プログラミング-言語・基礎・実践- 16－10
(1994. 3. 9)

μ-calculusによる再帰プロセスの合成[1]

木村 成伴，富樫 敦，白鳥 則郎

東北大学電気通信研究所

〒980 仙台市青葉区片平 2-1-1

Email : {kimura,togashi,norio}@shiratori.riec.tohoku.ac.jp

あらまし 代数的プロセスは通信プロトコルや並行プロセスの記述や検証問題などに用いられている．その反面，その記述の容易さや読解性にはやや難がある．この問題を解決する手がかりとして，本稿では，プロセスの性質を与え，それらを全て満たすプロセスを合成するアルゴリズムを提案する．プロセスの性質を表すために μ-calculus を用いるが，この論理演算子の内，不確定要素を持つ $\vee$ と μ 演算子を含む論理式は合成が困難なため，対象から除外している．さらに，このアルゴリズムが極限において，意図したプロセスと上記の制約の元で区別がつかないプロセスに収束することを示す．

和文キーワード μ-calculus , CCS, プロセス代数, 帰納推論

Synthesis Algorithm for Recursive Processes by μ-calculus

Shigetomo Kimura, Atsushi Togashi and Norio Shiratori

Research Institute of Electrical Communication, Tohoku University.

2-1-1 Katahira, Aoba-ku, Sendai 980, JAPAN

Email : {kimura,togashi,norio}@shiratori.riec.tohoku.ac.jp

Abstract Algebraic calculation method are used to describe communication protocol and concurrent programs etc., and applied to verification problem. On the other hand, it is not easy to describe processes or understand these applications. As a clue to solve the problem, this paper proposes synthesis algorithm of algebraic process using the enumeration of facts, which must be satisfied by the process. We adopt μ-calculus to represent facts of a process. However it is too difficult to synthesize a process from formulae with any operators of μ-calculus . To simplify the problem, we exclude $\vee$ and μ operators. The validity of the proposed algorithm can be stated that it synthesizes a process in the limit, which cannot be distinguished from the target one under the above restriction.

key words μ-calculus , CCS, process algebra, inductive inference

[1]本研究は一部，旭硝子財団，文部省科学技術研究補助金による援助を受けている．

1 Introduction

The theoretical study of the inductive inference was started by the identification of sequential machines by Moore in 1950's. It then developed into the theories of identification for systems, language grammars, and computer programs [1, 11].

The studies of process algebras started from the latter half of 1970's to give mathematical semantics for concurrent systems. Typical systems are CSP by Hoare[6] and CCS by Milner[10]. In Feb. 1990, ISO adopted LOTOS[3] as the international standard for OSI specification description language. Those algebraic formalization techniques are utilized as the descriptive languages for communicating processes and concurrent programs. They are also applied to the verification problem, by virtue of the mathematical formality. The processes, however, have the features of such as nondeterminacy and concurrency, so their operational semantics are completely different from those of the traditional automata and formal languages.

The inductive inference of the processes forms a basis for the automatic synthesis of the highly reliable communicating protocols and concurrent programs from the examples. However, little has been investigated for inductive inference of concurrent processes, due to the difficulties arising from those situations.

From such a viewpoint, we have already presented the algorithm that inductively synthesizes a basic process in a subclass of CCS, from concrete examples, modal formulae describing the properties of the process[7]. The validity and improvement of the approach have been demonstrated. However the expressive power of basic processes is weak, they cannot express recursive behavior of a system. Thus, it is too difficult for synthesis of recursive processes from the modal formulae within finite steps.

This paper presents a synthesis algorithm for recursive processes by μ-calculus . The μ-calculus [4, 9, 12] is too strong as a expressive language of properties for a process. To simplify this problem, we exclude $\vee$ and μ operators. The validity of the proposed algorithm can be stated that it synthesizes a process in the limit, which cannot be distinguished from the target one by a given enumeration of facts under the above restriction.

The outline of this paper is as follows: Section 2 presents the algebraic formulation of processes, together with μ-calculus . Section 3 gives an algorithm that synthesizes a process satisfying a given enumeration of formulae. Since the sequence of formulae is in general of infinite length, the algorithm does not terminate. Hence, the concept of the convergence in the limit is provided. The validity of the algorithm is concluded by showing that the infered sequence algorithm converges in the limit to a process which cannot be distinguished to the intended process by a given enumeration of facts under the above restriction.

2 Preliminaries

In this section, we briefly review the preliminary notions such as algebraic processes and μ-calculus . See [4, 5, 6, 9, 10, 12], for more detailed discussions.

