数理解析研究所講究録
906 巻 1995 年 146-153

Extension of Synthesis Algorithm of Recursive Processes to μ-calculus ★

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

Research Institute of Electrical Communication (電気通信研究所) / *Graduate School of Information Science* (情報科学研究科), *Tohoku University* (東北大学).
e-mail : {*kimura,togashi,norio*}*@shiratori.riec.tohoku.ac.jp*

Abstract

In our previous work, we proposed an inductive synthesis algorithm for recursive processes by a subset of μ-calculus. This paper presents an extension of the privious algorithm to a wide class of μ-calculus.

Keywords: Process Synthesis, Inductive Inference, Algebraic Process, CCS, μ-calculus

1 Introduction

This paper proposes an extended inductive synthesis algorithm for recursive processes which is a proper extension of the previous algorithm [2, 3]. To synthesize a process, formulae of μ-calculus, which must be satisfied by the target process, are given to the algorithm one by one since such formulae exist infinitely many in general. The correctness of the algorithm can be stated that the output sequence of processes by the algorithm converges to a process, which is strongly equivalent to the intended one in the limit.

Let $\mathcal{A}$ be an *alphabet*, a finite set of *actions*. Let $\mathcal{C}$ be a denumerable set of *process constants*. *Recursive terms* are defined by the following BNF:

$$p ::= \mathbf{0} \mid a.p \mid p + p \mid c$$

where $c \in \mathcal{C}$ and the meaning of c is defined by a *defining equation* $c \stackrel{\text{def}}{=} p$. A process c with the equation $c \stackrel{\text{def}}{=} p$ is abbreviated as $\mathbf{rec}\, c.p$. The notions of *free*, *bound*, *scope*, *open* and *closed* are defined in the same way as in λ-calculus. Closed terms are called (*recursive*) *processes.* When every free occurrence of c is within some subterm $a.q$ of p, c is called *guarded* in p. When every constant in p is guarded, p is called *guarded.*

Let $\mathcal{P}$ denote the set of all processes. Semantics of a recursive term is given by *a labeled transition relation* defined as $\rightarrow \subset \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. For $(p, a, q) \in \rightarrow$, we normally write $p \xrightarrow{a} q$. We use the usual abbreviations as $p \xrightarrow{a}$ for $\exists q \in \mathcal{P}$ such that $p \xrightarrow{a} q$ and $p \not\xrightarrow{a}$ for $\neg \exists q \in \mathcal{P}$ such that $p \xrightarrow{a} q$.

★A part of this study is supported by Grants from the Asahi Glass Foundation and Research Funds from Japanese Ministry of Education.

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

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

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

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

A relation R on $\mathcal{P}$ is a *strong bisimulation* if $(p,q) \in R$ implies, for all $a \in \mathcal{A}$:

(i) whenever $p \xrightarrow{a} p'$, then there exists q' such that $q \xrightarrow{a} q'$ and $(p',q') \in R$,
(ii) 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* (written by $p \sim q$) iff $(p,q) \in R$ for some strong bisimulation R [5].

We employ μ-calculus [1,4,8] to represent properties of a process. *Formulae* in μ-calculus are defined by the following BNF where $x \in \mathcal{X}$ and $a \in \mathcal{A}$:

$$f ::= \mathbf{tt} \mid x \mid f \vee f \mid \neg f \mid \langle a \rangle f \mid \mu x.f$$

The notion of freeness, boundness and scope for formulae in μ-calculus are defined similarly to the one for 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.

Satisfaction relation of formulae in a valuation $\mathcal{V}$ (written by $\models_{\mathcal{V}}$) is defined as follows where $p \in \mathcal{P}$:

(i) $p \models_{\mathcal{V}} \mathbf{tt}$.
(ii) $p \models_{\mathcal{V}} x$ if $p \in \mathcal{V}(x)$.
(iii) $p \models_{\mathcal{V}} f_1 \vee f_2$ if $p \models_{\mathcal{V}} f_1$ or $p \models_{\mathcal{V}} f_2$.
(iv) $p \models_{\mathcal{V}} \neg f$ if $p \not\models_{\mathcal{V}} f$, where $p \not\models_{\mathcal{V}} f$ means that p does not satisfy f.
(v) $p \models_{\mathcal{V}} \langle a \rangle f$ if there exists some q such that $p \xrightarrow{a} q$ and $q \models_{\mathcal{V}} f$.
(vi) $p \models_{\mathcal{V}} \mu x.f$ if $p \in S$ for all $S \subseteq \mathcal{P}$ such that $\forall q \in \mathcal{P}.q \models_{\mathcal{V}[S/x]} f$ implies $q \in S$.

