京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

京都大学 KYOTO UNIVERSITY

Title	Inductive Synthesis of Recursive Processes from Logical Properties
Author(s)	Kimura, Shigetomo; Togashi, Atsushi; Shiratori, Norio
Citation	数理解析研究所講究録 (1995), 902: 80-102
Issue Date	1995-03
URL	http://hdl.handle.net/2433/59387
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

Kyoto University

数理解析研究所講究録
902 巻 1995 年 80-102

Inductive Synthesis of Recursive Processes from Logical Properties *

Shigetomo Kimura (木村 成伴), Atsushi Togashi (富樫 敦) and Norio Shiratori (白鳥 則郎)

2-1-1 Katahira, Aoba-ku, Sendai, Japan.

Research Institute of Electrical Communication, Tohoku University.

〒 980-77 仙台市青葉区片平 2-1-1

東北大学電気通信研究所

e-mail : {kimura,togashi,norio}@shiratori.riec.tohoku.ac.jp

1 Introduction

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 [5, 15], CCS by Milner [19] and ACP by Bergstra and Klop [2]. In Feb. 1990, ISO adopted LOTOS [4] as the international standard for OSI specification description language. 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 such as non-determinacy and concurrency, so their operational semantics are completely different from those of the traditional automata and formal languages.

In the formal specification, modal or temporal logic are studied to express constrains or to verify a specification. For example, [8] and [9] used temporal logics to verify that specifications had good properties like 'safety' or 'liveness', and did not have bad properties such as 'deadlock'.

From opposite point of view, we can regard these formulae as facts, which a specification must satisfy. And we seem to be able to infer a specification from facts by using learning paradigms. Inductive inference is one of the learning paradigms and suitable for our purpose since an exact specification satisfying input facts can be inferred by means of inductive inference. Therefore, the inductive inference of the processes forms a basis for automatic synthesis of highly reliable communicating protocols and concurrent programs from the examples or the required properties. However, little has been investigated for the inductive inference of concurrent processes due to the difficulties arising from the process features such as non-determinacy and concurrency.

We have already presented the algorithm that inductively synthesizes a basic process in a subclass of CCS from concrete examples expressed in modal formulae which describe

*A part of this study is supported by Grants from the Asahi Glass Foundation and Research Funds from Japanese Ministry of Education. A preliminary version of this paper was published in the Proceedings of the fifth International Workshop on Algorithmic Learning Theory. LNCS, 1994

the properties of the process and have demonstrated the validity and improvement of the approach [16]. However, the expressive power of basic processes is weak. They cannot express the recursive behaviors of a system. It remains to propose a synthesis algorithm for recursive processes.

This paper presents an inductive synthesis algorithm for a recursive process. To synthesize a process, facts, which must be satisfied by the target process, is given to the algorithm one by one since such facts are infinitely many in general. When n facts are input to the algorithm, it outputs a process which satisfies the input n facts. And this generation process is repeated infinitely many times. To represent facts of a process, we adopt a subcalculus of μ-calculus [11, 17, 25], which can describe recursive behaviors. The correctness of the algorithm can be stated that the output sequence of processes by the algorithm converges to a process, which cannot be distinguished from the intended one (if we could know it) by a given enumeration of facts, in the limit.

In fact, the problem to synthesize a process is regarded as a satisfiability problem for logical calculi. The satisfiability problem is a decision problem to determine whether or not a given formula in the logic has a model. For example, Kozen [17] provided a tableau method for the μ-calculus. We will compare our method with related works and discuss the reason why we employ inductive inference to synthesize a process in detail in Section 6.

The outline of the paper is as follows: Section 2 presents the algebraic formulation of processes, together with μ-calculus. Section 3 discusses the discriminative power of a subcalculus of μ-calculus. Section 4 gives an algorithm that synthesizes a process satisfying a given enumeration of facts. Section 5 introduces a prototype system SORP (Synthesizer of Recursive Processes) based on the synthesis algorithm. The paper is concluded in Section 6, where related works and future problems are briefly discussed.

2 Preliminaries

In this section, we briefly review the preliminary notions such as algebraic processes and μ-calculus. See [11,13,15,17,19,25] 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 2.1 *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 *bound.* We say p is a *scope* of $\mathbf{rec}\, c.$ in $\mathbf{rec}\, c.p$. The occurrence of a process constant which is not within any scope of $\mathbf{rec}\, c.$ is called *free.* When every free occurrence of c is within some subterm $a.q$ of p, we call c is *guarded* in p. When every constant in p is guarded, p is also called *guarded.* If every occurrence of any process constant in p is bound, p is called *closed.* Otherwise it is called *open.* Closed terms are called *(recursive) processes.* Let $\mathcal{P}$ denote the set of all 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 the same as α-conversion in λ-calculus [14]. 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 a recursive term is given by *a labeled transition system* with actions as labels.

Definition 2.2 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 $s \xrightarrow{a} s'$. Thus, the transition relation can be written as $\rightarrow = \{\xrightarrow{a} \mid a \in \mathcal{A}\}$. $s \xrightarrow{a} s'$ may be interpreted as "in the state s an action a can be performed and after the action the state moves to s'". s' is called an *a-successor* of s. We use the usual abbreviations as e.g. $s \xrightarrow{a}$ for $\exists s' \in S$ s.t. $s \xrightarrow{a} s'$ and $s \not\xrightarrow{a}$ for $\neg\exists s' \in S$ s.t. $s \xrightarrow{a} s'$.

Definition 2.3 A transition relation on recursive terms is given by the following transition rules:

$$\frac{}{a.p \xrightarrow{a} p} \qquad \frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'} \qquad \frac{p\{\mathbf{rec}\, c.p/c\} \xrightarrow{a} p'}{\mathbf{rec}\, c.p \xrightarrow{a} p'}$$

where $p\{q/c\}$ is p except any free occurrences of c are replaced by q. □

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 [19,20].

Definition 2.4 A relation R over recursive terms 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 $(p', q') \in R$. □

Recursive terms 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.

Proposition 2.5 [12,19] *The following equations are satisfied on processes.*

1. $p + q \sim q + p$.
2. $p + (q + r) \sim (p + q) + r$.
3. $p + p \sim p$.
4. $p + \mathbf{0} \sim p$. □

This proposition can be easily extended for recursive terms. It is known that the equivalence given by the above proposition is sound and complete over strong equivalence, when only basic processes constructed by inaction, action prefix and summation, are considered [12]. Using this result, it is shown that any basic processes p can be equivalently transformed into a process of the following form $a_1.p_1 + \cdots + a_n.p_n (\stackrel{\text{def}}{=} \sum_{i=1}^{n} a_i.p_i)$. The order of $a_i.p_i$ is immaterial from equation 1 and 2. When $n = 0$, it is understood that $\sum_{i=1}^{n} a_i.p_i = \mathbf{0}$. In the following, based on that, we always assume both the commutative and associative law of $+$ on recursive terms to avoid heavy use of brackets. Furthermore, by definition, a process constant without its definition is strongly equivalent to $\mathbf{0}$. Therefore, without loss of generality, to the rest of this paper, we are concerned with only processes rather recursive terms.

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 [11, 17, 25], which includes a modality concerning actions, in order to describe dynamic properties of processes. It is an extension of Hennessy-Milner logic [12] to express recursive properties. So we will introduce it to represent facts of a process for the synthesis algorithm.

Definition 2.6 *formulae* in μ-calculus are defined inductively as follows:

1. $\mathbf{tt}$ (true) is a formula.
2. A variable $x \in \mathcal{X}$ is a formula, where $\mathcal{X}$ is a denumerable set of *logical variables.*
3. If f and f' are formulae, $f \vee f'$ and $\neg f$ 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 notion of freeness, boundness and scope for formulae in μ-calculus are defined similarly to the one for recursive terms or λ-calculus. 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. A formula f is sometime written as $f(x)$ to express the free occurrence of x in f. $f(g)$ denotes the resulting $f(x)$, where every free occurrence of x is replaced by g. In the replacement $f(g)$, every free occurrence of a variable in g is not bound in $f(g)$ by means of renaming bound variables. Recall that $\mathcal{P}$ is the set of all processes. Let $\mathcal{V} : \mathcal{X} \rightarrow 2^{\mathcal{P}}$ be a valuation, which assigns a set of processes to be satisfied to each variable, where $2^{\mathcal{P}}$ is the power set of $\mathcal{P}$. We adopt conventional notation $\mathcal{V}[S/x]$, which is the valuation $\mathcal{V}'$ that agrees with $\mathcal{V}$ except that $\mathcal{V}'(x) = S$.

The set of all closed formulae is written as $\mathcal{L}$. When a process p satisfies a formula f in a valuation $\mathcal{V}$, it is written as $p \models_{\mathcal{V}} f$. The symbol $\equiv$ is used to denote logical equivalence, i.e. $f \equiv f'$ means that $p \models_{\mathcal{V}} f$ iff $p \models_{\mathcal{V}} f'$ for all process p and for all valuations $\mathcal{V}$.

