知識工学と人工知能 47 － 1
（１９８６．７．１１）

依存関係を利用した階層的プランニング・システム　ＰＬＡＭ

金井　直樹
日本アイ・ビー・エム株式会社　サイエンス・インスティチュート

ＰＬＡＭ(PLAn Maker)は、目標とするプランを徐々に詳細なサブ・プランに分解して計画を作成する、階層的なプランニング・システムである。ＰＬＡＭは、ユーザ固有の問題に対するプランニング・システムを容易に構築できることを目標にして、開発された。

ＰＬＡＭでは、EP-Netというネットワークでプランを表現する。EP-Netは、Procedural Networkに、サブ・プラン間にある依存関係を表わすDependency linkという概念を導入したものである。サブ・プラン間に発生する相互作用などの問題は、このDependency linkを用いて、Criticsと呼ばれる知識が解決する。ＰＬＡＭでは、Criticsをユーザが容易に定義できるようになっている。本論文ではＰＬＡＭの知識表現、プランの作成方法などについて述べる。

PLAM : A Hierarchical Planning System Based on Dependency

Naoki Kanai

Science Institute, IBM Japan, Ltd.
5-19 Sanban-cho, Chiyoda-ku, Tokyo 102, JAPAN

PLAM (PLAn Maker) is a hierarchical planning system. This system treats goal plans abstractly, divides them into detailed sub plans, and makes really executable plans. Purpose of PLAM is to make domain dependent planning system easily on it. In PLAM, plans are expressed by EP-Net (Extended Procedural Network). EP-Net handles dependency between sub goals. Critics solves problems of interaction between sub plans. In PLAM, user can define new critics easily. In this paper, knowledge representation method and plan making method in PLAM are described.

１．はじめに

近年、AI(Artificial Intelligence)の産業界への応用が、大きな関心を集めている。なかでも工程設計、ライン設計、配車配船計画、ダイヤ作成などは、非常に複雑であり、専門家が経験をもとにして行っている。このような分野に対しては、プランニング・システムが非常に有効だと考えられる。プランニング・システムの目的は、解の探索空間を縮小すること、作成されたプラン内に存在するサブ・プラン間の競合を解決し、また、その解決方法の根拠を与えることなどがあげられる。

1971年に非階層的プランニング・システムＳＴＲＩＰＳ[1]が作られた。しかし、このシステムでは、サブ・プランを階層的に扱うことができず大きな問題を扱うことが難しかった。1975年に、Sacerdotiによって、階層的プランニング・システムＮＯＡＨ[2]が開発された。このシステムの大きな特徴は、サブ・プランの表現を階層的抽象的に扱えるようにしたこと、必要になるまでサブ・プラン間に時間的な前後関係を設定しないこと、サブ・プラン間に発生した相互作用などの問題を解決するための手法(これをCriticsと呼ぶ)を導入したことなどである。ＮＯＡＨを拡張したシステムとして、ＮＯＮＬＩＮ[3,4]、ＤＥＶＩＳＥＲ[5,6,7]などがある。ＮＯＮＬＩＮは、ＮＯＡＨの考え方に資源の概念を導入している。また、ＤＥＶＩＳＥＲは、時間の概念を導入している。しかし、これらのシステムは、Criticsを新しく作るのが難しく、システムの詳細を知らないユーザが、自分の問題に応じたプランニング・システムを構築することが難しいと考えられる。実際の問題にプランニング・システムを応用するためには、システムの詳細に立ち入ることなくユーザがシステムを利用できることが必要である。ＰＬＡＭは、このような要求を満たすことを目的として作られたプランニング・システムである。さらにＰＬＡＭでは、ＮＯＡＨで明らかになった問題にも対処している。以下に、ＰＬＡＭの詳細について述べる。

２．プランニング・システム　ＰＬＡＭ

ＰＬＡＭ(PLAn Maker)は、目標とする計画を抽象的なサブ・プランに分解し、徐々に詳細化して計画を作成する階層的なプランニング・システムであり、その基本的な考え方はプランニング・システムＮＯＡＨ[2]と同じである。