2.1 Algebraic Processes

Let $\mathcal{A}$ be an *alphabet*, a finite set of symbols. Its element is called an *action*. This corresponds to a primitive event of a process and this is assumed to be externally observable and controllable from the environment. Throughout this paper, it is assumed that we have a denumerable set $\mathcal{C}$ of *process constants*.

Definition 1 *Recursive terms are defined inductively as follows.*

1. *An inaction* $\mathbf{0}$ *and a process constant* $c \in \mathcal{C}$ *are recursive terms.*
2. *If* p *is a recursive term, an action prefix* $a.p$ *is a recursive term where* $a \in \mathcal{A}$.
3. *If* p_1 *and* p_2 *are recursive terms, their summation* $p_1 + p_2$ *is a recursive term.*
4. *A process constant* c *with a defining equation* $c \stackrel{\text{def}}{=} p$, *denoted as* $\mathbf{rec}\, c.p$ *is a recursive term, where* p *is a recursive term.* □

In a recursive term $\mathbf{rec}\, c.p$, every occurrence of c in p is called *closed*. If every occurrence of any process constant in p is closed, p is called *closed*. Closed terms are called *processes*. By renaming process constants, every term p is converted to a term p' such that if $\mathbf{rec}\, c_1.p_1$ and $\mathbf{rec}\, c_2.p_2$ are subterms in p' then $c_1 \neq c_2$. This conversion is exactly same as the one in α conversion in λ calculus. Thus, a term p can be represented as p with a set $\{c_1 \stackrel{\text{def}}{=} p_1, \ldots, c_n \stackrel{\text{def}}{=} p_n\}$ of defining equations, where every subterm of the form $\mathbf{rec}\, c.q$ in p is replaced by c.

Semantics of processes is given by a labeled transition system with actions as labels.

Definition 2 *A labeled transition system is a triple* $< S, \mathcal{A}, \rightarrow >$, *where* S *is a set of states and* $\rightarrow$ *is a transition relation defined as* $\rightarrow \subset S \times \mathcal{A} \times S$. □

For $(s, a, s') \in \rightarrow$, we normally write $s \stackrel{a}{\rightarrow} s'$. Thus, the transition relation can be written as $\rightarrow = \{\stackrel{a}{\rightarrow} | \ a \in \mathcal{A}\}$. $s \stackrel{a}{\rightarrow} s'$ may be interpreted as "in the state s an action a can be performed and after the action the state moves to s'". We use the usual abbreviations as e.g. $s \stackrel{a}{\rightarrow}$ for $\exists s' \in S$ s.t. $s \stackrel{a}{\rightarrow} s'$ and $s \stackrel{a}{\not\rightarrow}$ for $\neg\exists s' \in S$ s.t. $s \stackrel{a}{\rightarrow} s'$.

Definition 3 *A transition relation on processes is given by the following transition rules:*

$$\frac{}{a.p \xrightarrow{a} p} \qquad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}$$

$$\frac{\mathbf{rec}\, c.p \xrightarrow{a} p'}{p \xrightarrow{a} p'}$$

□

Definition 4 *c is guarded in p if every free occurrence of c is within some subterm $a.q$ of p. p is guarded if every constant is guarded in p.* □

Based on the operational semantics given by the transition system, several equivalences and preorders have been proposed in order to capture various aspects of the observational behavior of processes. One of those is the equivalence induced by the notion of a bisimulation [10].

Definition 5 *A relation R over processes is a strong bisimulation if $(p,q) \in R$ implies, for all $a \in \mathcal{A}$,*

1. *whenever $p \xrightarrow{a} p'$, then there exists q' such that $q \xrightarrow{a} q'$ and $(p',q') \in R$;*
2. *whenever $q \xrightarrow{a} q'$, then there exists p' such that $p \xrightarrow{a} p'$ and $(p',q') \in R$.* □

Processes p and q are *strongly equivalent* iff $(p,q) \in R$ for some strong bisimulation R. $p \sim q$ denotes that p and q are strongly equivalent. Clearly, $\sim$ is the largest strong bisimulation and an equivalence relation.

2.2 μ-calculus

The alternative characterization of equivalence on processes depends on the identification of a process with the properties it enjoys. Then we can say that two processes are equivalent if and only if they enjoy exactly same properties. In other words, two processes are inequivalent if one enjoys a property that the other does not enjoy. For this purpose, in this paper we adopt the μ-calculus [4, 9, 12], which have a modality concerning actions, in order to describe properties of processes.