The other logical notations $\mathbf{ff}$ ($\stackrel{\text{def}}{=} \neg\mathbf{tt}$), $f_1 \wedge f_2$ ($\stackrel{\text{def}}{=} \neg(\neg f_1 \vee \neg f_2)$), $[a]f$ ($\stackrel{\text{def}}{=} \neg\langle a \rangle \neg f$), and $\nu x.f(x)$ ($\stackrel{\text{def}}{=} \neg \mu x.\neg f(\neg x)$) can be defined as usual.

Proposition 1 [1] *Let $f(x)$ be a guarded formula, then we have:*

(i) $\mu x.f(x) \equiv \bigvee_{k>0} f^k(\mathbf{ff})$.
(ii) $\nu x.f(x) \equiv \bigwedge_{k>0} f^k(\mathbf{tt})$. □

Proposition 2 [1] *Processes p and q are strongly equivalent, i.e. $p \sim q$, iff $\mathcal{L}(p) = \mathcal{L}(q)$ where $\mathcal{L}$ is the set of all closed formulae and $\mathcal{L}(p) \stackrel{\text{def}}{=} \{f \in \mathcal{L} \mid p \models f\}$.* □

In the following, a formula, which expresses necessary and sufficient properties of a process, is defined. Let C be a set of process constants. $\mathcal{F}_C : \mathcal{P} \to \mathcal{L}$ is a function defined in the following way:

(i) $\mathcal{F}_C(\mathbf{0}) = \bigwedge_{a \in \mathcal{A}} [a]\mathbf{ff}$.
(ii) $\mathcal{F}_C(a.p)$
$= \langle a \rangle \mathcal{F}_C(p) \wedge [a]\mathcal{F}_C(p) \wedge \bigwedge_{b \in \mathcal{A}-\{a\}} [b]\mathbf{ff}$.
(iii) $\mathcal{F}_C(a_1.p_1 + \cdots + a_n.p_n)$
$= (\bigwedge_{i \in I} \langle a_i \rangle \mathcal{F}_C(p_i))$
$\wedge (\bigwedge_{i \in I} [a_i] \bigvee_{a_i = a_j} \mathcal{F}_C(p_j))$
$\wedge (\bigwedge_{a \in \mathcal{A}-A} [a]\mathbf{ff})$ where $n \geq 2$, $I = \{1, \ldots, n\}$ and $A = \{a_i \mid i \in I$ and $a_i \in \mathcal{A}\}$.
(iv) $\mathcal{F}_C(c) = \begin{cases} \nu x_c.\mathcal{F}_{C \cup \{c\}}(p) & \text{if } c \notin C \\ x_c & \text{if } c \in C \end{cases}$
where x_c is a fresh variable and $c \stackrel{\text{def}}{=} p$.

Proposition 3 [1,3] *Let p and q be processes:*

(i) $p \models \mathcal{F}_\emptyset(p)$.

(ii) $p \sim q$ *iff* $q \models \mathcal{F}_\emptyset(p)$. □

Proposition 4 [7] *Any formula can be equivalently converted to a formula without negation, i.e. a formula built up with* **tt**, **ff**, $\wedge$, $\vee$, $\langle a\rangle$, $[a]$, μ, *and* ν. □

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

2 Synthesis algorithm

A synthesis algorithm proposed here is an inductive one. It generates a process which satisfies given facts or properties of the intended target process. These facts are represented as formulae in μ-calculus and 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. It should be noted that p_o is neither known initially nor given in a precise manner.