ユーザは、ＰＬＡＭに対して２種類の知識を定義することができる。その１つは、サブ・プランに関する知識であり、もう１つは、サブ・プラン間の相互作用を解消するための知識である。ＰＬＡＭは、これらの知識をもとにしてプランを作成する。プランは、EP-Net(Extended Procedural Network)という、サブ・プランの半順序集合で表現される。サブ・プラン間に相互作用が発見されると、上記の知識を用いて発生した問題を解決するように、サブ・プランの並べ替えなどを行う。この時ＰＬＡＭでは、必要な部分だけ順序付けを行う最小拘束アプローチの手法を用いる。

ＰＬＡＭの大きな特徴は以下の点である。

(1) サブ・プラン間にdependency linkという依存関係の概念を導入し、依存関係を維持するための操作を用意したこと。
(2) プラン作成中に問題が生じた場合、バックトラックを行うこと。
(3) 相互作用を解消するための知識であるCriticsが容易に定義できること。

なお、ＰＬＡＭは、VM/CMS上のProlog(VM/Programming in Logic)で記述されている。

３．ＰＬＡＭにおけるプランの表現

３．１　サブ・プラン定義

ＰＬＡＭでは、ある一つのオペレーションをサブ・プランとして抽象的に表現する。サブ・プランには、そのオペレーションが実行されることによって達成されるゴールの情報、そのゴールが達成されるために、何をするべきかという情報、また、そのオペレーションを実行するために要求される前提条件に関する情報が蓄えられる。サブ・プランはPrologのファクトの形で定義される。　サブ・プランには、２種類のタイプがある。一つはさらに詳細なサブ・プランに展開することができる抽象レベルにあるサブ・プランであり、これを abstract subplanと呼ぶことにする。abstruct subplanは、述語 state:defineで定義される。もう一つはそれ以上展開することができないサブ・プランであり、現実におけるある行動を表わす。これをprimitive subplanと呼ぶことにする。primitive subplanは、述語state:def_primitiveで定義される。具体的には次のように定義される。

```
state:define(Action名,Goal,Expand_information_list).
state:def_primitive(Action名,Goal,Expand_information_list).
```

大文字で始まるものは変数である。Expand_information_listはExpand_informationの一個以上のリストからなる。Expand_informationは、slotとvalueの二つ組リストのリストからなる。slotには以下のものがある。

(1) status : valueとして、そのexpand_informationが使用されるための条件を持つ。つまり、サブ・プラン定義の中で複数のExpand_informationがある場合に、この情報を用いて、どのExpand_informationを用いるかを決定する。
(2) depend : valueとして、そのExpand_informationを用いた場合に、そのサブ・プランが依存関係にある状態を持つ。
(3) result : valueとして、そのExpand_informationを用いた場合に変化する状態を持つ。primitive subplanの定義でのみ用いられる。
(4) expand : valueとして、どのような手順にそのサブ・プランを展開するかという情報を持つ。abstruct subplanの定義でのみ用いられる。

例として、図3-1にブロック・ワールドにおけるサブ・プランの定義を示す。

```
state:define(make_clear(X), clear(X),
                    {
                     {{status, var(X) },
                      {expand, always }
                     },
                     {{status, pstate(on(Y,X)) },
                      {expand, pgoal(clear(Y)) & primitive(on(Y,Z)) }
                    }}).
```

図3-1 サブ・プラン定義例

この例では、Xが値とユニファイしていない時には何もせず、Xが値とユニファイしていて、YがXの上にある時には、clear(Y)という状態にしてから on(Y,Z)という状態にせよということを表わしている。

求める状態を実現するためには、そのための適切なサブ・プランを用いる必要がある。ＰＬＡＭでは、求める状態を定義中の Goalに含むサブ・プランを探し、そのサブ・プランを用いている。そのような候補に成りえるサブ・プランが複数個あった場合には、その中の一つを選択しプランを作成する。もし、後で問題が生じてそれ以上プランが作成できないときには、バックトラックが発生し、他の候補のサブ・プランを用いる。

このような形式でサブ・プランを定義すると、そのサブ・プランを実行した結果の世界の状態の変化を、一つのサブ・プラン中に詳細に記述する必要がなくなり、サブ・プランの表現がより抽象的になる。