Definition 6 *Formulae in μ-calculus are defined inductively as follows.*

1. *T (true) is a formula.*
2. *A variable $x \in X$ is a formula, where $\mathcal{X}$ is a denumerable set of logical variables.*
3. *If f and f' are formulae, $f \vee f'$, $\neg f$ are formulae.*
4. *If f is a formula, $\langle a \rangle f$ is a formula, where $a \in \mathcal{A}$.*
5. *If x is a variable and f is a formula with positive occurrence of x — x occurs within scopes of positive number of negations — $\mu x.f$ is a formula.* □

The set of all formulae is written as $\mathcal{L}$. In the following, we regard formulae as properties of processes. When a process p satisfies formula f, it is written as $p \models f$.

Definition 7 *Satisfaction relation of formulae is defined as follows:*

1. *For any process p, $p \models T$.*
2. *$p \models A_1 \vee A_2$, if $p \models A_1$ or $p \models A_2$.*
3. *$p \models \neg A$, if $p \not\models A$, where $p \not\models A$ means that p does not satisfy A.*
4. *$p \models \langle a \rangle A$, if there exists some q such that $p \xrightarrow{a} q$ and $q \models A$.*
5. *$p \models \mu x.f(x)$, if for any g such that $f(g) \supset g (\equiv \neg f(g) \vee g)$, $p \models g$.* □

Definition 8 *The following logical notations are used for convenience.*

1. $F \stackrel{\text{def}}{=} \neg T$.
2. $A_1 \wedge A_2 \stackrel{\text{def}}{=} \neg(\neg A_1 \vee \neg A_2)$.
3. $[a]A \stackrel{\text{def}}{=} \neg\langle a \rangle \neg A$.
4. $\nu x.f(x) \stackrel{\text{def}}{=} \neg\mu x.\neg f(\neg x)$. □

For a set of formulae $L (L \subseteq \mathcal{L})$ and a process p, $L(p)$ is defined as follows.

$$L(p) \stackrel{\text{def}}{=} \{A \in L \mid p \models A\}$$

Definition 9 *A variable x in a formula f is guarded, if every occurrence of x is within some scope of $\langle a \rangle$. A formula f is guarded if every variable in f is guarded.* □

Lemma 10 *Any formula can be equivalently converted to a formula without negation, i.e. a formula built up with T, F, $\wedge$, $\vee$, $\langle a \rangle$, $[a]$, μ, and ν.* □

In the following, we will consider closed formulae without negation.

Proposition 11 *[4] Let $f(x)$ be a guarded formula. Then the followings are satisfied:*

1. $\mu x.f(x) \equiv \bigvee_{k>0} f^k(F)$,
2. $\nu x.f(x) \equiv \bigwedge_{k>0} f^k(T)$. □

Proposition 12 *[4] Processes p and q are strongly equivalent, i.e. $p \sim q$, iff $\mathcal{L}(p) = \mathcal{L}(q)$.* □

Now, we focus on the formulae built from the propositions T, F, the modal operators $\langle a \rangle, [a]$ for $a \in \mathcal{A}$ and the logical connective $\wedge$. Let $\mathcal{L}_d$ be a set of formulae defined as

$$A ::= T \mid F \mid x \mid \langle a \rangle A \mid [a]A \mid A_1 \wedge A_2 \mid \nu x.A$$

where $x \in \mathcal{X}, a \in \mathcal{A}$. A relation $\leq_d$ on processes is defined by $p \leq_d q$ iff $p \models f$ implies $q \models f$, for all formulae $f \in \mathcal{L}_d$. Obviously, $\leq_d$ is a preorder and the resulting relation $\sim_d$, defined by $p \sim_d q$ iff $p \leq_d q$ and $q \leq_d p$, is an equivalence relation. So, $\leq_d$ turns out to be a partial order on the equivalence classes of processes with respect to $\sim_d$, i.e. $[p] \leq_d [q]$ iff $p \leq_d q$, where $[p] = \{q \mid p \sim_d q\}$.

Lemma 13 *$p \sim q$ implies $p \sim_d q$. But not vice versa.*

proof The implication is straightforward by the definition. The proper implication can be proved by the counter examples, e.g. $p = a.(b.\mathbf{0} + b.a.\mathbf{0} + b.a.b.\mathbf{0}) + a.a.\mathbf{0}$ and $q = a.(b.\mathbf{0} + b.a.\mathbf{0} + b.a.b.\mathbf{0}) + a.a.\mathbf{0} + a.(b.\mathbf{0} + b.a.\mathbf{0})$. □

Definition 14 *A relation R over processes is a simulation if $(p,q) \in R$ implies, for all $a \in \mathcal{A}$, if $p \xrightarrow{a} p'$, then there exists q' such that $q \xrightarrow{a} q'$ and $(p',q') \in R$.* □

Let $<$ be the union of all simulations.