Definition 5 *Let* U *be a set of pairs of formulae* $f \in \mathcal{L}$ *and signs* $+$, $-$, *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* $STL(\mathcal{X},\mathcal{A})$ *[1]. An element of* S *is called a* fact. □

Given an enumeration of facts, the algorithm synthesizes a process satisfying those facts. A process can be represented as a term p with a set $\{c_1 \stackrel{\text{def}}{=} p_1, \ldots, c_n \stackrel{\text{def}}{=} p_n\}$ of defining equations. In the algorithm, a process is represented as an identified process constant with 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. C can be omitted when it is not important.

In [2, 3], we proposed the synthesis algorithm which constructed recursive processes by formulae in μ-calculus without $\neg$, $\vee$ nor μ operator. Formulae with $\vee$ operator are ambiguous to synthesize processes. Especially, since a formula with μ operator (a μ-formula for short) involves infinitely many $\vee$ operators (see Proposition 1), it may cause backtracking infinite many times.
From this restriction, the limit process of the output sequence of the algorithm may not be equivalente to the intended target one. In fact, the limit process satisfies more properties than the target process. So, the formulae including $\vee$ or μ operators show that an output process of the algorithm satisfies some undesirable propeties. To complete the synthesis algorithm, the restriction for the input formulae must be relaxed.

The algorithm in Fig. 1 is an extension of the one in [2, 3]. This algorithm admits to input formulae involving μ or $\vee$ operators. Unfortunately, the following restrictions remain. First, any μ or ν operators must not occurr within the scope of the μ operator. Second, any μ operators must not occurr within the scope of the ν operator. To describe the algorithm, we adopt a language like Prolog, 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. 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. In the algorithm, the following abbreviations are adopted:

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

Algorithm 1 (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.*

Predicates: *See Fig. 1.* □

In the following, the extended parts of the algorithm from [2] will be explained. For the parts of [2], examples are given in Fig.2

One of the extensions is for the operator $\vee$ in (g). One of the subformula of $\vee$ is applied first to the current process, and if it happens to be inconsistent to the process, the other subformula is applied.