３．２　ＥＰ－Ｎｅｔ

ＰＬＡＭでは、プランはEP-Net(Expanded Procedural Network)というネットワークで表現される。 EP-Netは、Procedural Network[2]を拡張したものである。Procedural Networkでは、サブ・プランをノードで表わし、ノード間のリンクで時間的な前後関係を表わしている。ただし、Procedural Networkでは、時間的な前後関係のみ表現され、ノード間の依存関係を表現することができなかった。EP-Netでは、依存関係を表現するためにDependency linkという概念を導入している。EP-Netの簡単な例を図3-2に示す。四角いノードは、abstruct subplan、または、システムが使うノードを表す。カプセル型のノードは、primitive subplanを表す。点線は、Dependency linkを表す。

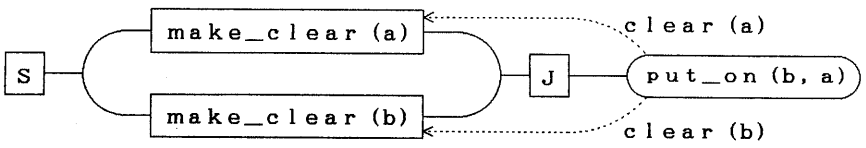


図3-2 EP-Netの例

3.3 Dependency link

EP-Netでは、あるノードと他のノード間に重要な依存関係があることをDependency linkによって表現する。依存関係は、あるサブ・プランと、そのサブ・プランを実行するために要求される世界の状態の間に発生する。この依存関係を考慮しないと正しいプランが作成できない可能性がある。たとえば、ノード間に相互干渉などの問題が発生した場合、それらの問題を解決するためにノードの順序付けを行う必要がある(これをノードのreorderingと呼ぶことにする)。この時、依存関係を考慮しないと、reorderingの結果、あるノードの前提条件である状態が破壊されてしまう可能性があり、作成されたプランの正当性は保証されない。図3-3にこの例を示す。ノードの上に記述されているのは、そのノードを実行した結果、変化した状態である。cとdが目的とする状態であるとする。ノード1では、aという状態が肯定され、ノード2では否定されているので、この二つのノード間には、相互作用が発生していると考えられ、この二つのノードを順序付けしないと、結果として得られるプランに曖昧性が発生する。そこでケース1、ケース2のようなreorderingが考えられるが、ここで、ノード3がノード1の状態aに対して依存しているとするとケース1のreorderingは誤っていることになる。また、サブ・プランをより詳細なサブ・プランに展開する時にも同様の状況が発生する可能性がある。時間的な前後関係にあるすべてのノードに依存関係があるとする方法もあるが、これは、プラン作成に制限を加え過ぎることになり正しい解を捨ててしまう可能性がある。

Dependency linkは、この様な問題に対処するために導入された概念である。ノードが、ある状態に対して依存関係にある時、そのノードとその状態を実現しているノードの間にDependency linkをはる。その時、Dependency linkは、属性としてその状態に関する情報を持つ。図3-3でDependency linkを考慮した場合には、ケース2が選択される。ただし、点線で表わされたのはDependency linkである。

