explain the semantics, we have to introduce three

primitive predicates 'at', 'after' and 'end' of type (b) referring to the execution points of atomic formulas.

at(L) : This formula is true when the matching of an atomic formula whose label is L begins.

after(L): This formula is true when an atomic formula whose label is L is refuted.

end(L) : This formula is true from the success of refutation of an atomic formula whose label is L until the atomic formula is backtracked.

Provided an atomic formula Q is of the form $P(t_1,\ldots,t_n)$, term(Q) denotes $(t_1,\ldots,t_n)$, tuple of terms which are arguments of P, and $\bar{Q}$ denotes the predicate symbol P as a label of atomic formula Q. ${}_j\bar{Q}$ denotes ${}_jP$, where j is an index introduced for distinguishing occurrences of identical predicate symbols in conclusions, and is assigned in order of appearance (top to down) in the program. Index j may take values from 1 to mp ,where mp is the number of such clauses with predicate symbol P in conclusion. For a predicate symbol P occurring in premise of some clause but not in conclusion of any clause, we assume mp=0. We also assume that the every occurrence of identical predicate symbol P in a premise is distinguished by being attached with suffix k as P^k.

For each type of clause (e.g. conditional, negative, positive clauses), we can now define Horn logic programming semantics, which are schemes for derivating ML formulas.

(i) Negative Clause $\leftarrow Q_1,\ldots,Q_n$

semantics:

Initial call rule:

$$\textbf{init} \supset at({}_1\bar{Q}_1) \wedge u=\emptyset \wedge v=term(Q_1)$$

Left to right rule:

$$\text{For } 1 \leq i < n, \forall q(\text{after}({}_q\bar{Q}_i) \supset \text{at}({}_1\bar{Q}_{i+1}) \wedge v = \text{term}(Q_{i+1}))$$

Termination rule:

$$\forall q(\text{after}({}_q\bar{Q}_n) \supset \Box\textbf{end})$$

$$\text{at}({}_{mq1+1}\bar{Q}_1) \supset \Box\textbf{fail}$$

Backtracking rule:

For all atomic formula X, for $1 < i \leq n$,

$$\forall q \forall \rho_1 \forall \rho_2((\text{at}({}_q\bar{X}) \wedge \rho_1 = u \wedge \rho_2 = v \wedge \bigcirc\text{at}({}_1\bar{Q}_i)) \supset$$
$$\Diamond(\text{at}({}_{mqi+1}\bar{Q}_i) \supset \bigcirc(\text{at}({}_{q+1}\bar{X}) \wedge \rho_1 = u \wedge \rho_2 = v)))$$

u is a constant of type e holding a current set of substitutions. (The semantic value of u is a function from states to sets of substitutions at each state, so that a set of substitutions depends on states.) A constant v holds a record of terms which are going to be unified. For a record of terms $T=(t_1,\ldots,t_n)$, and a set of substitution ρ, $T\circ\rho$ denotes a new record of resulting terms $(t_1\circ\rho,\ldots,t_n\circ\rho)$, where $t_i\circ\rho$ is a resulting term obtained by applying every substitutions in ρ to term t_i.

Initial call rule describes that unifiability of atomic formulas Q_1 and ${}_1Q_1$ is checked initially. Left to right rule shows that if Q_i is refuted then unifiability of Q_{i+1} and ${}_1Q_{i+1}$ is checked.

If Q_n is refuted, then the negative clause succeeds in its refutation. The second formula in the termination rule describes the finite failure.

Backtracking rule shows that if refutation of some atomic formula is finitely failed then unification of the previously succeeded atomic formula and its next matching alternative formula is checked.