The other extension is for μ-forumlae in (c). The formula $\mu x.f(x)$ says that the target process satisfies $f(x)$ repeatedly finite times, but must not execute infinite many times. When $\mu x.f(x)$ is input to the synthesis algorithm, it checks that whether or not the current process satisfies $f(x)$ infinitely many times, i.e. whether or not the process has loops satisfying $f(x)$ infinitely. If it does, the algorithm backtracks to the point before one of such loops was made. But even after backtracking, the synthesized process may satisfy the formula $f(x)$ infinitely. In such cases, the backtracking will occur infinitely, so the synthesis algorithm never terminates. The basic idea for termination is to check the existence of such fatal loops by drawing colored lines. Each μ or ν-formula is given the identification color. When the algorithm makes branches in a process graph, i.e. an action prefix of the process, or traces them by $\langle a \rangle$ or $[a]$ oparetors, the formula draws a line with its own color beside them. In the former case, solid lines are used. In the latter case, dashed lines are used. Using colored lines, the algorithm finds whether or not backtracking procedure occurs infinitely many times. Suppose a formula $\mu x.$ $f(x)$ is input to the algorithm. The algorithm traces a current process or makes new branches to construct a process satisfying the input formula. If the traced path has no loops, $f(x)$ cannot be satisfied infinitely many times. More precisely, it can be the case that $f(x)$ is built up only with $\langle a \rangle$, $\vee$ and $\wedge$ operators, e.g. $\langle a \rangle \langle b \rangle x$. But $\mu x.\langle a \rangle \langle b \rangle x$ is logically equivalent to **ff**. So such formulae are reduced to **ff** when these formulae are input (by *remove-consistent-mu* of $mp(S)$ in Algorithm 1). If the traced path has loops but each loop is not fully (not partially) drawn by colored solid lines, then $f(x)$ cannot be satisfied infinitely. But there is a path drawn fully, then $f(x)$ may be satisfiable. See Fig.3 and 4 as examples. In Fig.3, each process (*) and (**) has a loop satisfying $[a][b]z$ infinitely. The loop of (*) is fully drawn by a colored solid line (for a-branch by a thin line and b by a thick line). So $\mu z.[a][b]z$ is inconsistent to the process (*). On the other hand, the (**)'s loop is not fully drawn, i.e. b-branch is not drawn by solid line. Thus (**) is backtrackable. However any fully drawn loop does not occur inconsistent. In Fig.4, the process (*) has the fully drawn loop. But formulae $\mu z.[b][a][a]z$, $\mu z.[b][a]z$ and $\mu z.[a]z$ does not occur inconsistent, since the order of the traced path by each formula differs from the one of colored line. Therefore the algorithm must also check whether or not the orders of them are identified. This is why the dashed lines are needed. Some pair of μ-formulae can also construct in-

```
mpstart :- mp({c0:{tt} ≝0 }).
  % The initial process is 0 .
mp(S) :- % S is a set of process definitions.
  read-formula(f),
  % Input a formula.
  convert-formula(f, f'),
  % μx.([a]x ∧ ⟨a⟩⟨b⟩tt)
  %    ⟶ μx.([a]x ∧ ⟨a⟩(x ∧ ⟨b⟩tt))
  % νx.([a][b]x ∧ ⟨a⟩⟨b⟩⟨c⟩tt)
  %    ⟶ νx.([a][b]x ∧ ⟨a⟩([b]x ∧ ⟨b⟩(x ∧ ⟨c⟩tt)))