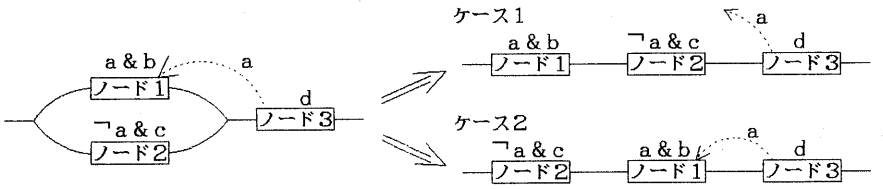


図3-3 reorderingの例

このように、Dependency linkを用いてノード間の依存関係を定義すれば、得られた解の正当性は保証され、また解の探索空間を狭め過ぎることもなく、プランを作成することができる。ＰＬＡＭでは、Dependency linkは、サブ・プラン定義におけるdepend slotで表現される。ＰＬＡＭは、サブ・プランを展開する時に、depend slotで定義された依存関係にある状態を実現しているノードをEP-Net中で探索し、条件を満たすノードの中から、そのサブ・プランより前で、かつ、最もそのサブ・プランに近いノードに対してDependency linkをはる。同様の考え方は、ＮＯＮＬＩＮ、ＤＥＶＩＳＥＲでも用いられている[3,6]。ＰＬＡＭでは、さらに、自動的にDependency linkを保持した上でEP-Netを操作するための機能を用意している。また、ＰＬＡＭは、Dependency linkをはる対象のノードが発見できなかった場合に依存関係にある状態を実現するためのサブ・プランを自動的に生成する機能も持っている。

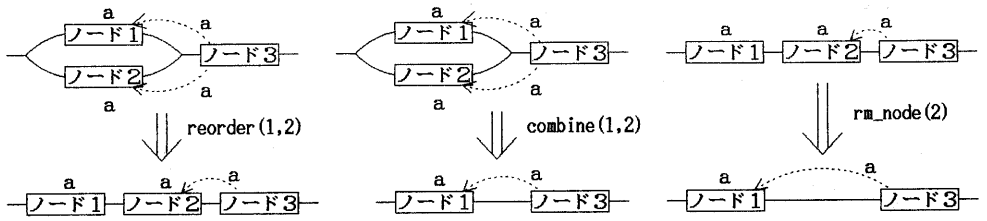


図3-4 EP-Net操作用述語の実行例

３．４　EP-Net操作用述語

ＰＬＡＭで用意されたEP-Net操作用の述語は、reorder、combine、rm_nodeの３種類である。それぞれ、Prologの述語として用意されている。これらの述語は、実行時にEP-Net中のDependency linkを自動的に維持する機能を持ち、Dependency linkが維持されない時にはfailする。

述語reorderは、同一パス上にない並列な二つのノードに対して用いられ、それらのノードに時間的な順序付けを行なう機能を持つ。述語combineは、冗長な二つの並列なノードに対して用いられ、それらのノードを一つにまとめる機能を持つ。述語rm_nodeは、ノードをEP-Netから除去する機能を持つ。それぞれの実行例を図3-4に示す。

３．５　プランの作成

次にＰＬＡＭでのプラン作成方法について述べる。まず、EP-Net中の全ての goal node(さらに詳細なサブ・プランに展開可能なノード)を、サブ・プラン定義に従って展開する。次に、EP-Net中に発生したノード間の相互作用などの問題をCriticsを用いて解決する。この時、ノードの展開とCriticsの適用をまとめてExpand phaseと呼ぶことにする。そして、展開可能なノードがなくなるまで、このExpand phaseを繰り返す。もし、プラン作成中に問題が発生しそれ以上プラン作成が続けられなくなったらバックトラックが起り、Expand_informationやCriticsのalternativeを用いて、プランニングを続行する。

４　サブ・プラン間相互作用の解消

４．１　ＴＯＭＥ

ＰＬＡＭではプラン間の相互作用を発見するためにＴＯＭＥ(Table Of Multiple Effects)[2]を用いる。ＴＯＭＥには、それぞれのノードにおける世界の状態の変化が記録される。同一の状態に関する変化が複数のノードで記録された場合、それらのノード間で相互作用が発生している可能性がある。ＴＯＭＥを用いることにより容易にそのようなノードを発見することができる。

４．２　ＣＲＩＴＩＣＳ

Criticsは、サブ・プラン間の相互作用を解消するための知識である。Criticsは、ノード間の相互作用をＴＯＭＥを用いて発見し、問題が生じていたらそれらの問題を解決するためにノードのreordering、選択を行う。ＮＯＡＨでは、General Purpose Criticsが用意されているが、前後関係にあるノード間に依存関係があるとしてCriticsを作成しているため、正当な解に到達できない可能性がある。また、Criticsの定義がシステムに埋めこまれているためシステムの詳細を知らないと、新しいCriticsを定義することが非常に難しい。プランニング・システムをユーザがアプリケーションに適用しようとするときに、Criticsが容易に表現できることは非常に重要なことである。