To compare the expressive power of this relation $\leq_d$, we have the following results for *simulation preorder* $<$ [10], *testing preorder* $\sqsubseteq_{\mathrm{MUST}}$ [5] and 2/3 *bisimulation preorder* $\leq_{2/3}$ [3].

Lemma 15 *$p \leq_d q$ implies $p < q$. But not vice versa.*

proof The implication is an easy routine application of the definition. Properness is proved by the examples $p_2 = a.b.\mathbf{0}$, $q_2 = a.\mathbf{0} + a.b.\mathbf{0}$. Clearly, we have $p_2 < q_2$. But, $p_2 \not\leq_d q_2$. □

Lemma 16 *$\leq_d$ and $\sqsubseteq_{MUST}$ are mutually incomparable.* □

proof When $p_1 = a.a.\mathbf{0} + a.b.\mathbf{0}$ and $q_1 = a.a.\mathbf{0} + a.b.\mathbf{0} + a.(a.\mathbf{0} + b.\mathbf{0})$, we have $p_1 \leq_d q_1$, but $p_1 \not\sqsubseteq_{\mathrm{MUST}} q_1$. On the other hand, when $p_2 = a.a.\mathbf{0}$ and $q_2 = a.(a.\mathbf{0} + b.\mathbf{0}) + a.a.\mathbf{0}$, we have $p_2 \sqsubseteq_{\mathrm{MUST}} q_2$, but $p_2 \not\leq_d q_2$. □

Lemma 17 *$p \leq_d q$ implies $p \leq_{2/3} q$. But not vice versa.* □

proof The implication is immediately from definition of 2/3 bisimulation preorder. Its reverse direction has the following counter example. When $p = a.(b.c.\mathbf{0} + b.d.\mathbf{0})$ and $q = a.(b.c.\mathbf{0} + b.d.\mathbf{0}) + a.b.c.\mathbf{0}$, then $p \leq_d q$ but $p \not\leq_{2/3} q$.

We have also the following relation which is more distinguish than $\leq_d$.

Definition 18 *A binary relation $\sqsubseteq_d$ over processes is a maximum relation which satisfy the following. If $p \sqsubseteq_d q$, for all $a \in Act$,*

1. *whenever $p \xrightarrow{a} p'$, then there exists q' such that $q \xrightarrow{a} q'$ and $p' \sqsubseteq_d q'$;*
2. *whenever $q \xrightarrow{a} q'$, then there exists p' such that $p \xrightarrow{a} p'_1, \ldots, p \xrightarrow{a} p'_n$ and $p'_1 + \cdots p'_n \sqsubseteq_d q'$, where $n \geq 1$.* □

Lemma 19 *$p \sqsubseteq_d q$ implies $p \leq_d q$. But not vice versa.*

proof The implication is straightforward by the definition of $\sqsubseteq_d$. A counter example for reverse direction is $p = a.(a.\mathbf{0} + b.\mathbf{0}) + a.(c.\mathbf{0} + d.\mathbf{0})$ and $q = a.(a.\mathbf{0} + b.\mathbf{0}) + a.(c.\mathbf{0} + d.\mathbf{0}) + a.(a.\mathbf{0} + c.\mathbf{0})$. □

After all, we have the following relation.

$$< \supset \leq_{2/3} \supset \leq_d \supset \sqsubseteq_d$$

3 Synthesis algorithm for a recursive process

This section describes the algorithm that synthesizes a recursive process. For synthesis, formulae of μ-calculus are being regarded as specific properties of the synthesized process.

3.1 Enumeration of facts

An algorithm we will propose now is an inductive one. It generates an process which satisfies given facts, the properties of the target process, represented as formulae in μ calculus. Thus, input to the algorithm is an enumeration of formulae to be satisfied by the target process. Let p_o be the intended target process to be generated from its concrete properties. It should be noted that p_o is neither known initially nor given in a precise mannar.

Definition 20 *Let U be a set of pairs a formula $A \in \mathcal{L}$ and $+$ (or $-$), i.e. $\langle A, + \rangle$(or $\langle A, - \rangle$). $S = \{A \mid \langle A, + \rangle \in U\} \cup \{\neg A \mid \langle A, - \rangle \in U\}$ is an enumeration of facts, if either $\langle A, + \rangle$ or $\langle A, - \rangle$ always belong to U, and S is consistent over the deductive system STL($\mathcal{A}$)[1][4]. A element of S is also called an enumeration of facts.* □

[1] STL($\mathcal{A}$) is sound but unfortunately not complete. A complete deductive system for μ-calculus is not found yet.