  remove-consistent-mu(f', f''),
  % If a subformula of f' is of a form μx.g and g
  % has ⟨a⟩, ∨ and ∧ operators and variable x
  % but not others, then μx.g is modified by ff.
  % e.g. [a]μx.⟨a⟩⟨b⟩x ⟶ [a]ff.
  makeproc(c0, 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 for the next fact.
% program clauses of makeproc(c, S, f,X)
% c: the current process constant
% S: the current set of process definitions
% f: the current formula to be satisfied by c
% X: inferred process(set of process definitions)
% Note c,S,f are meta variables.

% tt ................................................. (a)
makeproc(ci, S,tt, S).
% xj : a bound variable corresponding to the
% formula νxj.f(xj) ................................(b)
makeproc(ci, S, xj,X) :-
  is-nu-variable(xj),
  % Is xj a variable of a ν-formula?
  makeproc-nu(ci, S, xj,X).
makeproc-nu(ci, S, xi, S).
makeproc-nu(ci, S, xj,X) :- % Where i ≠ j.
  S' ← (S[cj:Cj ≝ pi + pj] −
        {ci:Ci ≝ pi}){xj/xi}{cj/ci},
  makeproc(cj, S', ∧Ci,X). ......................... (b*)
makeproc-nu(ci, S, xj,X) :-
  is-remake, % can backtrack?
  makeproc(ci, S, f(xj),X). ....................... (b**)
% xj : a bound variable corresponding to the
% formula μxj.f(xj) ................................(c)
makeproc(ci, S, xj,X) :-
  is-mu-variable(xj),
  % Is xj a variable of a μ-formula?
  makeproc-mu(ci, S, xj,X).
makeproc-mu(ci, S, xi,X) :- fail.
makeproc-mu(ci, S, xj,X) :- % Where i ≠ j.
  no-colored-cycle,
  no-overlapped-mu-path,
  makeproc(ci, S, f(xj),X). ......................... (c*)
% ⟨a⟩f ............................................... (d)
makeproc(ci, S, ⟨a⟩f,X) :-
  transit(cj, a, ci), % ∃cj such that ci -a→ cj.
  free-variables(f,C),
  % get free variables of f to C.
  full-coloring(a, ci, cj,C),
  % draw lines colored by every color of C
  % beside the a-branch from ci to cj
  makeproc(cj, S, f,X).
makeproc(ci, S, ⟨a⟩f,X) :-
  get-new-process-constant(cj),
  free-variables(f,C),
  full-coloring(a, ci, cj,C),
  makeproc(cj, S[ci:Ci ≝ pi + a.cj, cj:{tt} ≝0 ],
    f ∧ (∧{fk | [a]fk ∈ Ci}),X). ...................... (d*)
% [a]f ............................................... (e)
makeproc(ci, S, [a]f, S) :-
  is-valid(∧Ci ⊃ [a]f). % ⊨ ∧Ci ⊃ [a]f
makeproc(ci, S, [a]f, S[ci:(Ci ∪ {[a]f}) ≝ pi]) :-
  not-transit(ci, a). % ci -a↛ .
makeproc(ci, S, [a]f,X) :-
  free-variables(f,C),
  broken-line-coloring(a, ci, cj,C),
  % draw dashed lines colored by every color of C
  % beside the a-branch for all cj.ci -a→ cj.
  forall(cj, ci, S[ci:Ci ∪ {[a]f} ≝ pi], f,X).
  % ∀cj.ci -a→ cj,
   makeproc(cj, S[ci:Ci ∪ {[a]f} ≝ pi], f,X).
% f1 ∧ f2 ............................................ (f)
makeproc(ci, S, f1 ∧ f2,X) :-
  makeproc(ci, S, f1,Y),
  makeproc(ci,Y, f2,X).
% f1 ∨ f2 ............................................ (g)
makeproc(ci, S, f1 ∨ f2,X) :-
  makeproc(ci, S, f1,X).
makeproc(ci, S, f1 ∨ f2,X) :-
  makeproc(ci, S, f2,X).
% νx.f(x) ............................................ (h)
makeproc(ci, S, νx.f(x),X) :-
  get-fresh-color(C),
  coloring-to-variable(x,C),
  makeproc(ci, S, f(xi),X).
% μx.f(x) ............................................ (i)
makeproc(ci, S, μx.f(x),X) :-
  get-fresh-color(C),
  coloring-to-variable(x,C),
  makeproc(ci, S, f(xi),X).
```

