

Prolog のコルーチン・インタプリタの検証について

古川 康一 新田 克己

(電子技術総合研究所)

1. はじめに Prolog のインタプリタは、1971年にマルセイユ大学で Colmerauer 等によって開発されて以来、現在までに国内外の各所で作られている。Prolog インタプリタの特徴は、後戻り制御とユニファケーションであり、それをいかにうまく実現するかが問われてきた。後戻り制御を実現するには、手続きの再帰呼出しだけでは不十分であり、それ以上の制御メカニズムが必要であることが知られていた。本稿では、コルーチン制御メカニズムを用いて Prolog インタプリタの後戻り制御を実現した。インタプリタは AND/OR 探索木の2種の節、すなわち AND 節と OR 節に対応した2種のプロセスから成り立っている。これらのプロセスはプログラムの解釈実行時に動的に生成・削除されて動く。本稿では、まず次節で、これら2種のプロセス (AND プロセスと、OR プロセス) の記述を行い、その動きの概略を説明する。3節では、このインタプリタが正しく動くことを証明する。4節では、セミ・コルーチンによるインタプリタの記述との

等価性について述べる。最後に、コール・チン・インタプリタ
上での種々の拡張について、簡単にふれる。

2. AND/OR プロセスの記述 AND/OR プロセスを記
述するのに、本稿では、PAD (Problem Analysis
Diagram) を用いる。PAD はニ村によって作られたフ
ロー・チャート的一种であるが、記述が簡単で、労力が省け
る (実際に、必要とする線分の数が、通常のフロー・チャ
ートよりずっと少なく済む) のと同時に、プログラムの構造
が、図上によく表現される。

PAD による AND プロセスの表現を Fig.1 に示す。AND
プロセスは、主プログラム、および各クローズの右辺の実行
時に作成され、ゴール列の達成を行う。引数として受けと
れたゴール列の先頭から、順次ゴールを達成して行き、すべて
達成したら、その旨を親に報告する。ここで、親は主プログ
ラムを管理するコマンド・システムか、あるいは OR プロセ
スの1つである。各ゴールの達成は、個々に OR プロセスを
作り、それらに委ねる。個々の OR プロセスから成功の報告
を受ければつぎのゴールに進んでよいが、失敗した場合には
後戻り制御が必要になる。Fig.1 の  以下がそのための処理

である。図の ENTER (L) は、一度この AND プロセスが成功した後で他の所で失敗したときに、その失敗がこのプロセスにまで及んだときの再実行のための入口である。B-stack は、失敗した時にどこまで実行を逆上するかを記憶するためのスタックである。

OR プロセスの PAD 表現を Fig. 2 に示す。OR プロセスは個々のゴールに対して一つずつ作られ、そのゴールの達成のために働く。OR プロセスは、ゴールとマッチする候補手続きをプログラムの中から順に探してゆき、その手続き本体の達成を試みる。本体の実行は、そこで AND プロセスを一つ作り、それに委ねる。もしそれが成功なら自分も成功したことになる、その旨、親 (AND プロセス) に報告する。もし失敗したら、つぎの候補手続きを探して、ゴールの達成を再度試みる。すべての候補手続きが失敗したら、この OR プロセス自身、失敗である。そのとき、親へその旨報告して、自分は消滅する。

3. コルーチン・インタプリタの検証 コルーチン・インタプリタの検証を、本稿では、制御部分とユニファケーション部分に分けて行う。

3.1. 制御部分 はじめに、コルーチン・インタプリタが AND-OR 木をすべての可能性について探索していくことを証明する。

定理 1. Prolog プログラムの AND-OR 木が有限であれば、対応する AND/OR プロセスは、それらが起動されるたびに解がある限り 1 つずつ解を生成し、最後は "失敗" で終る。

この証明は、AND-OR 木の深さについての帰納法で行う。はじめに帰納法のベースとして、深さ 0 の AND プロセスおよび深さ 1 の OR プロセスについて上の定理を証明する。

(1). 深さ 0 の AND プロセス

深さ 0 の AND プロセスは、空のゴール列を処理する。Fig. 1 のプログラムにおいて、`curr-goal` は `nil` とするので、直ちに成功する。