Definition 2.7 Let p be any process. Satisfaction relation of formulae in a valuation $\mathcal{V}$ is defined as follows:

1. $p \models_{\mathcal{V}} \mathbf{tt}$.
2. $p \models_{\mathcal{V}} x$ if $p \in \mathcal{V}(x)$.
3. $p \models_{\mathcal{V}} f_1 \vee f_2$ if $p \models_{\mathcal{V}} f_1$ or $p \models_{\mathcal{V}} f_2$.
4. $p \models_{\mathcal{V}} \neg f$ if $p \not\models_{\mathcal{V}} f$, where $p \not\models_{\mathcal{V}} f$ means that p does not satisfy f.
5. $p \models_{\mathcal{V}} \langle a \rangle f$ if there exists some q such that $p \xrightarrow{a} q$ and $q \models_{\mathcal{V}} f$.
6. $p \models_{\mathcal{V}} \mu x.f(x)$ if $p \in S$ for all $S \subseteq \mathcal{P}$ such that $\forall q \in \mathcal{P}.q \models_{\mathcal{V}[S/x]} f(x)$ implies $q \in S$. □

Note that a valuation is immaterial for close formulae f in the sense that $p \models_{\mathcal{V}} f$ for some $\mathcal{V}$ iff $p \models_{\mathcal{V}'} f$ for all $\mathcal{V}'$. In the following, $\models_{\mathcal{V}}$ is abbreviated as $\models$ if there is no conflict about its valuation.

Definition 2.8 The following logical notations are used for convenience:

1. $\mathbf{ff} \stackrel{\text{def}}{=} \neg\mathbf{tt}$.
2. $f_1 \wedge f_2 \stackrel{\text{def}}{=} \neg(\neg f_1 \vee \neg f_2)$.
3. $[a]f \stackrel{\text{def}}{=} \neg\langle a \rangle \neg f$.
4. $\nu x.f(x) \stackrel{\text{def}}{=} \neg \mu x.\neg f(\neg x)$. □

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

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

Our definition of μ-calculus differs from that of STA($\mathcal{X}, \mathcal{A}$) in [11]. Fortunately, each system has same expressive power when $\mathcal{A}$ is finite and only guarded formulae are concerned. The set of all formulae in STA($\mathcal{X}, \mathcal{A}$) is defined in the following BNF:

$$f ::= \mathbf{tt} \mid \text{Nil} \mid x \mid Af \mid f + f \mid f \vee f \mid \neg f \mid \mu x.f \quad \text{where } x \in \mathcal{X} \text{ and } A \subseteq \mathcal{A}.$$

Definition 2.9 Let p be any process. Satisfaction relation of a formula in STA($\mathcal{X},\mathcal{A}$) in a valuation $\mathcal{V}$ is defined as follows:

1. $p \models_{\mathcal{V}} \mathrm{Nil}$ if $p \sim \mathbf{0}$.
2. $p \models_{\mathcal{V}} Af$ if $\exists p_i$ and $a_i \in A$ $(1 \leq i \leq n)$ such that $p \sim \sum_{i=1}^{n} a_i.p_i$ and $p_i \models_{\mathcal{V}} f$ for each p_i.
3. $p \models_{\mathcal{V}} f_1 + f_2$ if $\exists p_1$ and p_2 such that $p_1 \models_{\mathcal{V}} f_1$, $p_2 \models_{\mathcal{V}} f_2$ and $p \sim p_1 + p_2$.
4. The satisfaction relation for other syntactical constructs is defined in the exactly same way as in Definition 2.7. □

Proposition 2.10 [11]

1. *$f + f \equiv f$ for f of the form Af',* $\mathbf{tt}$, Nil.
2. *$f + \mathrm{Nil} \equiv f$.*
3. *$\emptyset f \equiv \mathbf{ff}$.*
4. *$(A_1 \cup A_2)f \equiv A_1 f \vee A_2 f \vee (A_1 f + A_2 f)$.*
5. *$f_1 + (f_2 \vee f_3) \equiv (f_1 + f_2) \vee (f_1 + f_3)$.*
6. *$\neg(Af + \mathbf{tt}) \equiv (\mathcal{A} - A)\mathbf{tt} \vee A\neg f \vee ((\mathcal{A} - A)\mathbf{tt} + A\neg f) \vee \mathrm{Nil}$.*
7. *$\neg(\sum_i A_i f_i) \equiv \neg(\sum_i A_i f_i + \mathbf{tt}) \vee [(\bigwedge_i((\mathcal{A} - A_i)\mathbf{tt} \vee A_i \neg f_i)) + \mathbf{tt}]$.*
8. *$\mu x.C_1[\{a\}C_2[x + f]] \equiv \mu x.C_1[\{a\}C_2[\mu y.(C_1[\{a\}C_2[y]] + f)]]$ for any formula f and contexts $C_1[]$ and $C_2[]$ which are formulae with the hole $[]$.*

proof We prove only 8 since it is not included in [11]. It roughly means a variable appearing as a summand of some subterm can be eliminated. Since we are concerned with guarded formulae, any equation of the form $x = f(x)$ has a unique least fixed point, i.e. $\mu x.f(x)$. Let $\hat{x} = \mu x.C_1[\{a\}C_2[x + f]]$, the unique least fixed point of the equation $x = C_1[\{a\}C_2[x + f]]$. Thus we have $\hat{x} = \hat{C}_1[\{a\}\hat{C}_2[\hat{x} + \hat{f}]]$, where $\hat{t} = t\{\hat{x}/x\}$ for $t = C$ (a context) or $t = f$ (a formula). Let $\hat{y} = \hat{x} + \hat{f}$. Then $\hat{y} = \hat{C}_1[\{a\}\hat{C}_2[\hat{y}]] + \hat{f}$. So $\hat{y}$ is the fixed point of the equation $y = \hat{C}_1[\{a\}\hat{C}_2[y]] + \hat{f}$. Therefore $\hat{x}$ is the fixed point of the equation $x = C_1[\{a\}C_2[\mu y.C_1[\{a\}C_2[y]] + f]]$. Hence we get the result. □

For example,

$$\begin{aligned}
\mu x.\{a\}(x + \{b\}\mathrm{Nil}) &\equiv \mu x.\{a\}\mu y.(\{a\}y + \{b\}\mathrm{Nil}) \\
&\equiv \{a\}\mu y.(\{a\}y + \{b\}\mathrm{Nil}), \\
\mu x.\{a\}\mu y.(\{b\}(x + y + \{c\}\mathrm{Nil})) &\equiv \mu x.\{a\}\mu y.\{b\}\mu z.\mu w.(\{b\}w + \{a\}\{b\}z + \{c\}\mathrm{Nil}) \\
&\equiv \{a\}\{b\}\mu z.\mu w.(\{b\}w + \{a\}\{b\}z + \{c\}\mathrm{Nil}).
\end{aligned}$$

It is easy to see that μ-calculus is embedded into STA($\mathcal{X},\mathcal{A}$). The diamond operator $\langle a\rangle$ in μ-calculus is not provided in STA($\mathcal{X},\mathcal{A}$). However if we define $\langle a\rangle f \stackrel{\mathrm{def}}{=} \{a\}f + \mathbf{tt}$ as

in [11], the inclusion is obvious. The opposite direction is not so trivial. We can show the other inclusion when $\mathcal{A}$ is finite and a formula is guarded. From Proposition 2.10, we can assume any summand (an operand of $+$) is either of the form $\mathbf{tt}$ or $\{a\}f$, where $a \in \mathcal{A}$, by the following algorithm since each formula is guarded

1. Remove any $\neg$ operators by logical connective rules and Proposition 2.10.6 and 7.
2. Eliminate any variables included in a summand by Proposition 2.10.8.
3. Convert any action set of each prefix operator to a singleton set by Proposition 2.10.3 and 4.
4. Delete $\vee$ and Nil operators within summand by Proposition 2.10.2 and 5.

Then we can show a translation function from STA($\mathcal{X},\mathcal{A}$) to μ-calculus.

Definition 2.11 A translation function $H(f)$ from a formula f in STA($\mathcal{X},\mathcal{A}$) into the one in μ-calculus is defined in the following:

1. $H(\mathbf{tt}) = \mathbf{tt}$.
2. $H(\text{Nil}) = \bigwedge_{a\in\mathcal{A}}[a]\mathbf{ff}$.
3. $H(x) = x$.
4. $H(\neg f) = \neg H(f)$.
5. $H(f_1 \vee f_2) = H(f_1) \vee H(f_2)$.
6. $H(\mu x.f) = \mu x.H(f)$.
7. $H(\{a\}f) = \langle a\rangle H(f) \wedge [a]H(f) \wedge \bigwedge_{c\in\mathcal{A}-\{a\}}[c]\mathbf{ff}$.
8. $H((\sum_{i\in I}\{a_i\}f_i) + \mathbf{tt}) = \bigwedge_{i\in I}\langle a_i\rangle H(f_i)$ where I is a finite index set.
9. $H(\sum_{i\in I}\{a_i\}f_i) = (\bigwedge_{i\in I}\langle a_i\rangle H(f_i)) \wedge (\bigwedge_{i\in I}[a_i]\bigvee_{a_i=a_j} f_j) \wedge (\bigwedge_{a\in\mathcal{A}-A}[a]\mathbf{ff})$ where I is a finite index set and $A = \{a_i \mid i \in I\}$. □

Let $\models_{\mathcal{V},\text{STL}(\mathcal{A},\mathcal{X})}$ ($\mathcal{V}$ is sometimes omitted as $\models_{\text{STL}(\mathcal{A},\mathcal{X})}$) be a satisfaction relation for formulae of STL($\mathcal{A},\mathcal{X}$). Then we have the following results.