Using p_o, the enumeration of facts may be defined as follows.

$$U = \{\langle A, + \rangle \mid p_o \models A, A \in \mathcal{L}\} \cup \{\langle A, - \rangle \mid p_o \not\models A, A \in \mathcal{L}\}$$

Since p_o is not known a priori, we must consider U as in Definition 20.

3.2 Synthesis algorithm

The algorithm generates a process which satisfies given facts. A fact of disjunctive form, e.g. $A \vee B$, or $\mu x.f(x)$, has a non-determinacy, so we can not avoid reconstruction of processes for such formulae. To remedy the difficulty, we forcus on the restricted set $\mathcal{L}_d$ of formulae.

Note that a formula with ν operator also have nondeterminacy uncertain information how many times loops of process branches unfold.

Given an enumeration of facts, the algorithm synthesize a process satisfying those facts. In the algorithm, a process is represented as a set of process definitions. Each process definition **rec** $c.p$ is associated with a set C of formulae, denoted as $c{:}C$, which must be satisfied by the corresponding process constant. C can be omitted when it is not important.

To describe the algorithm, we adopt a language like PROLOG[2], where I/O predicates can backtrack as well. For brief description, let c_i denote process constants associating with the process definitions $c_i \stackrel{\text{def}}{=} p_i$ or $c_i{:}C_i \stackrel{\text{def}}{=} p_i$ where C_i is a set of formulae. The initial state of a process is always fixed to c_0.

For the fact of the form $\nu x.f(x)$, it is important from which process constant is applied to the formula. In the case that it applied from c_i, it express by changing its variable x to x_i. Since the variable x is a bound variable, the meaning of the formula is not changed. And we assume further that we can identify the original formula $\nu x_i.f(x_i)$ from x_i. Also we adopt the following abbreviations.

$S[c_1{:}C_1 \stackrel{\text{def}}{=} p_1, \cdots, c_k{:}C_k \stackrel{\text{def}}{=} p_k]$: replacing the process definitions of $c_1, \cdots, c_k$ in S to $c_1{:}C_1 \stackrel{\text{def}}{=} p_1, \cdots, c_k{:}C_k \stackrel{\text{def}}{=} p_k$, or adding $c_i{:}C_i \stackrel{\text{def}}{=} p_i$ to S if $c_i \notin S$.

$S\{x/y\}$: substituting x to a free variable y in S.

Algorithm 21 (Synthesis algorithm)

Input: Enumeration of facts $A_1, A_2, \cdots$. It is an enumeration of formulae to be satisfied a synthesized process. The order of them is arbitrary.

Output: Sequence of processes $p_1, p_2, \cdots$. Each p_k satisfies whole inputted formulae A_1 to A_k.

mpstart :- mp($\{c_0{:}\{T\} \stackrel{\text{def}}{=} \mathbf{0}\}$).
% Set an initial process to $\mathbf{0}$.

mp(S) :- % S is a set of process definitions.
read-formula(A), % Input a formula.
makeproc(c_0, S, A, X), % Synthesize a process from A.
write-process(X), % Output above result.
mp(X). % Go to next step.

% makeproc(c, S, A, X)
% c is a current process constant.
% S is a set of process definitions.
% A is a formula applied to c.
% X is a synthesized process, set of process definitions.

% T : Return S since T is satisfied any processes.
makeproc(c_i, S, T, S).
% F : Since F means inputted formulae are
% inconsistent, backtrack.
% x_j : This is a variable of $\nu x_j.f(x_j)$
makeproc(c_i, S, x_i, S).
% No action if there is already a loop.
makeproc(c_i, S, x_j, X) :- % When $i \neq j$.
$S' \leftarrow (S[c_j{:}C_j \stackrel{\text{def}}{=} p_j + p_i] -$
$\{c_i{:}C_i \stackrel{\text{def}}{=} p_i\})\{x_j/x_i\}\{c_j/c_i\}$,*% c_i become to c_j.*
makeproc($c_j, S', \wedge\{p : p \in C_i\}, X$).
% Apply every formula in C_i to c_j. (a)
makeproc(c_i, S, x_j, X) :-
% When the above predicate occurs inconsistent.
is-remake,
% Judge whether backtraking is allowed or not.
makeproc($c_i, S, \nu x_j.f(x_j), X$).
% Apply $\nu x_j.f(x_j)$ to c_i. (b)
% $\langle a \rangle B$: Make a branch labeled a, and add a process
% satisfying B after it.
makeproc($c_i, S, \langle a \rangle B, X$) :-
$\exists(c_i \xrightarrow{a} c_j)$ *such that makeproc(c_j, S, B, X).*
% Let c_j satisfy B.
makeproc($c_i, S, \langle a \rangle B, X$) :-
get-new-process-constant(c_j),
% Get a fresh process constant c_j.
makeproc($c_j, S[c_i{:}C_i \stackrel{\text{def}}{=} p_i + a.c_j, c_j{:}\{T\} \stackrel{\text{def}}{=} \mathbf{0}], B \wedge (\wedge\{f_k : [a]f_k \in C_i\}), X$).
% where $\wedge\emptyset \stackrel{\text{def}}{=} T$. (c)
% $[a]B$: Let every processes after acting a satisfy B
makeproc($c_i, S, [a]B, S$) :-
$\models \wedge C_i \supset [a]B$
% No action if $[a]B$ have already satisfied.
makeproc($c_i, S, [a]B, S[c_i{:}C_i \cup \{[a]B\} \stackrel{\text{def}}{=} p_i]$) :-
$c_i \not\xrightarrow{a}$. *% Only add $[a]B$ in C_i if c_i cannot act a.*
makeproc($c_i, S, [a]B, X$) :-
$\forall c_j.c_i \xrightarrow{a} c_j$,
makeproc($c_j, S[c_i{:}C_i \cup \{[a]B\} \stackrel{\text{def}}{=} p_i], B, X$).
% Let c_j satisfy B.
% $B_1 \wedge B_2$: Apply B_1 in first and then do B_2.