もう一度起動されると、実行が E_2 から再開されるが、 B -stack は空であるので、今回は失敗する。

ところが高さ 0 の and 節はユニット・クローズなので、その解はそれ自身しかない。やえに、この場合は定理が成り立つ。

(2) 高さ 1 の OR プロセス

いま、高さ 1 の OR プロセス $O1$ が候補クローズ $C_1, \dots, C_i$ を調べて、 $C_1 \vee \dots \vee C_i$ のすべての可能な解を生成し E と仮定する。プログラムの制御は、Fig. 2 の m にある。そして、つぎの候補は C_{i+1} である。

いま、ゴールが C_{i+1} のクローズ・ヘッドとユニファイトされたとある。すると、 C_{i+1} に対応する AND プロセス $A1$ が作られ、起動される。それは高さが 0 の AND プロセスだから、前の証明より直ちに成功し、"while success" loop に入る。ここで解の親の AND プロセスに渡され、 $O1$ は E_2 で中断される。

つぎに再び起動されると、それは $A1$ を再び起動する。今回は、直ちに失敗するので、制御は "delete curr-andp", "recover state" とすすみ、"for all" loop へ戻る。候補クローズがなくなるまで、以上の過程をくりかえし、なくなる

れど、01 は失敗する。高さ1の or 節は、それに下っているすべてのユニット節のみが、その解の対象となり、実際にユニッティングが成功したもののみが解となるので、上の計算過程は、すべての可能な解を見つけ出している。ゆえに、この場合も定理が成り立つ。

(3) 深さが $2N$ の AND プロセス (インダクション・ステップ)

帰納法の仮定は、つぎの通りである。

「高さが $2N-1$ あるいはそれ以下の任意の OR プロセスは、すべての可能な解を、それが起動される毎に 1 つずつ生成し、最後は "失敗" で終る」

このインダクション・ステップの証明には、つぎのレマが必要である。

レマ $A0$ を、高さが $2N$ 以下の AND 節とし、 $G_1, \dots, G_n$ をそのゴール列とする。いま、制御が M_k あり、*current goal* が G_i であるとある。さらに、その時点での *B-stack* の状態を α とし、変数環境を β とする。すると A 中の実行を続けることにより、変数環境 β の下で、 $G_i \wedge \dots \wedge G_n$ のすべての可能な解を、 A 中で再起動される毎に 1 つずつ生

或レ、 G_i が失敗した直後に④に来た時点でのB-stackの状態は α になっている。

インダクション・ステップの証明は、このレニマの i を1とすればよい。

レニマの証明は、つぎの通りである。まず、つぎの補題が成り立つ。

補題 M においてB-stackのトップにあるORプロセスがゴール G_i を担当しているとする。そのとき、 $curr_goal$ は G_{i+1} である。

この補題は、"while fail" ループについての数学的帰納法により、容易に証明できる。レニマの証明も数学的帰納法にするが、 $G_1, \dots, G_i, \dots, G_n$ の i についての逆順で行う。

a) ベース. ($i=n$)

$curr_goal$ は G_n である。A ϕ を実行すると、 G_n を達成するためORプロセス ORP_n が生成され、起動される。

ORP_n が成功すると ORP_n はB-stackへつまれ、A ϕ はE2で中断する。つぎに再起動されると、④に行き、B-stack

から ORP_n を pop up し, それを再起動する。 ORP_n から解が生成される間, この同じ過程をくりかえす。ところで, もとの定理の証明中の帰納法の仮定により, ORP_n はすべての可能な解を生成する (ORP_n の深さは $2N-1$ 以下だから)。ゆえに, プログラムは, 環境 β の下で, すべての可能な解を生成する。もし ORP_n が失敗すると, それは削除され, 制御は ④ に行くが, そのときの B-stack の状態は, ORP_n を pop up した状態であるが, それは, もともと ORP_n を push する前の状態 α と同じである。ゆえに, この場合, レニマが成り立つ。

b) イニテリション・ステップ

$i \geq j+1$ の場合, レニマが成り立つと仮定して, $i=j$ のときにも成り立つことを示す。いま, $G_1 \wedge \dots \wedge G_{j-1}$ が達成され, 制御は, ゴール G_j の実行直前で M にあるものと考えよう。実行をつづけると, G_j のための OR プロセス ORP_j が生成され, 起動される。もしそれが成功すると, それは B-stack に push され, curr-goal が 1 つ進む。そこでのプロセス A の状態は, 帰納法の仮定における状態に等しい。すなわち, $G_1 \wedge \dots \wedge G_j$ が達成され, 制御は, ゴール G_{j+1} の実行直前で M にある。ゆえに, この時点での変数環境の下