Lemma 2.12 *Let p be a process and f a formula in* STL($\mathcal{A},\mathcal{X}$). *Then $p \models_{\mathcal{V},\text{STL}(\mathcal{A},\mathcal{X})} f$ iff $p \models_{\mathcal{V}} H(f)$.* □

Proposition 2.13 *When $\mathcal{A}$ is finite and a guarded formulae are only concerned, μ-calculus has same expressive power with STA($\mathcal{X},\mathcal{A}$).* □

The following two propositions, the same results for STA($\mathcal{X},\mathcal{A}$), can be proved by Proposition 2.13.

Proposition 2.14 *Let $f(x)$ be a guarded formula. Then the followings are satisfied:*

1. $\mu x.f(x) \equiv \bigvee_{k>0} f^k(\mathbf{ff})$.

2. $\nu x.f(x) \equiv \bigwedge_{k>0} f^k(\mathbf{tt})$. □

Proposition 2.15 *Processes p and q are strongly equivalent, i.e. $p \sim q$, iff $\mathcal{L}(p) = \mathcal{L}(q)$.* □

The next proposition shows that the negation can be removed from a formula.

Proposition 2.16 *Any formula can be equivalently converted to a formula without negation, i.e. a formula built up with* $\mathbf{tt}$, $\mathbf{ff}$, $\wedge$, $\vee$, $\langle a\rangle$, $[a]$, μ, *and* ν. □

From now on, we will consider closed formulae without negation.

3 A subcalculus of μ-calculus

In section 4, an inductive synthesis algorithm for recursive processes is introduced. The algorithm generates a process which satisfies given formulae. However, a formula of disjunctive form, e.g. $f \vee g$, or $\mu x.f(x)(\equiv \bigvee_{k>0} f^k(\mathbf{ff}))$ is ambiguous to our purpose, i.e. synthesis of processes. Consider the formula $\langle a\rangle\mathbf{tt} \vee \langle b\rangle\mathbf{tt}$. It says the target process can execute either a or b (or both). When it is input to the synthesis algorithm, the algorithm is unsure which formula, i.e. $\langle a\rangle\mathbf{tt}$ or $\langle b\rangle\mathbf{tt}$, is really needed. Suppose that the algorithm trusts $\langle a\rangle\mathbf{tt}$ and outputs a process p which satisfies it. But after some times, $[a]\mathbf{ff}(\equiv \neg\langle a\rangle\mathbf{tt})$ may be input. In such case, the algorithm must backtrack at the point before p was synthesized, and adopt the other formula (i.e. $\langle b\rangle\mathbf{tt}$). Especially, since a formula with μ operator has infinite many $\vee$ operators (see Proposition2.14), it may cause backtracking infinite many times. To remedy the difficulty, we focus on the formulae without $\vee$ and μ operators. Let $\mathcal{L}_d$ be the set of all formulae defined in the following BNF:

$$f ::= \mathbf{tt} \mid \mathbf{ff} \mid x \mid \langle a\rangle f \mid [a]f \mid f \wedge f \mid \nu x.f$$

where $x \in \mathcal{X}, a \in \mathcal{A}$. A relation $\leq_d$ on recuresive process is defined by $p \leq_d q$ iff $p \models_{\mathcal{V}} f$ implies $q \models_{\mathcal{V}} f$ for all formulae $f \in \mathcal{L}_d$ and for all valuations $\mathcal{V}$. Note that only closed formulae suffice to define $\leq_d$. 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 recursive process with respect to $\sim_d$, i.e. $[p] \leq_d [q]$ iff $p \leq_d q$, where $[p] = \{p' \mid p \sim_d p'\}$.

Lemma 3.1 *For a formula $f \in \mathcal{L}_d$, recursive processes p, q and a valuation $\mathcal{V}$, we have the following claims:*

1. $p \models_{\mathcal{V}} f$ *and* $q \models_{\mathcal{V}} f$ *imply* $p + q \models_{\mathcal{V}} f$.

2. $a.p + a.q \models_{\mathcal{V}} [a]f$ *implies* $a.(p+q) \models_{\mathcal{V}} [a]f$. □

Proposition 3.2 *For any processes p, p', q, r and any action a, the followings are satisfied:*

1. $p \leq_d q \iff a.p \leq_d a.q$.
2. $p \leq_d q \Longrightarrow p + r \leq_d q + r$.
3. *If* $r \not\xrightarrow{a}$ *then* $a.p + r \leq_d a.p' + r \iff a.p + a.q + r \leq_d a.p' + a.q + r$.
4. $a.p + a.q \leq_d a.p + a.q + a.(p + q)$. □

Lemma 3.3 $p \sim q$ *implies* $p \sim_d q$. *But not vice versa.* □

As the discriminative power of this relation $\leq_d$, we have the following result on comparison with the *ready simulation preorder* $\leq_{RS}$ [10].

Definition 3.4 A *ready simulation preorder* is a binary relation R on processes such that whenever $(p, q) \in R$ and $a \in \mathcal{A}$ then:

1. if $p \xrightarrow{a} p'$ then $\exists q'.q \xrightarrow{a} q'$ and $(p', q') \in R$.
2. if $q \xrightarrow{a}$ then $p \xrightarrow{a}$. □

Let $\leq_{RS}$ be the union of all ready simulation preorders. Then we have the following result.

Lemma 3.5 $p \leq_d q$ *implies* $p \leq_{RS} q$. *But not vice versa.* □

We have also the following relation which has more discriminative power than $\leq_d$.

Definition 3.6 A binary relation $\sqsubseteq_d$ over processes is a maximum relation which satisfies the followings. 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 exist $p'_1, \ldots, p'_n$ such that $p \xrightarrow{a} p'_i$ for each p'_i, and $p'_1 + \cdots + p'_n \sqsubseteq_d q'$, where $n \geq 1$. □

Lemma 3.7 $p \sqsubseteq_d q$ *implies* $p \leq_d q$. *But not vice versa.* □

After all, we have the following theorem. See Fig. 1. In the figure, the arrows indicate proper inclusion relation between preorders, i.e. $R \rightarrow R'$ means R is properly included by R' (R' has more discriminative power than R). About traces [19], failures [5] and simulation preorders [19] and thier relationship, see [10] for more detail.

Theorem 3.8 $\leq_{RS} \supsetneq \leq_d \supsetneq \sqsubseteq_d \supsetneq \sim$ □

4 Synthesis algorithm

This section describes an inductive synthesis algorithm for recursive processes. Formulae in μ-calculus are regarded as specific properties of the intended process.

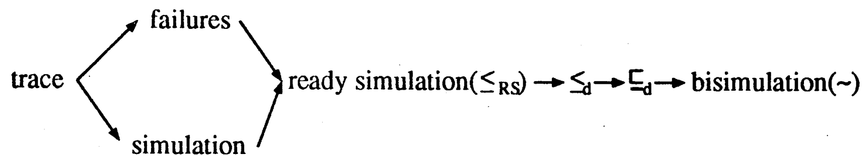

図 1: The relationship of preorders.

4.1 Enumeration of facts

An algorithm we will propose now is an inductive one. It generates a process which satisfies given facts, the properties of the intended target process, represented as formulae in μ-calculus. Thus, the 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 manner.

Definition 4.1 Let U be a set of pairs of formulae $f \in \mathcal{L}$ and a sign $+$ (or $-$), i.e. $\langle f, + \rangle$ (or $\langle f, - \rangle$) such that either $\langle f, + \rangle$ or $\langle f, - \rangle$ always belongs to U for every formula $f \in \mathcal{L}$. $S = \{f \mid \langle f, + \rangle \in U\} \cup \{\neg f \mid \langle f, - \rangle \in U\}$ is an *enumeration of facts* if S is consistent in the deductive system $\mathrm{STL}(\mathcal{X}, \mathcal{A})$[1] [11]. An element of S is called a *fact*. □

If we used p_o, the enumeration of facts might be defined as follows:

$$S = \{f \in \mathcal{L} \mid p_o \models f\}$$

Unfortunately, this definition of an enumeration of facts is impossible. Since p_o is not known a priori, we must consider S from U in Definition 4.1 as an enumeration of facts.