Fig. 1. The synthesis algorithm

finite branches. See Fig.5. In such cases, each colored line by μ-formulae are overlapped, and if one μ-formula draws a solid line, the others draw dashed lines. In the rest of paper, these pairs of lines are called overlapped μ-paths. In Fig.5, the process (*) has overlapped μ-paths. Each a-branch is drawn by thin solid line and thick dashed line. And the b-branch is drawn by reverse order of the above. The process (**) is the same case, though starting points of each lines are difference.

Theorem 6 *Assume that there exists a process satisfying initial segments $f_1, \cdots, f_n$ of an enumeration of facts, where $n \geq 1$. Assume Algorithm 1 outputs a set of process definitions S_{n-1} for the $n-1$ facts,*

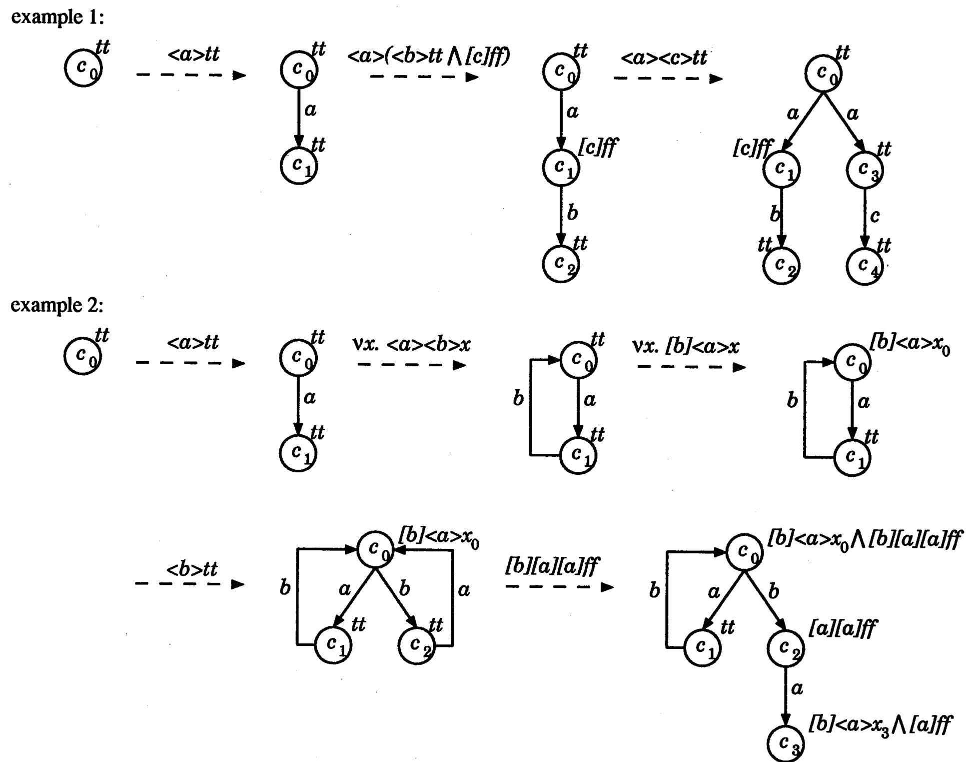

Fig. 2. Examples for the synthesis algorithm.

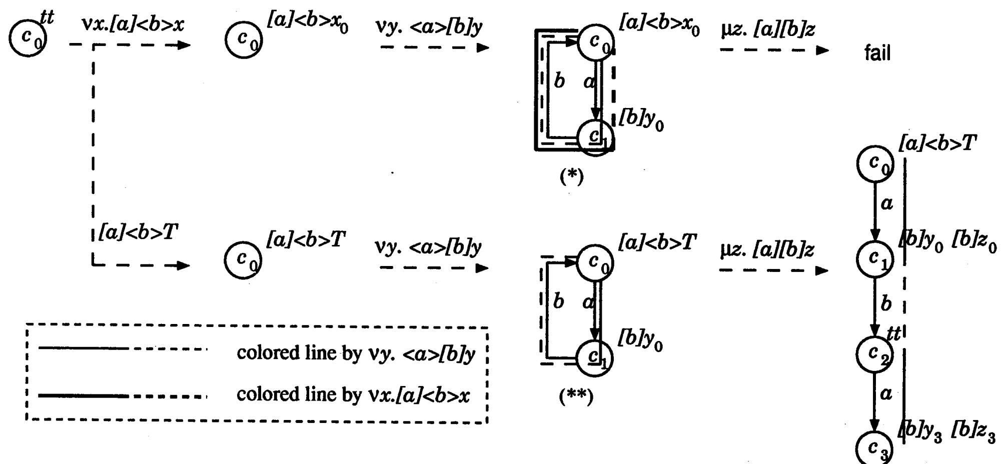

Fig. 3. An example for colored line.

$f_1, \cdots, f_{n-1}$ also. For the n-th fact, f_n, the followings are satisfied:

(i) The algorithm terminates and outputs a set of process definitions S_n with the process constant c_0 (the initial state of S_n).

(ii) c_0 with S_n satisfies f_n.

(iii) c_0 with S_n satisfies $f_1, \cdots, f_{n-1}$.

Proof.[sketch of proof] (i) 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**), (c*) and (d*) in Algorithm 1. Without using the above clauses, the algorithm terminates. There-

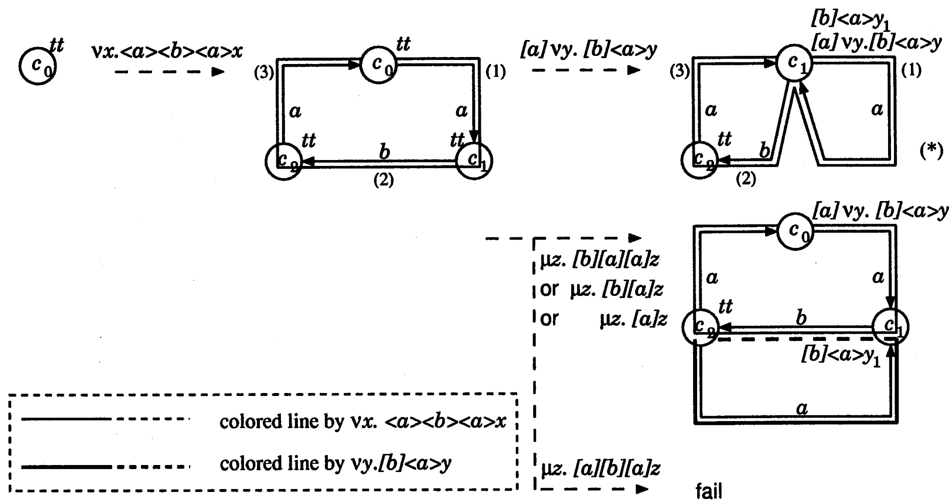

Fig. 4. An example for colored line.