で、 A 中の以下の実行は、 $G_{j+1} \wedge \dots \wedge G_m$ のすべての可能な解を、それが再起動される毎に、1つつ生成する。もし G_{j+1} が失敗すると、制御は④に行き、 B -stack の状態は、 G_{j+1} をつむ前の状態になっている。ゆえに、プログラムは、つぎに ORP_j を pop up して、それを再起動する。 ORP_j が解を生成する間、 A 中は、この過程をくり返す。 ORP_j は、すべての可能な解を生成することが（定理1の帰納法の仮定から）知られており、 G_j の各解について、 A 中は $G_{j+1} \wedge \dots \wedge G_m$ の可能な解をすべて生成するので、 A 中は、 $G_j \wedge G_{j+1} \wedge \dots \wedge G_m$ の（初期環境 β の下での）すべての可能な解を生成する。最後に ORP_j が失敗すると、 A 中は ORP_j を削除し、つぎに制御が⑤に達した時点での B -stack の状態は、 ORP_j を push する前の状態に戻っている。ゆえに、この場合も $\leq$ が成り立つことが分かる。

(4) 深さが $2N+1$ の OR プロセス (インタクション・ステップ)

本証明は、深さが1の OR プロセスの証明と本質的に変わらないので、ここでも省略する。

3. 2 ユニフィケーション部分

コピーによるユニフィケーションアルゴリズム [Brue80] を Fig.3.1 に示す。この図で、 $X\theta$ は証明しようとしているゴールのパラメータ、 X は選択されたクローズのヘッドのパラメータである。 $X\theta(X)$ が変数であれば、親(子)のプロセスの中に、その変数の値を記憶するスロット $XX\theta(XX)$ が必ず存在する。また、ある structure がコピーされる時、その中に含まれる変数は、対応するスロットとリンクされ、同一の値を持つことが保証される。

このアルゴリズムで、 $X\theta$ を現在の環境で評価した値は θ と、このプロセス内のスロットにより構成できる。また X についても同様である (証明は略す)

さらに、structure を結ぶポインタは常にプロセスから heap 又は copy stack へ向う。しかも、copy stack 中の情報が消えるのは backtrack の時のみである。従って、プロセスが消滅しても必要な structure は消えることなく、必ずアクセスすることができる。(structure sharing では、他のプロセス内の情報を参照するため、determinate に成功したプロセスを消滅させると、必要な情報をアクセスすることができなくなる可能性がある)

4. セミ・コルーチン・インタプリタとの等価性

[田村 82] は, Prolog インタプリタがセミ・コルーチンによって記述できることを示した。本稿で与えたコルーチン・インタプリタは, 本質的に田村等のセミ・コルーチン・インタプリタと同じである。田村等は, AND プロセスは, 実はプロセスである必要がないことを示した。その記述は, さらに AND プロセスをループでなく, 再帰呼出しを用いて実現している。田村等の AND プロセスをループ表現にしたものと, 我々のコルーチン・インタプリタを比較したが, Fig. 4 である。

Fig. 4 (a) は, 我々のコルーチン・インタプリタでの制御の流れを示している。図の丸で囲んだ部分は, 制御を中継しているだけで, これを除くことができる。そうしたのが, Fig. 4 (b) であり, これは田村等のセミ・コルーチン・インタプリタの制御の流れになっている。

我々のコルーチン・インタプリタでは, 制御の流れが, 隣り合う AND プロセスと OR プロセス間にしかないので, 理解が容易となり, 検証も, その分, 容易になっていると思われる。田村等のセミ・コルーチン・インタプリタは, 我々のコルーチン・インタプリタの一種の改良になっていると考えられるであろう。

5. おわりに

本論文では、コル-4に基づく Prolog インタプリタの記述と、その正当性の証明を与えた。本インタプリタの記述に用いた PAD 表現の上で、カットの導入、決定過程における B-stack の pop up による最適化などが、容易に表現できる ([Furukawa 82])。さらに、並列インタプリタへの拡張も可能であることも、ここで示した。