4.2 Synthesis algorithm

As we mentioned in section 3, our algorithm restricts input formulae to elements of the set $\mathcal{L}_d$ in order to avoid non-determinacy arising from both $\vee$ and μ operators. To take account of this restriction, the definition of an enumeration of facts must be modified, i.e. define an enumeration of facts as it Definition 4.1 and remove formulae which do not belong to $\mathcal{L}_d$ from an enumeration of facts. Note that a formula with ν operator also has non-determinacy, i.e. how many times loops of process branches unfold.

Given an enumeration of facts, the algorithm synthesizes a process satisfying those facts. Recall that a process can be represented as a term p with a set $\{c_1 \stackrel{\mathrm{def}}{=} p_1, \ldots, c_n \stackrel{\mathrm{def}}{=} p_n\}$ of defining equations. In the algorithm, a process is represented as a set of process definitions. Each process definition $\mathbf{rec}\, c.p$ is associated with a set C of formulae, denoted as $c{:}C$, which must be satisfied by the corresponding process constant c. C can be omitted when it is not important. To describe the algorithm, we adopt a language like Prolog [6], where I/O predicates

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

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. Thus, a set $\{c_0 \stackrel{\text{def}}{=} p_0, \ldots, c_n \stackrel{\text{def}}{=} p_n\}$ of process definitions determines the process c_0 with its set of process definitions.

For a fact of the form $\nu x.f(x)$, it is important to take an identification of formulae $\nu x.f(x)$ (or bound variables x) with process constants c. If c_i corresponds to x, i.e. the formula $\nu x.f(x)$, the variable x is renamed by x_i. Since x is a bound variable, the meaning of the formula is not changed. We assume further that we can recall the original formula $\nu x_i.f(x_i)$ from x_i. Also we adopt the following abbreviations:

$\bigwedge\{f_1, \ldots, f_n\} = f_1 \wedge \ldots \wedge f_n$ where $\wedge\emptyset \stackrel{\text{def}}{=} \mathbf{tt}$.

$S[c_1{:}C_1 \stackrel{\text{def}}{=} p_1, \cdots, c_k{:}C_k \stackrel{\text{def}}{=} p_k]$: The resulting set of process definitions S where the process definitions of $c_1, \cdots, c_k$ in S are replaced by $c_1{:}C_1 \stackrel{\text{def}}{=} p_1, \cdots, c_k{:}C_k \stackrel{\text{def}}{=} p_k$, respectively, or $c_i{:}C_i \stackrel{\text{def}}{=} p_i$ is added to S if $c_i{:}C_i \stackrel{\text{def}}{=} p_i \notin S$.

$S\{x/y\}$: The resulting S where a free variable y is substituted for x in S.

Now, we are in a position to state the synthesis algorithm. In order to help the understanding of the algorithm, simple comments are attached directly to the corresponding predicates which begin with the mark "%". The detail explanation of the algorithm will be stated after the completion of the algorithm.

Algorithm 4.2 [Synthesis algorithm]

Input: Enumeration of facts $f_1, f_2, \cdots$. It is an enumeration of formulae be satisfied by the intended target process. The order of them is arbitrary.

Output: Sequence of inferred processes $p_1, p_2, \cdots$. Each p_k satisfies the whole input formulae f_1 to f_k.

mpstart :- *mp*($\{c_0{:}\{\mathbf{tt}\} \stackrel{\text{def}}{=} \mathbf{0}\}$). % The initial process is **0**.

mp(S) :- % S is a set of process definitions.
    read-formula(f), % Input a formula.
    makeproc($c_0, S, f,$ X), % Modify the current process according to the new fact f,
        % the result is set to X.
    write-process(X), % Output the result.
    mp(X). % Continue the synthesis process for the next fact.

% program clauses of *makeproc*($c, S, f,$ X) [2]

[2] In the following procedures (clauses), we use the several meta variables and Prolog-like variables whose intended meaning are explained below:

% **tt** .... (a)
$makeproc(c_i, S, \mathbf{tt}, S)$.

% x_j : a bound variable corresponding to the formula $\nu x_j.f(x_j)$ .... (b)
$makeproc(c_i, S, x_i, S)$.
$makeproc(c_i, S, x_j, \mathtt{X})$:- % Where $i \neq j$.
    $S' \leftarrow (S[c_j{:}C_j \stackrel{\text{def}}{=} p_i + p_j] - \{c_i{:}C_i \stackrel{\text{def}}{=} p_i\})\{x_j/x_i\}\{c_j/c_i\}$,
    $makeproc(c_j, S', \wedge C_i, \mathtt{X})$. .... (b*)
$makeproc(c_i, S, x_j, \mathtt{X})$:-
    is-remake,
    $makeproc(c_i, S, f(x_j), \mathtt{X})$. .... (b**)

% $\langle a \rangle f$ .... (c)
$makeproc(c_i, S, \langle a \rangle f, \mathtt{X})$:-
    $exists(c_j, c_i, S, f, \mathtt{X})$. % $\exists c_j$ such that $c_i \xrightarrow{a} c_j$ and $makeproc(c_j, S, f, \mathtt{X})$.
$makeproc(c_i, S, \langle a \rangle f, \mathtt{X})$:-
    get-new-process-constant(c_j),
    $makeproc(c_j, S[c_i{:}C_i \stackrel{\text{def}}{=} p_i + a.c_j, c_j{:}\{\mathtt{tt}\} \stackrel{\text{def}}{=} \mathbf{0}], f \wedge (\wedge\{f_k \mid [a]f_k \in C_i\}), \mathtt{X})$. ...... (c*)

% $[a]f$ .... (d)
$makeproc(c_i, S, [a]f, S)$:-
    $is\text{-}valid(\wedge C_i \supset [a]f)$. % $\models \wedge C_i \supset [a]f$
$makeproc(c_i, S, [a]f, S[c_i{:}(C_i \cup \{[a]f\}) \stackrel{\text{def}}{=} p_i])$:-
    $not\text{-}transit(c_i, a)$. % $c_i \stackrel{a}{\not\rightarrow}$.
$makeproc(c_i, S, [a]f, \mathtt{X})$:-
    $forall(c_j, c_i, S[c_i{:}C_i \cup \{[a]f\} \stackrel{\text{def}}{=} p_i], f, \mathtt{X})$.
    % $\forall c_j.c_i \xrightarrow{a} c_j, makeproc(c_j, S[c_i{:}C_i \cup \{[a]f\} \stackrel{\text{def}}{=} p_i], f, \mathtt{X})$.

% $f_1 \wedge f_2$ .... (e)
$makeproc(c_i, S, f_1 \wedge f_2, \mathtt{X})$:-
    $makeproc(c_i, S, f_1, \mathtt{Y})$,
    $makeproc(c_i, \mathtt{Y}, f_2, \mathtt{X})$.

% $\nu x.f(x)$ .... (f)
$makeproc(c_i, S, \nu x.f(x), \mathtt{X})$:-
    $makeproc(c_i, S, f(x_i), \mathtt{X})$. □

---

c: the current process constant (meta variable)
S: the current set of process definitions (meta variable)
f: the current formula to be satisfied by c (meta variable)
$\mathtt{X}$: the inferred process — a set of process definitions (Prolog-like variable)

Now, we explain the intuitive function of the clauses.

(a): If the current formula is **tt**, simply return S since **tt** is satisfied by any processes. Note that there are no clauses for the formula **ff**. Since **ff** indicates that the input formulae are inconsistent, therefore it needs backtracking for this case. By means of backtracking mechanism, the intended process will be eventually generated.

(b): If the current formula is x_i, return S since there already exists a recursive loop. If the current formula is x_j (a process variable) which does not correspond to the current process constant c_i, c_i with S needs modifications, since c_i must satisfy the formula x_j (i.e. $\nu x_j.f(x_j)$) which must be satisfied by c_j. Therefore, in the clause identify c_i and c_j at first, then modify c_j again to satisfy every condition in C_i (See Fig. 2). The third clause in the case of logical variable will be invoked when identification of c_i and c_j makes contradiction. They may arise from the direct recursive loop, i.e. wrong connection of c_i and c_j. However, this is not always the cases. Therefore, we need a controlling predicate. The predicate, *is-remake*, judges whether or not unfolding of $\nu x_j.f(x_j)$ is necessary in such a way that *is-remake* succeeds iff the unfolding of the formula $\nu x_j.f(x_j)$ is necessary. Its intended function (meaning) will be explained after the explanation of the algorithm.

(c): If the current formula is of the form $\langle a\rangle f$, the clause generally makes a branch labeled with a and constructs a new process satisfying f as an a-successor of c_i. However, if there already exists an a-successor c_j of c_i such that c_j can be modified to satisfy f, then neither new constants nor new processes are created. Otherwise, the clause creates a new branch followed by a new process by getting a fresh process constant c_j.