(ii) Conditional Clause $\quad {}_jP \leftarrow R_1,\ldots,R_n$

Semantics:

Unification rule:

$$\forall\rho_1\forall\rho_2((at({}_j\bar{p})\wedge\rho_1=u\wedge\rho_2=v\wedge match(v\circ u,term({}_jP)))$$
$$\supset\bigcirc(u=\rho_1\cup mgu(\rho_2\circ\rho_1,term({}_jP))\wedge at({}_1\bar{R}_1)\wedge v=term(R_1)))$$

Left to right rule: For $1\leq i<n$,

$$\forall q(after({}_q\bar{R}_i)\supset(at({}_1\bar{R}_{i+1})\wedge v=term(R_{i+1})))$$

Top to down rule:

$$(at({}_j\bar{P})\wedge\sim match(v\circ u,term({}_jp)))\supset at({}_{j+1}\bar{P})$$

Success rule:

$$\forall q(after({}_q\bar{R}_n)\supset after({}_j\bar{P}))$$

Backtracking rule:

For all atomic formulas X, for $1<i\leq n$,

$$\forall q\forall\rho_1\forall\rho_2((at({}_q\bar{X})\wedge\rho_1=u\wedge\rho_2=v\wedge\bigcirc at({}_1\bar{R}_i)\supset$$
$$\Diamond(at({}_{mqi+1}\bar{R}_i)\supset\bigcirc(at({}_{q+1}\bar{X})\wedge\rho_1=u\wedge\rho_2=v)))$$

All the rules for conditional clauses are similar to the rules for negative clauses.

'match' is a predicate constant which is true if its arguments (tuples of terms) have the most general unifier. 'mgu' is a function constant which gives one of the most general unifier of its arguments.

Top to down rule describes that if the conclusion is not unifiable with an atomic formula whose terms are indicated by v, then the next alternative conditional or positive clause is selected. Success rule means that if refutations of all the atomic formulas in a premise succeeded then the conclusion succeeds in its refutation.

Backtracking rule is the same as in the case of negative clauses.

Note that suffix k for distinguishing occurrences of the same predicate symbols in premises is considered in the induction step of proof procedure, as will emerge later.

(iii) Positive Clause $_jP \leftarrow$

Semantics:

Success rule:

$$\forall\rho_1\forall\rho_2(at(_j\bar{P})\wedge\rho_1=u\wedge\rho_2=v\wedge match(v\circ u, term(_jP)) \supset \bigcirc(u=\rho_1\cup mgu(\rho_2\circ\rho_1, term(_jP))\wedge after(_j\bar{P}))$$

Top to down rule:

Same as the case of conditional clause.

Success rule describes that if $_jP$ matches with atomic formula whose predicate symbol P, then the refutation of the atomic formula is succeeded.

§4. General proof procedure for total correctness

In the following, we present a general inductive proof procedure for total correctness of Horn logic programming.

By using semantic rules previously mentioned, total correctness is expressed by the formula

$$\mathbf{init}\supset\Diamond(\mathbf{end}\wedge P((t_1,\ldots,t_n)\circ u))$$

where P is a n-place predicate which should be satisfied with the results.

The steps are as follows.

[Step1]: Assignment of assertion.

For each conditional clause, assign an appropreate intermittent assertion of the form $A(m)\supset\Diamond B$, where m is a variable on which

structural induction is performed.

[Step2]: Generation of verification conditions.

For each conditional clause, generate a verification condition as follows:

For each occurrence of predicate symbols appearing in its premise, make an assertion from the assertion assigned at step 1 to a conditional clause whose conclusion has the predicate symbol, in which non-logical constants of labels and non-logical constants corresponding variables of atomic formulas are associated with suffix attached to the predicate symbol in premise.

Provided that Q(P) is the conjunction of such rewritten assertions for a conditional clause whose conclusion is an atomic formula with predicate symbol P and S is the conjunction of formulas generated from a program by rules in section 3 , if R(m) is an assertion $A(m) \supset \Diamond B$ for the conditional clause, then the verification conditions are $S \supset R(0)$ and $Q(P) \wedge S \supset R(m)$ which works as inductive step, and in which n of the assertion $A(n) \supset \Diamond B'$ in Q(P) must be smaller than m in structure.

[Step3]: Proof of verification condition.

Verify the verification conditions for all the conditional clauses in the program and establish the assertions.

(Example)

To prove the total correctness $\forall n(\mathbf{init} \supset \Diamond \mathbf{end} \wedge *w \circ u = N!)$ for the following simple factorial program,

$\leftarrow$ Fact(N, *w)

$_1$Fact(0, S(0)) $\leftarrow$

$_2$Fact(S(*x), *y) $\leftarrow$ Mul(s(*x), *z, *y), Fact(*x, *z)

we can assign the following assertion to the third clause.