最も重要な最適化は tail recursion program を、スタックを使わずに実行するような仕組みの導入であるが、それは、すぐ後で削除してもよいプロセスを再利用することで達成できる。

本研究により、Prolog インタプリタの仕組みがより容易に理解され、各種の研究がさらに進むことを期待する。

参考文献

- [Bru 80] Bruynooghe M. "The Memory Management of Prolog Implementation" Logic Programming Workshop
- [Furukawa 82] Furukawa K et al. "Prolog Interpreter Based on Concurrent Programming" Prolog Conference
- [田村 82] 田村直之 神戸大修士論文

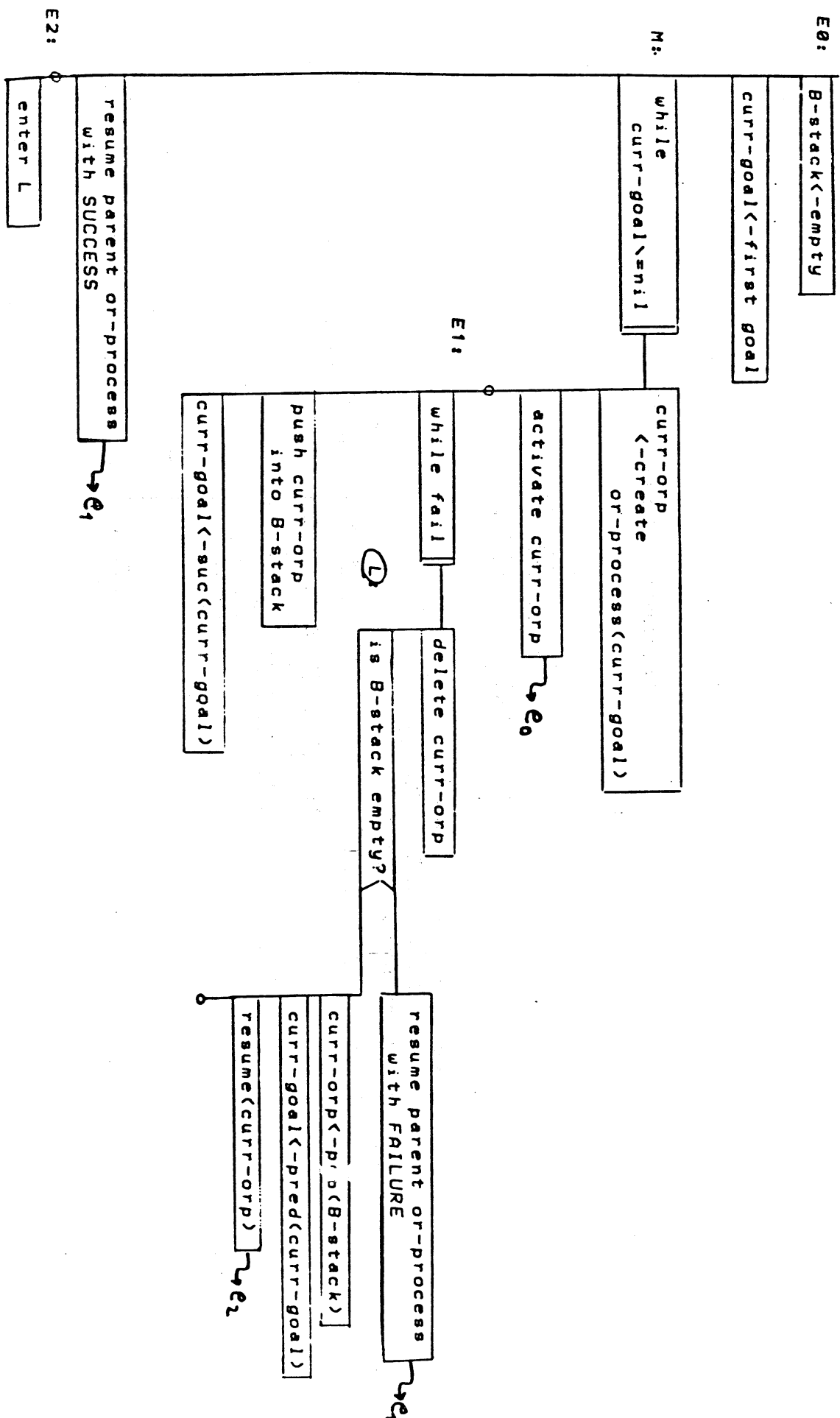


Fig.1. AND-PROCESS(GOALS)