ＰＬＡＭでは、ノードの依存関係を自動的に保持するようなEP-Net操作用の述語を用意し、さらに、Critics発動条件などの情報を簡単に定義できるようにして、上記の問題を解決している。このため、General Purpose Criticsが解の探索空間を狭め過ぎることがなくなり、また、ユーザは、容易に新しいCriticsを定義することができる。この結果、目的とするプランニング・システムを簡単に構築することができる。

４．３　ＣＲＩＴＩＣＳの定義

Criticsは、Prologの節の形で定義される。以下にCriticsの定義を示す。

(1) state:critics(Critic名,target,Arg_patern) <- Search_arg_patern.
(2) state:critics(Critic名,fire,Arg_patern) <- Execute.
(3) state:critics(Critics名,when,When).
(4) state:critics(Critics名,type,Type).

Criticsに蓄えられる知識は、Criticsが発動すべき条件、問題を解決するための方法、発動する形式の３種類にわけられる。

Critics発動のための条件は、(1)の形で定義される。Search_arg_paternはPrologの述語であり、ユーザが定義する。Search_arg_paternの目的は、そのCriticsが対処すべき問題を発見し、その問題を発生させたノードの情報をArg_paternに返すことである。

問題を解決するための方法についての知識は、(2)の形で定義される。Executeは、ユーザが定義するPrologの述

語であり、上記のSerach_arg_paternによって求められた、Arg_paternの情報を用いる。ユーザは、ＰＬＡＭが用意しているEP-Net操作用の述語を用いることにより、Executeのプログラムを記述することができる。これらの述語は、EP-Net中の全てのDependency linkが維持されるかのチェックを行い、維持されない場合はfailする。

Criticsが発動する形式に関する知識は、(3)と(4)の形で表現される。(3)に定義されるのは、そのCriticsがいつ発動されるかの知識である。Whenに設定できる値とその意味は以下のものである。

at_first ： プランニングを始める時のみ発動できる。
at_last ： 全てのノードが展開された時のみ発動できる。
always ： いつでも発動できる。

(4)に定義されるのは、そのCriticsの性格に関する知識である。Typeは、そのCriticsの性格を表わす三つのタイプ(Type1,Type2,Type3)のリストで表される。それぞれのTypeに設定できる値とその意味を以下に示す。

Type1	necessary	： そのCriticsが、必ず成功しなければならない。
	if_possible	： 失敗してもよい。
Type2	all	： targetにより得られた全てのArg_paternに対して発動する。
	anyone	： targetにより得られたArg_paternのうち、一つに対して発動する。
Type3	once	： そのCriticsは、一度だけ実行される。
	once_a_loop	： そのCriticsは、1 Expand phaseについて一度だけ実行される。
	anytimes	： 何回でも実行される。

４．４ ＣＲＩＴＩＣＳとＤｅｐｅｎｄｅｎｃｙ ｌｉｎｋ

ＰＬＡＭでは、３つのGeneral Purpose Criticsを用意している。resolve_interaction_criticは、あるノードが並列状態にあるDependency linkを破壊する可能性がある場合に、ノードのreorderingを行なってDependency linkを維持するためのCriticsである。そのような状況は図4-1のように４通りが考えられる。