$\forall n \forall \rho(at(_2Fact) \wedge u = \rho \wedge v \circ u = (N, *w)) \supset \Diamond(after(_2Fact) \wedge u = \rho u\{*w/N!\})$,

where N stands for s(...(s(0))...) and n is the number of a function s in N, and N! stands for s(...(s(0))...) in which the number of a function s is n!.

This general proof procedure reflects the structure of Horn logic programming.

§5. Semantics of extensions to Horn logic program

5.1. Cut operator

Prolog programming uses a special symbol '!' called cut operator.

$$_jP \leftarrow R_1,\ldots,R_{k-1} \; ! \; R_k,\ldots,R_n$$

According to the semantics defined above, we can now define the semantics of cut operator by supplimenting following rules.

Semantics:

Left to right rule:

$$\forall q \; (after(\; _q\bar{R}_{k-1}) \supset at(_1\bar{R}_k) \wedge v=term(R_k))$$

Cut rule:

$$at(_{mrk+1}\bar{R}_k) \supset at(_{mp+1}\bar{P})$$

If a cut operator is used in a negative clause, we simply have

$$at(_{mrk+1}\bar{R}_k) \supset \square \textbf{fail}.$$

Left to right rule describes that if refutation of R_{k-1} is succeeded then the matching of R_k and $_1R_k$ is checked. Cut rule means that if the refutation of $R_k,\ldots,R_n$ is finitely failed (that is all the backtracking is failed) then the refutation of P is finitely failed without backtracking to alternative check for R_{k-1}.

5.2. Pseudo-parallel execution

Parallelism condidered in this paper is so called 'and-parallel'.

We now extend the Horn logic programming to allow to express parallel execution by a special symbol '//'.

$$_jP \leftarrow R_1//.../\!/R_n$$

The declarative reading of the clause is unchanged by the symbol '//'. Operationally, however, each atomic formula R_i is intended to be executed pseudo-parallely. (i.e. all refutations derived from the refutations of R_i's ($1 \leq i \leq n$) are interleaved.) Provided that execution is implemented on a single stack as in the case of IC-Prolog[14], the semantics of the pseudo-parallel execution is formalized by the following rules.

Semantics:

Parallel rule:

$$\forall\rho_1\forall\rho_2\ ((at(_j\bar{P})\wedge\rho_1=u\wedge\rho_2=v\wedge match(v\circ u, term(_jP)))$$
$$\supset\bigcirc(u=\rho_1\cup mgu(\rho_2\circ\rho_1, term(_jp))\wedge\forall i(\Diamond(at(_1\bar{R}_i)\supset v=term(R_i)))))$$

For $1\leq i\leq n$, **init**$\supset\sim at(_1\bar{R}_i)$ <u>until</u> $at(_j\bar{P})$

Top to down rule:

Same as the usual case.

Backtracking rule:

For all atomic formulas X,Y ($X\neq Y$), for $1\leq i\leq n$,

$$\forall q\forall r\forall\rho_1\forall\rho_2\ [(at(_q\bar{X})\wedge\rho_1=u\wedge\rho_2=v\wedge match(v\circ u, term(_qX)))==>$$
$$(at(_{mri+1}\bar{R}_i)\wedge\bigcirc(at(_{q+1}\bar{X})\wedge\rho_1=u\wedge\rho_2=v))]$$
$$\boxplus((at(_r\bar{Y})\wedge match(v\circ u, term(_rY)))\supset\Diamond at(_{r+1}\bar{Y}))$$

Success rule:

For $1\leq i\leq n$, $\forall q(after(_q\bar{R}_i)\supset(end(_q\bar{R}_i)$ <u>until</u> $at(_{q+1}\bar{R}_i)))$

$$\bigwedge_{1\leq i\leq n}(\exists q\ end(_q\bar{R}_i))\supset after(_jP)$$

Model assumption:

$$\mathbf{init} \supset \Box(\exists x(at(x)) \vee \mathbf{end} \vee \mathbf{fail})$$
$$\wedge \Box \forall x \forall y(x \neq y \supset \sim(at(x) \wedge at(y)))$$