(d): For the current formula $[a]f$ every a-successor must be checked and modified to satisfy the subformula f. This is done by the last clause of this case. This check can be easily verified if the condition $\models \wedge C_i \supset [a]f$ holds. This is why we attach the condition to each process definition, i.e. a process constant. If c_i cannot perform the action a, it is sufficient to add $[a]f$ to C_i.

(e): If the current formula is a conjunction $f_1 \wedge f_2$, apply f_1 and f_2 in this order.

(f): For the recursive formula $\nu x.f(x)$, rename the bound variable x into x_i to adjust it to c_i,

Whenever the formula $\nu x.f(x)$ is applied to a process constant, the procedure *makeproc* tries to make a loop at the nearest place from the applied process constant. Especial, for the first time, it tries to make direct loop to the applied process constant. However, making a loop at the nearest place sometimes conflicts with the facts. Such situations are illustrated in the Fig. 3 and 4. In Fig. 3, the direct loop created at the first stage by the formula $\nu x_0.\langle a\rangle\langle b\rangle x_0$ conflicts with the third fact $[a][b][b]$**ff**. Thus, the direct loop must be unfolded to avoid the conflict. In Fig. 4, the process constant c_1 with the condition $[b]x_0$ at the first stage has potential power to make a loop, possibly actuated by some facts, e.g. $[a]\langle b\rangle\langle c\rangle$**tt** in this example. Then, the created direct loop conflicts with the third fact $[c]$**ff**.

To avoid the unnecessary unfoldings of loops, the procedure, *is-remake*, checks whether or

not the current process is in the situations in Fig. 3 or 4 whenever invoked, that is the current process does not satisfy the given facts. Then, *is-remake* forces backtracking if the current process is not in the situations in Fig. 3 or 4. Otherwise, the procedure succeeds, i.e. direct loops are unfolded once stated as above. In the case illustrated in Fig. 3, *is-remake* traces the path which is passed by a formula occurring inconsistent, and backtracking is allowed if the path has one or more loops and does not end on these loops. In the case of Fig. 4, *is-remake* traces the path in the same way as the previous case, and backtracking is allowed if the path is at the beginning of a loop, but does not go inside the loop, instead takes the different path. In each case, the information about which formula made each branch is needed.

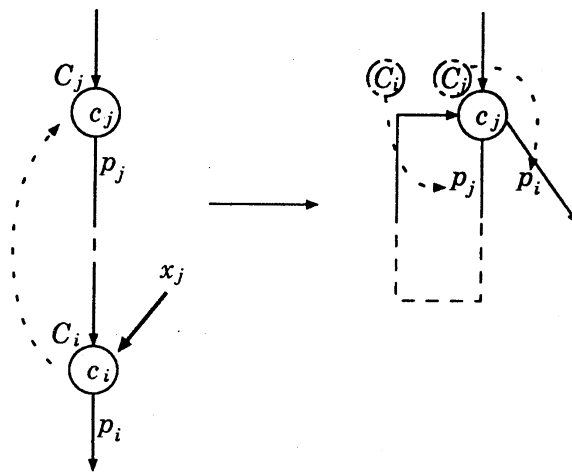

図 2: The function of the clause (b*).

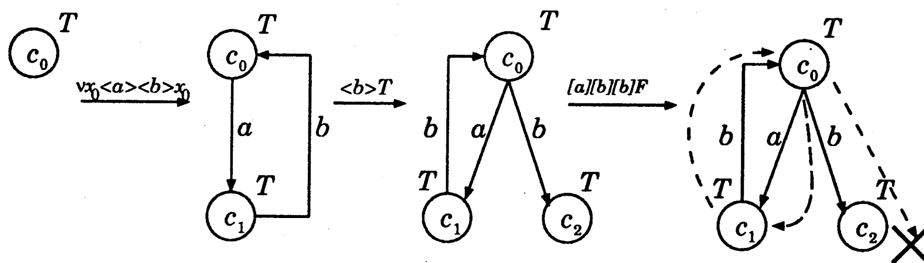

図 3: Some action sequences are possible since a loop is constructed.

4.3 Results of the Algorithm

Lemma 4.3 *Let S be a set of process definitions applied to the predicate makeproc and $c{:}C \stackrel{\text{def}}{=} p \in S$. If f is a formula such that $c \models f$ then makeproc$(c, S, f, \mathtt{X})$ terminates with $\mathtt{X} \equiv S$ except that several formulae may be added to some sets of formulae labeled at process constants. Also if f has been already applied to makeproc, $S \equiv \mathtt{X}$.* □

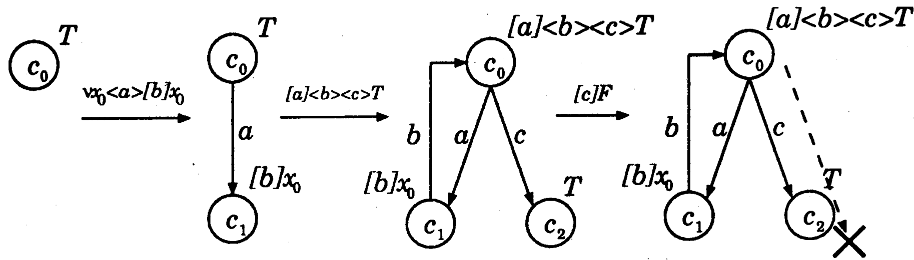

図 4: Some action sequences are possible since a branch modifies a loop.

Theorem 4.4 *Assume that there exists a process p_n satisfying initial segments $f_1, \cdots, f_n$ of an enumeration of facts, where $n \geq 1$. Assume Algorithm 4.2 outputs a set of process definitions S_{n-1} for the $n-1$ facts, $f_1, \cdots, f_{n-1}$ also. For the n-th fact, f_n, we have the followings:*

1. *The algorithm 4.2 terminates and returns an output, which is a set of process definitions S_n with the process constant c_0 (the initial state of S_n).*
2. *c_0 with S_n satisfies f_n.*
3. *c_0 with S_n satisfies $f_1, \cdots, f_{n-1}$.*

proof 1. When the predicate *makeproc* calls itself recursively, let f be a given formula to it, and g be a formula to call itself. Then, the size of g — the number of operators constructing the formula — can be greater than the size of f, only in the clauses (b*), (b**) and (c*) in the Algorithm 4.2. Without using the clauses (b*), (b**) and (c*), the algorithm terminates. Therefore, it is sufficient to consider them only. Instinctively, the non-termination of the algorithm means the following cases.

(i) Application of a set of formulae continues infinitely many times as if it is a chain reaction. This situation corresponds to (c*). In (c*), the algorithm adds a new branch from the current process definition. A process after the branch must satisfy every formula in the labeled set at the process definition. However, another process definition may be added as a new branch by the process, and the algorithm may arrive at (c*) again. If the above situation continues, the predicate *makeproc* does not terminate.

(ii) Process reconstruction continues infinitely many times. This case corresponds to (b*) and (b**). The clause (b*) makes a loop as short as possible to satisfy a formula with a ν operator. However, in the cases of Fig. 3 and 4, the loop must be unfolded once by backtracking at (b**). Unfortunately unfolded loop may also be in the situation of Fig. 3 or 4 and arrive at (b**) again. Repeating the above, process reconstruction continues infinitely many times. Note that in this case, *makeproc* synthesizes one or more branches with infinite depth.

Suppose the given enumeration of facts has no ν operator. Then the case (ii) dose not arise, since both of definitions (b*) and (b**) are not used. We consider only the case (i). However, since each formula has no process variable, the size of a formula used for recursive call in (c*) is less than one given to it. Consequently, the algorithm terminates.

Next, we assume that there exists a formula with ν in enumeration of facts. We have the following cases.

- Only the case (i) is arisen.
  From the previous lemma, *makeproc* neither makes new branches nor process definitions from a formula which are already given. Since the size of each element of a set of formulae for each process definition is finite, the set of formulae can be applied within only finite range of the process. Therefore application of formulae is saturated in finite time, and then *makeproc* terminates.

- Only the case (ii) is arisen.
  The definition (c*) unfolds a loop once. To arise the case (ii), there must exists some given formulae which negate the loop infinitely many times and satisfy the condition of *is-remake*. A formula without ν operator cannot negate it infinitely. Even if there is a formula with ν operator, the part which negates the loop, i.e. the formula **ff**, must occur periodically. Therefore when a loop is unfolded finitely many times, the formula not only negates points inside a loop but also negates points outside the loop. Hence *is-remake* fails and *makeproc* terminates.

- Both the cases (i) and (ii) are arisen.
  We can assume that both the cases (i) and (ii) arise alternatively. Then, firstly, some formulae negate the point of a loop, and thus the loop is unwound. Next, a set of formulae, which is labeled at a process definition, are applied to the unwound loop or certain branches. After all, the transmition of formulae arrives at the point of the loop, and these processes are repeated. To negate infinitely, there must exist at most one formula which have one or more ν operators and negates the loop. This argument similar the previous one, and thus the algorithm terminates.