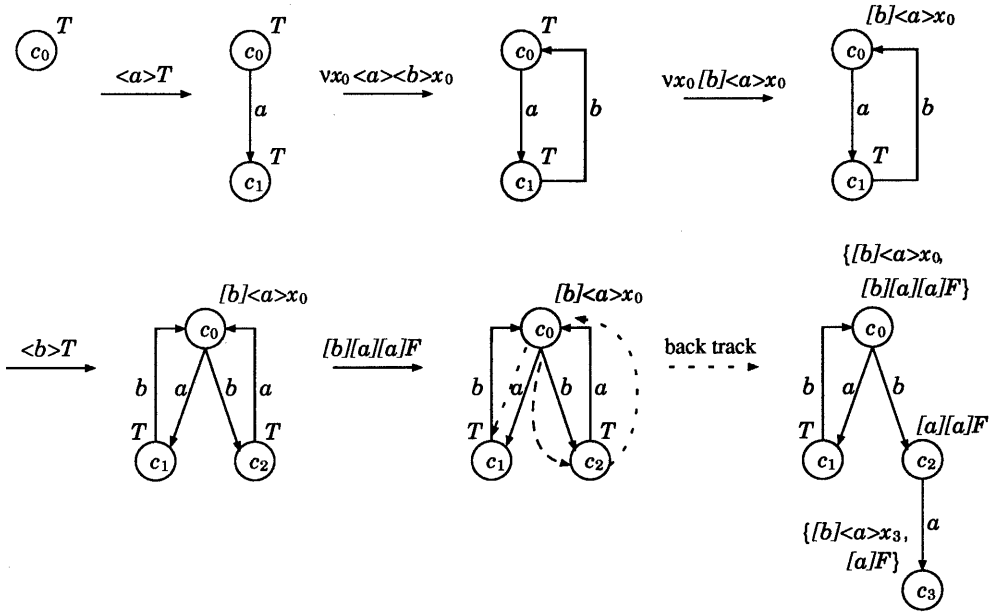

Figure 1: Some action sequences are possible since a loop is constructed.

$makeproc(c_i, S, B_1 \wedge B_2, X)$:-
  $makeproc(c_i, S, B_1, Y)$,
  $makeproc(c_i, Y, B_2, X)$.
% $\nu x.f(x)$: *Modify bound variable* x *to* x_i *to adjust it*
% *to* c_i.
$makeproc(c_i, S, \nu x.f(x), X)$:-
  $makeproc(c_i, S, f(x_i), X)$.

% is-remake : Judge whether backtraking is allowed or not.
is-remake :-

The predicate makeproc makes as a short loop as possible from a formula $\nu x.f(x)$. *A shorter loop makes inconsistent in the following cases. Thus, the loop may be extended for suitable times by backtracking. Unless the following, extending loop may arise that makeproc do not terminate. This is-remake decide whether backtracking is allowed or not. In fact, only either the cases of Fig.1 or 2 can be allowed. In these figures, each circle labeled* c_i *represents process constant* c_i. *A set of formulae labeled each process definition is described at a corner of each circle. We omit brackets when the set has only one element. Between circles, there are arrows labeled an action. These mean transition relation between them. Vartical arrows mean state transition of the algorithm when each labeled formula is entered.*

In the case of Fig.1, is-remake trace the path which is passed by a formula occurring inconsistent. And backtracking is allowed if the path has one or more loops and dose not end on these loops. In the case of Fig.2, is-remake trace the path same as the previous case. And it is allowed if the path enters the begging of a loop, but dose not go to body of the loop, and exits the loop directly. In each case, the information about which formula made each branch is needed.