Apparently, if the conclusion ${}_jP$ matches with some atomic formula in a premise of some other clause, then any refutation of R_i can be proceeded. However, notice that one and only one possible atomic formula can be selected at a time. This fact is described in model assumption. Backtracking rule is rather complex than that of non-parallel case. The rule describes that if refutation of some atomic formula is finitely failed, then an alternative conclusion is selected to be refuted for such atomic formula that matched with some conclusion, and from that time till the finite failure, there is no other unification that was not canceled. (i.e. backtracked.)

To describe this computation model we need <u>until</u> operator. This operator is also used for success rule for describing that refutation of conclusion succeeds when all the atomic formula in its premise have been succeeded. This is because the execution of the refutation is interleaved.

§6. Concluding remarks

There are several extensions to Horn logic programming, some of which we have considered. The modal logic introduced here is so powerful that such extension as shared variables can be formalized in this logic. More expressive logic so called intensional logic may be useful for formalizing coroutine control and stream variable.[15],[16] In general, branching time logic is used for formalizing non-deterministic processes[17], however, in the case that the order of execution is concerned, this logic is not appropriate.

References

[1] M.H.V.Emden, R.A.Kowalski : The Semantics of Predicate Logic as a Programming Language, JACM 23(4), pp.733~742, Oct. 1976.

[2] R.A.Kowalski : Logic of Problem Solving, North-Holland, Amsterdam, 1978.

[3] C.J.Hogger : Derivation of Logic Program, JACM 28(2), pp372-392, April 1981.

[4] D.Harel : On the Total Correctness of Nondeterministic Programs, Theor. Comput. Sci. 13, 1981.

[5] E.Y.Shappiro : Algorithmic Program Debugging, Research Report 237 Yale University, May 1982.

[6] S. Owicki: Axiomatic Proof Techniques for Parallel Programs, Ph. D. Th., Cornell University, August 1975.

[7] A.Pnueli, The Temporal Logic of Programs. 18th Annual Symposium on Foundations of Computer Science. Nov, pp.46~57, 1977.

[8] Z.Manna, A.Pnueli :The Modal Logic of Programs, 6th International Colloquim on Automata, Language and Programming, Graz, Austria, Lecture Notes in Computer Science, Vol. 71, Springer Verlag, pp.386~408, July 1979.

[9] A.Pnueli : The Temporal Semantics of Concurrent Programs. In Khan, Ed., Semantics of Concurrent Computation, Springer Lecture Notes in Computer Science, Springer-Verlog, pp.1~20, 1979.

[10] D.Gabbay, A.Pnueli, S.Sheloh, J.Stave :The Temporal Analysis of Fairness :Seventh ACM Symposium on Principles of Programming Languages. Las Vegas, NV, January 1980.

[11] Z.Manna : Logics of Programs : Information Processing 80, S.H. Lavington(ed.) North-Holland Publishing Company, pp.41~51,1980.

[12] D.Harel : First Order Dynamic Logic, Lecture Notes in Computer

Science 68, Springer-Verlag, Berlin, 1979

[13] B.Hailpern : Verifying Concurrent Processes Using Temporal Logic. Technical Report 195. Computer Systems Laboratory, Stanford University, August 1980.

[14] K.L.Clark, F.G.McCabe and S.Gregory: IC-PROLOG Language Features, Logic programming, Academic Press, pp253-266, 1982.

[15] N.Yonezaki and T.Katayama : Functional Specification of Synchronized Processes based on Modal Logic, Proc. of 6th ICSE, pp208-217, Sep.1982.

[16] K.L.Clark, S.Gregory : A Relational Language for Parallel Programming, Proc. of Conference on Functional Programming and Computer Architecture. pp.171~178, Oct. 1981.

[17] L. Lamport : 'Sometime' is Sometimes 'Not Never' On the Temporal Logic of Programs, Seventh ACM Symposium on Principles of Programming Languages. Les Vegas, NV. pp.174-185, January 1980.