2. By the following procedure, *is-satisfied*(f_n, i), we make sure that c_i satisfy f_n.

procedure *is-satisfied*(f, i);
case f **of**

tt : Obviously $c_i \models f$.

ff : It means that there does not exist a process which satisfies input formulae. Thus, this case does not arise.

$\langle a \rangle f'$: From the algorithm, there exists c_j such that *makeproc* selects it when the formula is applied. This c_j also satisfies $c_i \xrightarrow{a} c_j$. Then make sure *is-satisfied*(f', j) is satisfied.

$[a]f'$: If $c_i \not\xrightarrow{a}$, then obviously $c_i \models f$. In the case that $c_i \xrightarrow{a} c_j$, make sure *is-satisfied*(f', j) is satisfied for any c_j.

$g \wedge h$: Make sure *is-satisfied*(g, i) and *is-satisfied*(h, i) are satisfied.

$\nu x.g(x)$: Make sure *is-satisfied*$(g(x_i), i)$ is satisfied.

x_i : Let its original formula be $\nu x_j.g(x_j)$. Suppose $x_j = x_i$. Since the procedure comes here, $c_j \models g^n(\mathbf{tt})$ where reconstructing loops for the original formula arise in $n-1$ times and $n \geq 1$. As $g^n(x)$ is monotonicity for n, repeating this procedure leads to $c_j \models \bigwedge_{n \geq 0} g^n(\mathbf{tt})$. Therefore we show that $c_j \models \nu x_j.g(x_j)$. Since length of each input formula is finite and the synthesis algorithm terminates, the procedure terminates in finite steps.

3. Same as 2. □

The algorithm is a non terminating procedure. Therefore, we show its correctness by using the concept of convergence in the limit, which has been a key idea in inductive learning paradigm [21].

Definition 4.5 Assume an algorithm reads in an enumeration of facts, and returns 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. □

Lemma 4.6 *Assume p is an intended process, and the inferred sequence of processes by the Algorithm 4.2 converges in the limit to a process p'. Then $p \leq_d p'$.* □

We construct a formula which has sufficient information to synthesize a process. For preliminary, we need the following definition.

Definition 4.7 If a set of process definitions S satisfies the following conditions, we call S a *complete set of process definitions* of a process p:

1. S has the initial process definition $c_0 \stackrel{\text{def}}{=} p_0$ such that $p \sim c_0$.
2. Each process definition is of the form $c \stackrel{\text{def}}{=} a_1.c_1 + \cdots + a_n.c_n$, where $n \geq 0$ (when $n = 0$, $a_1.c_1 + \cdots + a_n.c_n \stackrel{\text{def}}{=} \mathbf{0}$), $a_i \in \mathcal{A}$ and each c_i is a process constant whose process definition belongs to S.
3. For any process definition $c \stackrel{\text{def}}{=} q$ in S, c_0 in the deleted set $S - \{c \stackrel{\text{def}}{=} q\}$ of process definitions is not equivalent to p, i.e. $c_0 \not\sim p$ any more.

Every guarded process p can be translated to a complete set of process definition of p by the following algorithm. We represent this set $\mathcal{P}(p)$.

Algorithm 4.8 Let p be a guarded process and S a set of process definitions associated with p. At first, get a fresh process constant c_0 which is not included in S, then add $c_0 \stackrel{\text{def}}{=} p$ to S. Finally apply the following transformation rules to S until S is not modified any more.

1. If $c \stackrel{\text{def}}{=} a.p \in S$ and p is not a process constant then get a fresh process constant c' and $S \leftarrow S[c \stackrel{\text{def}}{=} a.c',\ c' \stackrel{\text{def}}{=} p]$.
2. If $c \stackrel{\text{def}}{=} p + \mathbf{0} \in S$ then $S \leftarrow S[c \stackrel{\text{def}}{=} p]$.
3. If $c \stackrel{\text{def}}{=} p + p + r \in S$ then $S \leftarrow S[c \stackrel{\text{def}}{=} p + r]$.
4. If $c \stackrel{\text{def}}{=} p + a.q \in S$ and q is not a process constant then get a fresh process constant c' and $S \leftarrow S[c \stackrel{\text{def}}{=} p + a.c',\ c' \stackrel{\text{def}}{=} q]$.
5. If $c \stackrel{\text{def}}{=} p + c'$ and $c' \stackrel{\text{def}}{=} q \in S$ then $S \leftarrow S[c \stackrel{\text{def}}{=} p + q]$. Note that $c \neq c'$ since p is guarded.
6. If $c \stackrel{\text{def}}{=} c'$ and c' is a process constant then $S \leftarrow (S - \{c \stackrel{\text{def}}{=} c'\})\{c/c'\}$.
7. If $c \stackrel{\text{def}}{=} q \in S$ where $c \neq c_0$, and c dose not occur in any other process definition then $S \leftarrow S - \{c \stackrel{\text{def}}{=} q\}$. □

For example, if $S = \{c_0 \stackrel{\text{def}}{=} a.c_1,\ c_1 \stackrel{\text{def}}{=} b.(c_0 + c_1)\}$, then $\mathcal{P}(c_0) = \{c_0 \stackrel{\text{def}}{=} a.c_1,\ c_1 \stackrel{\text{def}}{=} b.c_2,\ c_2 \stackrel{\text{def}}{=} a.c_1 + b.c_2\}$. Observe that c_0 in S and c_0 in $\mathcal{P}(c_0)$ are strongly equivalent.

Lemma 4.9 *For any guarded process p, Algorithm 4.8 terminates and $P(p)$ is a complete set of process definitions of p.* □

Now, we can construct a sound and complete formula $\mathcal{F}(p)$ for a process p w.r.t. $\leq_d$ in our restricted μ-calculus. For processes p and q, *soundness* means that $p \leq_d q$ implies $q \models \mathcal{F}(p)$ and *completeness* means that $q \models \mathcal{F}(p)$ implies $p \leq_d q$.

In the full μ-calculus, Stirling [24] gave the formation of the sound and complete formula $\mathcal{F}'(p)$ for p w.r.t. the strong equivalence $\sim$ in the sense that $p \sim q$ iff $q \models \mathcal{F}'(p)$ for all processes p and q. As an example, consider the process $p = a.p_1 + a.p_2$, then we have $\mathcal{F}'(p) = \langle a\rangle\mathcal{F}'(p_1) \wedge \langle a\rangle\mathcal{F}'(p_2) \wedge [a](\mathcal{F}'(p_1) \vee \mathcal{F}'(p_2))$. Unfortunately our restricted calculus has no $\vee$ operators. We cannot use $\vee$ operators to define $\mathcal{F}(p)$ for p. So it seems that $\mathcal{F}(p) = \langle a\rangle\mathcal{F}'(p_1) \wedge \langle a\rangle\mathcal{F}'(p_2)$. But this is not sufficient since p_1 and p_2 may have common properties, i.e. $\mathcal{L}_d(p_1) \cap \mathcal{L}_d(p_2) \neq \emptyset$. Thus $\mathcal{F}(p)$ should be $\langle a\rangle\mathcal{F}'(p_1) \wedge \langle a\rangle\mathcal{F}'(p_2) \wedge [a]\bigwedge(\mathcal{L}_d(p_1) \cap \mathcal{L}_d(p_2))$ though it is not inductive definition. The function $\mathcal{F}(p)$ with the auxiliary function $\mathcal{F}_S(C)[\mathcal{C}]$ in the next definition gives the inductive formation of a sound and complete formula for a process p in $\mathcal{L}_d$. Instinctively, $\mathcal{F}_S(C)[\mathcal{C}]$ is a formula which is logically equivalent to $\bigwedge(\bigcap_{c \in C} \mathcal{L}_d(c))$, where C is a set of process constants and $\mathcal{C}$ is a family of sets of process constants. $\mathcal{C}$ is used technically to make recursive loops on formulae.