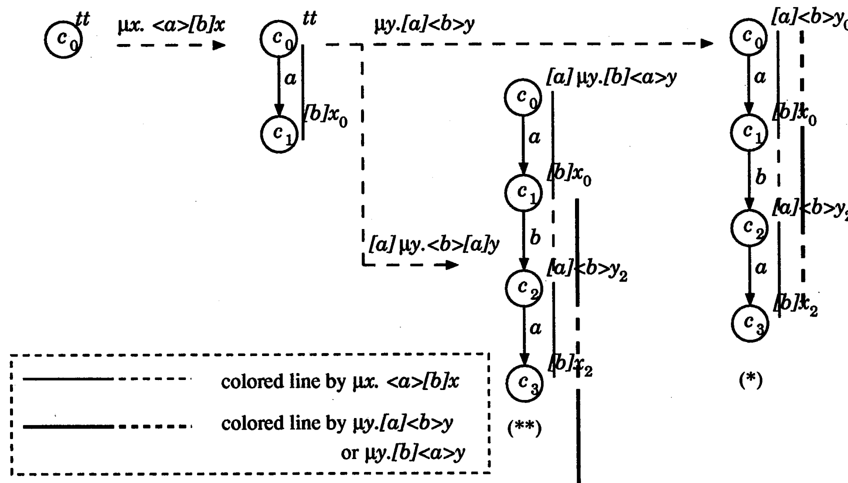

Fig. 5. An example for overlapped μ-paths

fore, it is sufficient to consider them only. Suppose the algorithm dose not terminate for some input formulae. Then the following seven cases must be considered.

(1) (b*) and (b**) occur infinitely, but none of other cases do.
(2) (c*) does, but none of other cases do.
(3) (d*) does, but none of other cases do.
(4) (b*),(b**) and (c*) do, but (d*) does not.
(5) (b*),(b**) and (d*) do, but (c*) does not.
(6) (c*) and (d*) do, but not (b*) and (b**).
(7) The whole cases do.

The impossibility of cases (1), (2) and (4) is already proved in [3]. The one of other cases is checked by the following predicates.

(3) by *convert-formula*(f, f') and *remove-consistent-mu*(f', f'') in $mp(S)$.
(5) by *no-overlapped-mu-path* in (c*).
(6) by *no-colored-cycle* in (c*).
(7) by *no-colored-cycle* in (c*) and *remove-consistent-mu*(f', f'') in $mp(S)$.

(ii) and (iii) The proof of them are almost same as in [3]. □

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 [6].

Definition 7 *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 8 *Assume p is an intended process, and the inferred sequence of processes by the Algorithm 1 converges in the limit to a process* p'. *Then* $p \sim p'$. □

The validity of Algorithm 1 is also shown by the following theorem.

Theorem 9 *Under the assumption of algorithm 1, if there exists a process p satisfying an enumeration of facts, the inferred sequence of processes by Algorithm 1 converges in the limit to a process* p' *such that* $p \sim p'$.

Proof. By Proposition 3, Theorem 6 and Lemma 8. □

3 Conclusion

This paper presents the synthesis algorithm, which is extention of the algorithm in [2,3], for a recursive process. We show the output sequence of the algorithm converses to a process which is strong equivalent to the target one in the limit.

References

[1] 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)

[2] Kimura, S., A. Togashi and N. Shiratori: "Synthesis Algorithm for Recursive Processes by μ-calculus", Lecture Notes in Artificial Intelligence, **872**, pp.379–394(1994)

[3] Kimura, S., A. Togashi and N. Shiratori: "Inductive Synthesis of Recursive Processes from Logical Properties", Information and Computation, (submitted)

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

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

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

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

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

[9] Togashi, A. and S. Kimura: "Inductive Inference of Algebraic Processes based on Hennessy-Milner Logic", Trans. IEICE, **E77-A-10**, pp.1594–1601(1994).