□

Theorem 22 *Assume there exists a process* p_n *satisfying the initial segment* $A_1, \cdots, A_n$ *of an enumeration of fact, where* $n \geq 1$. *Assume also Algorithm 21 outputs a set of process definitions* S_{n-1} *for the* $n-1$ *facts* $A_1, \cdots, A_{n-1}$. *For the n-th fact* A_n, *we have the followings.*

1. *Algorithm 21 terminates and returns an output, a set of process definitions* S_n *with a process constant* c_0, *the initial state of* S_n.
2. c_0 *with* S_n *satisfies* $A_1, \cdots, A_n$. □

proof 1. The predicate *makeproc* call itself recursively. Only in the cases of definitions (a) to (c) of Algorithm 21, the size of a inputted formula — the number of operators constructing the formula — may be greater

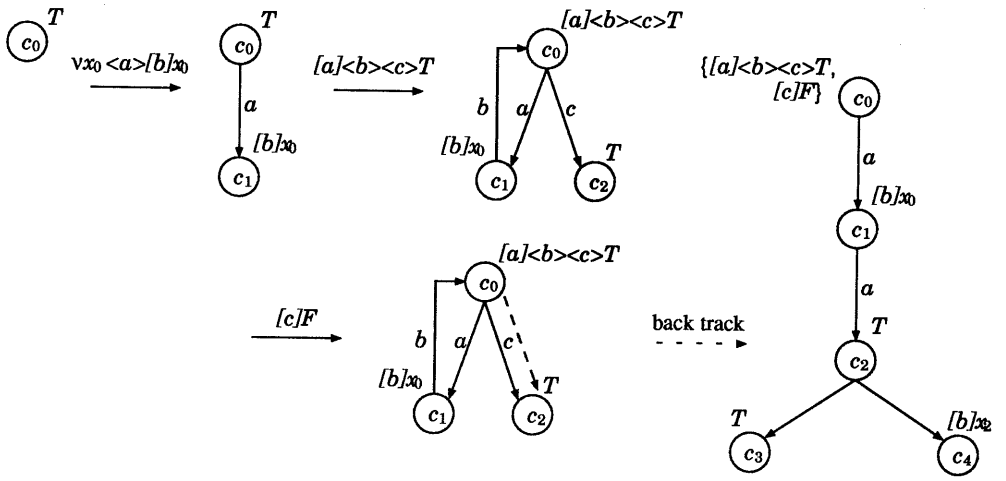

Figure 2: Some action sequences are possible since a branch modified to a loop.

than the size of the formula which is used in the recursive call. Therefore, the algorithm terminate necessarily when a process is synthesized without using these definitions. In the following, we consider of the definitions (a) to (c). Intuitionally, what the algorithm may not terminate by (a) to (c) have the following meaning.

(i) Sets of process definitions are applied as if chain reaction. (corresponding to (c)).
When the algorithm add a new branch from a process definition, the process after the branch must satisfy formulae of the set of formulae labeled the process definition. It is solved in (c) by applying $\wedge\{f_k : [a]f_k \in C_i\}$ to c_j. From this process, another process definition is added a new branch, and then the algorithm may arrive at (c) again. If these arise continuously, the predicate *makeproc* will not terminate.

(ii) Process reconstructing arise infinitely many times. (corresponding to (a) and (b)).
The definition (a) makes a loop to satisfy a formula with a ν operator. It makes as short loop as possible, but may arise inconsistent. In this case, the loop must be extended for suitable times by backtrack. This process may arise infinite many times, that is, a branch which have infinite depth are synthesized.

Firstly, when every inputted formulae has no ν operators, both of definitions (a) and (b) are not called. The definition (c) get a new process constant, and then a process are synthesized from the process constant. Since each inputted formula has no process variable, the algorithm dose not make a loop to existing node for it. Therefore each size of formulae used by recursive call after (c) less than the size of the formula. Consequently, the algorithm will terminate when formulae without ν are inputted only.

Next, we assume that there exist more than one formulae with ν in inputted formulae. It is enough that we consider the above (i) and (ii). In first, we assume one of (i) or (ii) will arise, and the other will not.

- Only (i) arise.
  The predicate *makeproc* may call itself recursively. We show this recursive call terminate in finite times.

  Since the size of each inputted formulae is finite, the size of each elements of a set of formulae labeled each process definitions is also finite. From the previous lemma, *makeproc* make neither new branches nor process definitions from a already inputted formula. Then a set of formulae can transmit within only finite range. Therefore transmition of formulae will be saturated, and then the recursive call of *makeproc* terminate.