Definition 4.10 Let S be a complete set of process definitions, $C_0 = \{c \mid c \stackrel{\text{def}}{=} p \in S\}$ and $\mathcal{C}$ be a set of subsets of C_0. For $C \subseteq C_0$, a formula $\mathcal{F}_S(C)[\mathcal{C}]$ is defined in the following mutual recursive equations:

$$\mathcal{G}_S(C,a)[\mathcal{C}] = \begin{cases} [a]\mathbf{ff} \ \text{ if } c \not\xrightarrow{a} \text{ for any } c \in C, \\ [a](\mathcal{F}_S(\bigcup_{c \in C} s(c,a))[\mathcal{C}]) \ \text{ if there exist } c, c' \in C \text{ such that } c \xrightarrow{a} \text{ and } c' \not\xrightarrow{a}, \\ (\bigwedge_{C' \in comb(\{s(c,a) \mid c \in C\})} \langle a\rangle(\mathcal{F}_S(C')[\mathcal{C}])) \wedge [a](\mathcal{F}_S(\bigcup_{c \in C} s(c,a))[\mathcal{C}]) \ \text{ otherwise.} \end{cases}$$

$$\mathcal{F}_S(C)[\mathcal{C}] \;=\; \begin{cases} x_C \text{ if } C \in \mathcal{C}, \\ \nu x_C.\bigwedge_{a\in\mathcal{A}}(\mathcal{G}_S(C,a)[\mathcal{C}\cup\{C\}]) \text{ otherwise.} \end{cases}$$

where $s(c,a) = \{c' \in C \mid c \xrightarrow{a} c'\}$ and $comb(\{C_1,\ldots,C_n\}) = \{\{c_1,\ldots,c_n\} \mid c_1 \in C_1,\ldots,c_n \in C_n\}$ for $n \geq 0$. For a guarded process p, we define $\mathcal{F}(p) \stackrel{\text{def}}{=} \mathcal{F}_{\mathcal{P}(p)}(\{c_0\})[\emptyset]$ where c_0 is the initial process constant of $\mathcal{P}(p)$. □

Let consider $S = \{c_0 \stackrel{\text{def}}{=} a.c_1,\ c_1 \stackrel{\text{def}}{=} a.c_2,\ c_2 \stackrel{\text{def}}{=} a.c_0 + a.c_1 + b.c_1\})$ and $\mathcal{A} = \{a,b\}$, then $\mathcal{F}(c_0)$ is given by the following equations, where a family of sets of process constants is omitted.

$$\begin{aligned}
\mathcal{F}(c_0) = \mathcal{F}_S(\{c_0\}) &= \nu x_{\{c_0\}}.\langle a\rangle\mathcal{F}_S(\{c_1\}) \wedge [a]\mathcal{F}_S(\{c_1\}) \wedge [b]\mathbf{ff} \\
\mathcal{F}_S(\{c_1\}) &= \nu x_{\{c_1\}}.\langle a\rangle\mathcal{F}_S(\{c_2\}) \wedge [a]\mathcal{F}_S(\{c_2\}) \wedge [b]\mathbf{ff} \\
\mathcal{F}_S(\{c_2\}) &= \nu x_{\{c_2\}}.\langle a\rangle x_{\{c_0\}} \wedge \langle a\rangle x_{\{c_1\}} \wedge [a]\mathcal{F}_S(\{c_0,c_1\}) \wedge \langle b\rangle x_{\{c_1\}} \wedge [b]x_{\{c_1\}} \\
\mathcal{F}_S(\{c_0,c_1\}) &= \nu x_{\{c_0,c_1\}}.\langle a\rangle\mathcal{F}_S(\{c_1,c_2\}) \wedge [a]\mathcal{F}_S(\{c_1,c_2\}) \wedge [b]\mathbf{ff} \\
\mathcal{F}_S(\{c_1,c_2\}) &= \nu x_{\{c_1,c_2\}}.\langle a\rangle\mathcal{F}_S(\{c_0,c_2\}) \wedge \langle a\rangle x_{\{c_1,c_2\}} \wedge [a]\mathcal{F}_S(\{c_0,c_1,c_2\}) \wedge [b]x_{\{c_1\}} \\
\mathcal{F}_S(\{c_0,c_2\}) &= \nu x_{\{c_0,c_2\}}.\langle a\rangle x_{\{c_0,c_1\}} \wedge \langle a\rangle x_{\{c_1\}} \wedge [a]x_{\{c_0,c_1\}} \wedge [b]x_{\{c_1\}} \\
\mathcal{F}_S(\{c_0,c_1,c_2\}) &= \nu x_{\{c_0,c_1,c_2\}}.\langle a\rangle x_{\{c_1,c_2\}} \wedge \langle a\rangle x_{\{c_0,c_1,c_2\}} \wedge [a]x_{\{c_0,c_1,c_2\}} \wedge [b]x_{\{c_1\}}
\end{aligned}$$

We show $\mathcal{F}(p)$ is a sound and complete formula of p in the following lemmas and proposition.

Lemma 4.11 *Let S be a complete set of process definitions, $C_0 = \{c \mid c \stackrel{\text{def}}{=} p \in S\}$ and $\mathcal{C}$ be a set of subsets of C_0. For any $C \subseteq C_0$ and for any $c \in C$, $c \models \mathcal{F}_S(C)[\mathcal{C}]$.* □

Definition 4.12 For two processes p and q, let C_p and C_q be the sets of all process constants of $\mathcal{P}(p)$ and $\mathcal{P}(q)$ respectively, and $c_0 \in C_p$ and $c_0' \in C_q$ be initial process constants of C_p and C_q respectively. The *corresponded relation* over subsets of C_p and C_q is a binary relation $=_{p,q} \subseteq 2^{C_p} \times 2^{C_q}$ defined in the following:

1. $\{c_0\} =_{p,q} \{c_0'\}$.
2. For $C = \{c_1,\ldots,c_n\} \subseteq C_p$ and $C' = \{c_1',\ldots,c_m'\} \subseteq C_q$, suppose $C =_{p,q} C'$. Then for any action a, $\{c_i'' \mid c_i \xrightarrow{a} c_i'', 1 \leq i \leq n\} =_{p,q} \{c_j''' \mid c_j' \xrightarrow{a} c_j''', 1 \leq j \leq m\}$.
3. For $C \subseteq C_p$ and $C' \subseteq C_q$, if $C =_{p,q} C'$ then $\{c'' \mid c \xrightarrow{a} c'', c \in C\} =_{p,q} \{c''' \mid c' \xrightarrow{a} c''', c' \in C'\}$ for any action a. □

Lemma 4.13 *For two processes p and q, let C_p and C_q be the set of all process constants of $\mathcal{P}(p)$ and $\mathcal{P}(q)$, and $c_0 \in C_p$ and $c_0' \in C_q$ be initial process constants of C_p and C_q respectively. Let $C \subseteq C_p$ and $C' \subseteq C_q$ and suppose $C =_{p,q} C'$ and for any $c' \in C'$ for some $\mathcal{C} \subseteq 2^{C_p}$, $c' \models \mathcal{F}_{\mathcal{P}(p)}(C)[\mathcal{C}]$. If $c \models f$ for $c \in C$, then $c' \models f$ for $c' \in C'$.*

proof The proof is by structural induction on f. First, we prove the lemma for the case that f has no ν operator. It is sufficient to consider only the cases $f = \langle a \rangle f'$ and $f = [a]f'$. Another cases are immediate. In the following, we abbreviate $\mathcal{P}(p)$ as S for easiness description.

When $f = \langle a \rangle f'$, let C be $\{c_i \mid 1 \le i \le n\}$ and C' be $\{c'_j \mid 1 \le j \le m\}$. Note that $c_i \xrightarrow{a}$ for $c_i \in C$ since $c_i \models f$. So $\mathcal{G}_S(C, a) = (\bigwedge_{C'' \in comb(\{s(c,a) | c \in C\})} \langle a \rangle (\mathcal{F}_S(C'')[\mathcal{C} \cup \{C\}])) \wedge [a](\mathcal{F}_S(\bigcup_{c \in C} s(c, a))[\mathcal{C} \cup \{C\}])$. From assumption, we can define $C'' = \{c''_i \mid c_i \xrightarrow{a} c''_i, c''_i \models f', 1 \le i \le n\}$. Then there exists $C''' = \{c'''_j \mid c'_j \xrightarrow{a} c'''_j, c'''_j \models \mathcal{F}_S(C'')[\mathcal{C} \cup \{C\}], 1 \le j \le m\}$. Since $C'' =_{p,q} C'''$ and by the induction hypothesis, $c'''_j \models f'$ for $c'''_j \in C'''$. Therefore $c' \models f$ for every $c' \in C'$.

When $f = [a]f'$, first we assume $c \xrightarrow{a}$ for any $c \in C$. Then let C'' be $\bigcup_{c \in C} s(c, a)$, and C''' be $\bigcup_{c' \in C'} s(c', a)$. From assumption, for any $c'' \in C''$, $c'' \models f'$. And for any $c''' \in C'''$, $c''' \models \mathcal{F}_S(C'')$. Since $C'' =_{p,q} C'''$ and by the induction hypothesis, $c''' \models f'$ for any $c''' \in C'''$. Therefore $c' \models f$ for every $c' \in C'$. Another cases are same as the above.

Finally, from Proposition 2.14, $c_q \models f$ for any formula f. □

Proposition 4.14 *1.* $p \models \mathcal{F}(p)$.

2. $p \le_d q$ *implies* $q \models \mathcal{F}(p)$.

3. $q \models \mathcal{F}(p)$ *implies* $p \le_d q$. □

The validity of Algorithm 4.2 is also shown by the following theorem.

Theorem 4.15 *Under the assumption of algorithm 4.2, if there exists a process p satisfying an enumeration of facts, the inferred sequence of processes by Algorithm 4.2 converges in the limit to a process* p' *such that* $p \le_d p'$. □

5 A Prototype for the Process Synthesis System

In this section, we introduce a prototype system SORP : Synthesizer of Recursive Processes based on the Algorithm 4.2. This system adopts a graphical user interface to display the synthesized processes (See Fig. 5). The system is implemented using SICStus Prolog and X-window system.