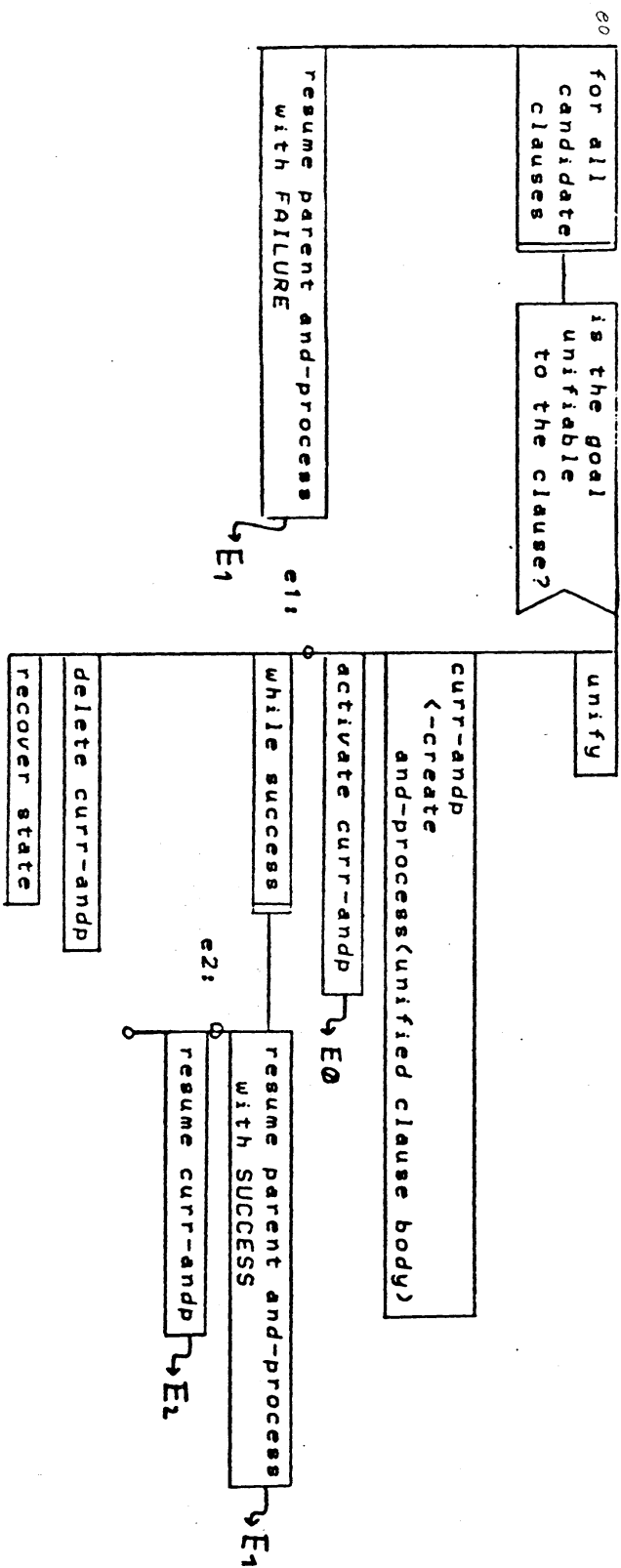


Fig.2. OR-PROCESS(GOAL)