- Only (ii) arise.
  The definition (c) unwind a loop once. For (ii), it need that there exists some formulae which negate the loop infinite many times and satisfy the condition of *is-remark*. If these formulae have no ν operators, they can not negate infinitely. Though they have, the part of formulae which negate the

loop, i.e. the formula F, occur periodically. Therefore they can not negate only the loop. After all, *makeproc* will terminate.

- Both (i) and (ii) arise.
  From the previous arguments, we know that only ether (i) or (ii) can not arise. Thus we assume both (i) and (ii) arise alternatively. Since the loop is unwound infinitely, the process which is made from the formulae has a branch with infinite length. For the assumption, it need that there exist a formula which negate the loop with the top of the branch.Remember that a set of formulae can transmit within only finite range if they have no ν operators. From this fact, there exist a formula which have one or more ν operators and negate the loop. Thus this argument conclude the previous one, and then we show that the algorithm terminate.

2. Omitted. □

The algorithm is a non terminating procedure. Therefore, we show its validity by using concept of convergence in the limit, which is often used in inductive learning theory.

Definition 23 *Assume an algorithm inputs an enumeration of facts, and outputs processes sequentially. After some time if the output process is always p, then the inferred sequence by this algorithm converges in the limit to p over the enumeration of facts.* □

The validity of Algorithm 21 is also shown by the following theorem.

Theorem 24 *Under the assumption of algorithm 21, if there exists a process p satisfying an enumeration of facts, the inferred sequence of processes by Algorithm 21 converges in the limit to a process p' such that $p \leq_d p'$.* □

4 Conclusion

This paper presented the synthesis algorithm for a recursive process based on the enumeration of facts, which must be satisfied by the process. And its validity is also discussed.

The algorithm, however, restrict formulae within $\mathcal{L}_d$. To be a complete algorithm, whose infered sequence of processes converges in the limit to a process which is equivalent to a target one, we must remove the restriction. As we mentioned, a formula with ether $\vee$ or μ operator have uncertain information. For example, consider a formula such as $\langle a\rangle T \vee \langle b\rangle T$. The synthesis algorithm is uncertain which formula is a really needed fact. Suppose that the algorithm trust one of them (e.g. $\langle a\rangle T$) and output a process p which satisfy the formula. And after some times, $[a]F(\equiv \neg\langle a\rangle T)$ may be inputted. Then, it must return at the point before p was synthesized, and adopt the other formula (i.e. $\langle b\rangle T$). Especially, a formula with *mu* operator has infinite many $\vee$ operators (see Proposition11). Thus it may cause to backtrack infinitely. To solve this problem, the predicate *is-satisfied* will be more complicate. And the time or space complexity of the algorithm is not discussed, which is left for a future study.

References

[1] Angluin D.: "Learning Regular Sets from Queries and Counterexamples", Inf. and Comput, **75**, pp.87–106(1987).

[2] Clocksin W.F., Mellish C.S.: "Programming in Prolog", Springer-Verlag(1981).

[3] Fantechi A., Gnesi S., Ristori G.: "Compositional Logic Semantics and LOTOS", Protocol Specification, Testing and Verification, XL., IFIP, pp.365–378

[4] Graf S., Sifakis J.: "A Logic for the Description of Non-deterministic Programs and Their Properties", Inf. and contr., **68**, pp.254–270(1986)

[5] Hennessy M.: "Algebraic Theory of Processes", J. ACM., **32**, 1, pp.137–161(1985).

[6] Hoare C.A.R.: "Communicating Sequential Process", Prentice Hall(1985).

[7] Kimura S., Togashi A., Noguchi S.: "A Synthesis Algorithm of Basic Processes by Modal Formulas" (in Japanese), Trans. IEICE, **J75-D-I**, pp.1048–1061(1992).

[8] Kimura S., Togashi A., Noguchi S.: "Synthesis of Algebraic Processes Based on Process Enumeration" (in Japanese), Trans. IEICE, **J75-D-I**, pp.1132–1143(1992).

[9] Kozen D.: "Results on the Propositional μ-calculus ", Theoret. Comput. Sci., **27**, pp.333–354(1983).

[10] Milner R.: "Communication and Concurrency", Prentice-Hall(1989).

[11] Shapiro E.Y.: "Inductive Inference of Theories From Facts", Technical Report 192, Yale Univ(1981).

[12] Stirling C.: "An Introduction to Modal and Temporal Logics for CCS", Lecture Notes in Comput. Sci. **491**, Springer-Verlag, pp.2–20(1991).