As an example, we input three formulae, in Fig. 3, to the prototype system. Its output in Fig. 5. The extreme left picture in Fig. 5 shows an I/O display, where we input the three formulae, i.e. $\nu x.\langle a \rangle \langle b \rangle x$, $\langle b \rangle \mathbf{tt}$, and $[a][b][b]\mathbf{ff}$, and quit the system. Note that '\$x:' in the picture means '$\nu x.$'. Three other pictures are output processes which the system synthesizes in each input step. Note that each process corresponds to the one in Fig. 3.

6 Concluding Remarks and Related Works

This paper presented the synthesis algorithm for a recursive process based on the enumeration of facts, which must be satisfied by the intended target process. Its validity was also discussed.

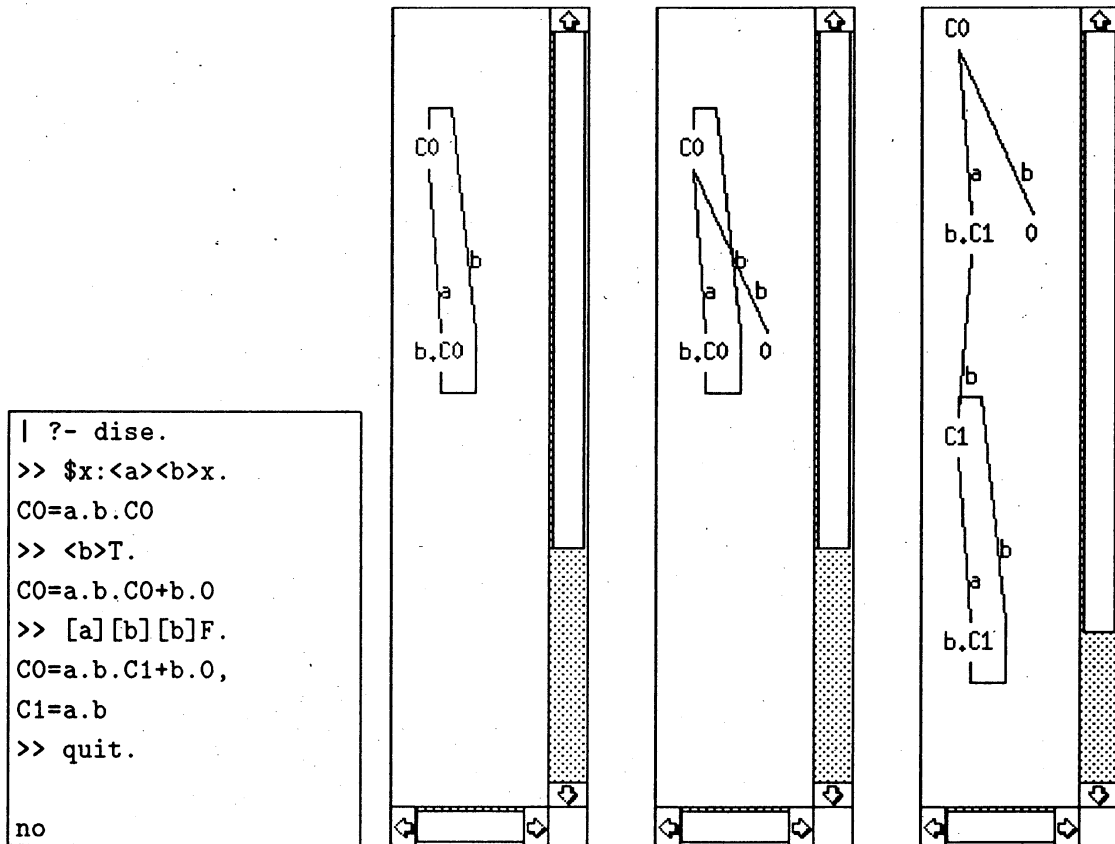

図 5: Output examples of the prototype system.

As mentioned in the introduction, little had been investigated for inductive inference of processes. However, some deductive approaches exist. These approaches find a model which satisfies a consistent formula. Kozen [17] provided an algorithm by tableau method to show consistency of a formula in μ-calculus. The algorithm builds a finite tree-like model which a consistent formula f satisfies. Since the model is a tree, not a graph, the algorithm cannot make a loop, i.e. a recursive process. Actually, he showed that the depth of the model is exponential in $|f|^3$. Streett and Emerson [27] presented a decision procedure to build an automaton model which satisfies a given formula f. The built automaton model is a finite tree with states in $O(2^{2^{|p|}})$. Similar approaches for temporal logic are in [18] and [3]. [18] presented a satisfiability algorithm to create a model satisfied by a given formula in liner time propositional temporal logic by using tablaue method, though its logic has no fixed point operators. [3] proposed νTL, which is liner time temporal logic with fixed point operators μ and ν and provided an algorithm that constructed a graph model of given formula.

Stirling's work [22] seems to be related with our work. In [22], he showed a sound and complete deductive system NL for finite processes. Using NL, we can deduce that a process does satisfy a certain formula. For example, $p \vdash \langle a \rangle f$ implies $p + q \vdash \langle a \rangle f$, and also $p \vdash [a]f$, $q \vdash [a]f$ imply $p + q \vdash [a]f$. From these rules, a formula $[a]\langle b \rangle T \wedge \langle a \rangle \langle c \rangle T$ can infer, for

example, a process $a.b.\mathbf{0}+a.(b.\mathbf{0}+c.\mathbf{0})$. In this sense, we can regard his system as a deductive system for process synthesis. Of course, NL has no recursive expressions. NL is needed to be extended to synthesize recursive processes effectively.

The difference between our approach and deductive ones is whether input formulae are fixed or not. When the number of input formulae is finite and the sequence of formulae is fixed, our algorithm gives similar results as the deductive one. In practice, however, a complete specification may not be given. After synthesizing, the user may want to input more facts and/or to add more functions to an output process. Our approach has advantages over deductive ones in such a situation.

There is a restriction on input formulae in our algorithm. The formulae must be within $\mathcal{L}_d$. However overcoming the problem leads us to a process synthesis algorithm whose output converges in the limit to a process equivalent to an intended target one.

The time or space complexity of the algorithm is not discussed and is left for a future study.

参考文献

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

[2] Bergstra, J.A. and J.W. Klop: "Process Algebra for Synchronous Communication", Info. and Cont., **60**, pp109–137(1984).

[3] Banieqbal, B. and H. Barringer: "Temporal Logic with Fixed Points", Lecture Notes in Comput. Sci. **398**, pp62–74, Springer-Verlag(1989).

[4] Brinksma, E.: "A Tutorial on LOTOS", Proc. IFIP Workshop on Protocol Specification, Testing and Verification V, North-Holland, pp73–84(1986).

[5] Brookes, S.D., C.A.R Hoare and A.W. Roscoe: "A Theory of Communicating Sequential Processes", J. ACM., **31**, 3, pp.560–599(1984).

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

[7] Emerson, E.A.: "Temporal and Modal Logic", Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V., pp.995–1072(1990).

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

[9] Gotzhein, R.: "Specifying Communication Services with Temporal Logic", Protocol Specification, Testing and Verification, XL, pp.295–309(1990).

[10] van Glabbeek, R.J.: "The Linear Time – Branching Time Spectrum", Lecture Notes in Comput. Sci. **458**, Springer-Verlag(1990).

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

[12] Hennessy, M. and R. Milner: "Algebraic Laws for Nondeterminism and Concurrency", J. ACM., **32**, 1, pp.137–161(1985).

[13] Hennessy, M.: "Algebraic Theory of Processes", The MIT Press(1988).

[14] Hindley, J.R. and J.P. Seldin: "Introduction to Combinators and λ-Calculus", London Mathematical Society Student Texts 1, Cambridge Univ. Press(1986).

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

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

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

[18] Manna, Z. an P. Wolper: "Synthesis of Communicating Processes from Temporal Logic Specifications", ACM Trans. on Programming Languages and Systems, **6-**, 1, pp68–93(1984).

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

[20] Park, D.: "Concurrency and automata on infinite sequences", Lecture Notes in Comput. Sci. **104**, pp.167–183, Springer-Verlag(1981).

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

[22] Stirling, C.: "A Proof-Theoretic Characterization of Observational Equivalence", Theoretical Computer Science, **39**, pp.27–45(1985).

[23] Stirling, C.: "A Modal Characterization of Observational Conguruence on Finite Terms of CCS", Info. and Comput., **68**, pp.125–145(1986).

[24] Stirling, C.: "Modal Logics For Communicating Systems", Theoretical Computer Science, **49**, pp.311–347(1987).

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

[26] Stirling, C.: "Modal and Temporal Logics", Handbook of Logic in Computer Science volume 2, Oxford Science Publications, pp.477–563(1992).

[27] Streett, R.S. and E.A. Emerson: "An Automata Theoretic Decision Procedure for the Propositional Mu-Calculus", Info. and Comput. **81**, Academic Press, pp249–264(1989).