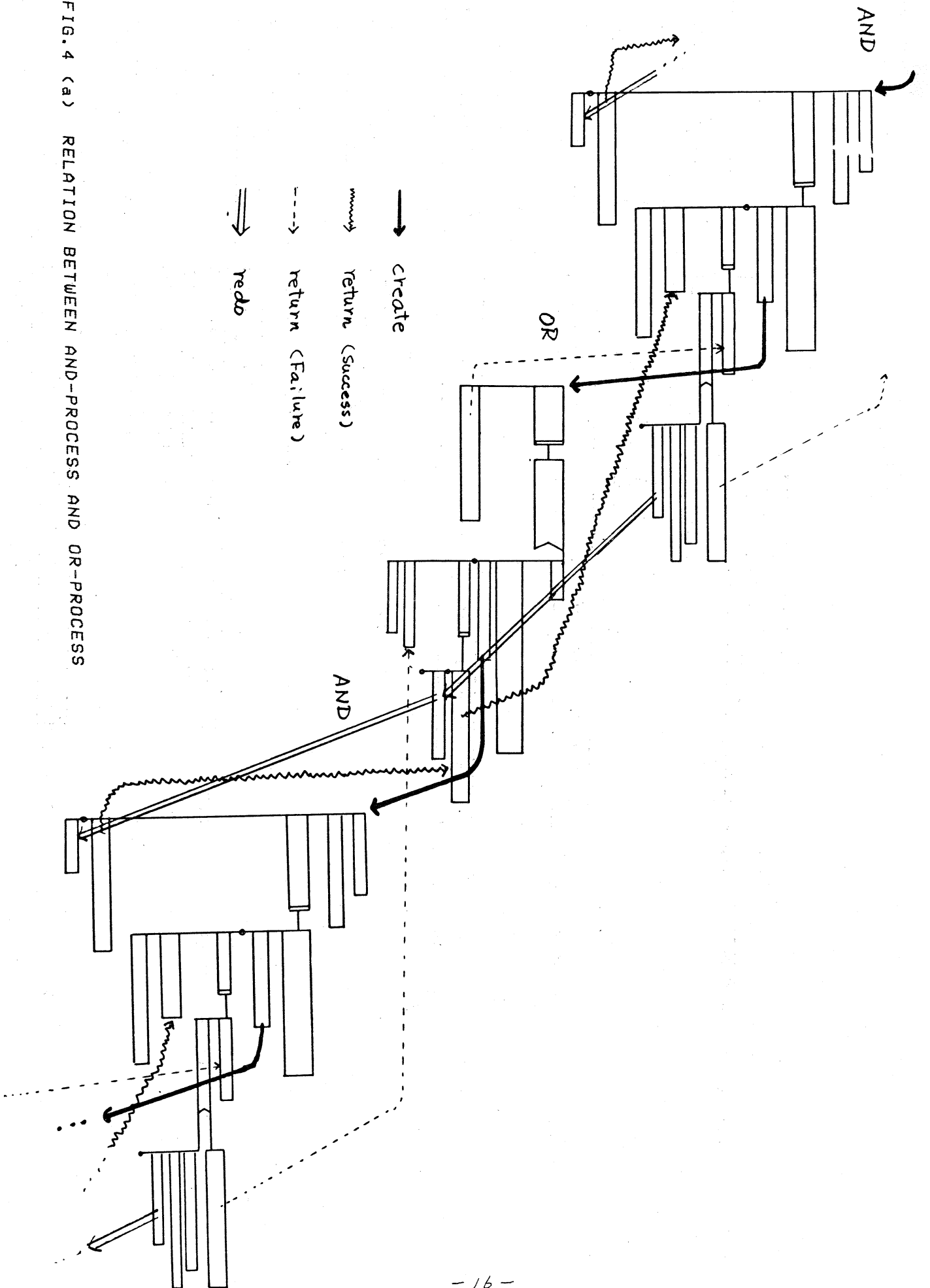


FIG. 4 (a) RELATION BETWEEN AND-PROCESS AND OR-PROCESS

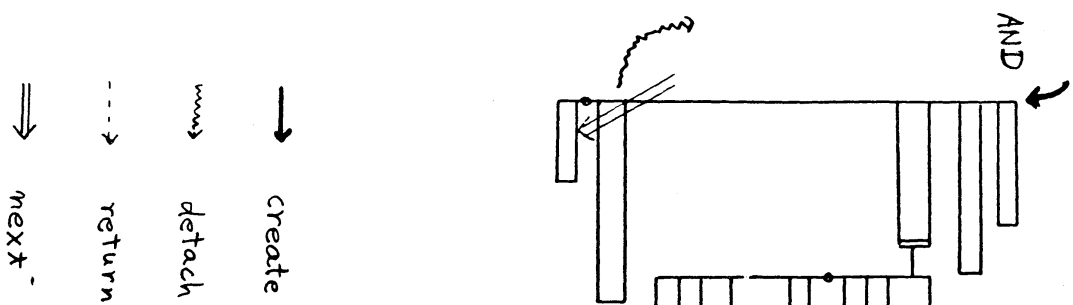


FIG. 4 (b) RELATION TO TAMURA'S COROUTINE