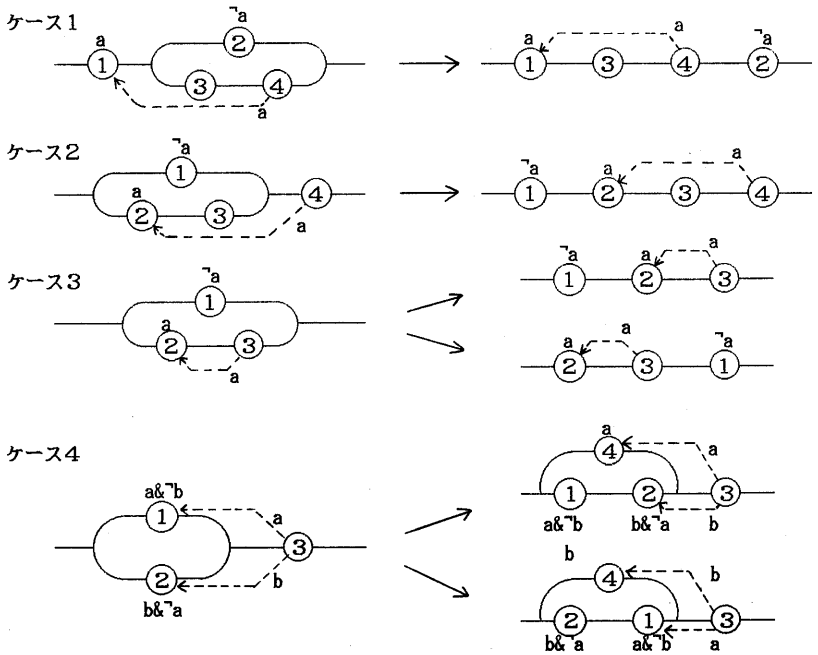


図4-1 resolve interaction criticの実行例

ケース１およびケース２の場合には、図のようにreorderingを行えばよい。ケース３の場合には、Dependency linkを保った上でのreorderingの方法が２通りある。ＰＬＡＭでは、まずこのうちの一つを選択しプラン作成を続行する。もし、後で問題が生じそれ以上プラン作成ができなくなった場合には、バックトラックしてもう一つのreorderingを行う。ケース４の場合には、どのようにreorderingしてもDependency linkが維持できない。この時には、今までのEP-Netに新しいノードを追加してDependency linkを作り、プラン作成を続ける。この操作は同一のDependency linkに関して一回だけ行われる。複数回この操作を繰り返すと無限ループに陥る可能性があるからである。

eliminate_redundant_criticとeliminate_same_criticは、冗長なノードを除去するためのCriticsである。もし、そのCriticsの実行によってDependency linkが維持できない場合には何も行わない。これらのCriticsは、ＮＯＡＨで用意されていたGeneral Purpose Criticsと目的は同じだが依存関係を適切に維持する点が異なっている。

５　ＰＬＡＭの実行例

ＰＬＡＭの実行例を次ページ図5-1に示す。実線の四角で表されたノードは、abstruct subplan、または、システムが使うノードである。点線のノードは、サブ・プランで実現しようとした状態が既に成り立っていることを表す。カプセル型のノードは、primitive subplanである。Expand phase 2では、状態clear(b)に関して相互作用の問題が発生し、それを解決するためにreorderingが行なわれている。この例を実行するのに3081上でcpu timeで約420msecかかる。この他にも工程設計のプランニング・システムの作成も行った。この結果、工程設計のためのCriticsが容易に作成できることが確認された。

６　おわりに

プランニング・システムＰＬＡＭは、サブ・プラン間の依存関係をDependency linkで表現すること、および、バックトラックを行うことにより、解空間を過剰に切捨ることなくプランを作成することできる。さらに、Dependency linkを維持するEP-Net操作用の述語を用意し、Criticsの持つ性格や発動のタイミングなどの情報を簡単に定義できるようにしたため、ユーザは容易にユーザ固有の問題に対処するためのCriticsを定義することができる。また、サブ・プランを抽象的に表現することができるため、ユーザは一度に大量の情報を処理する必要がない。

以上に述べた特徴を持つことにより、ＰＬＡＭは、ユーザの持つ問題に応じたプランニング・システムを容易に構築できる環境をユーザに提供している。

ＰＬＡＭの今後の課題としては、weak dependencyに関する問題がある。ＰＬＡＭでは、あるサブ・プランを展開する時に、status slotに定義された条件を満たすExpand_informationを用いる。しかし、reorderingの結果、この条件が成立しなくなる可能性がある。この場合には、他のExpand_informationを用いて再展開する必要がある。このように使用するExpand_informationと世界の状態の間には依存関係が存在する。これをweak dependencyと名付けた。ＰＬＡＭの現在のバージョンでは、このweak dependencyは考慮していないため、ユーザは、そのような事態が発生しないようにサブ・プランを定義する必要がある。ユーザに対して、より使いやすい環境を提供するために、今後、weak dependencyにも対処するようＰＬＡＭを拡張することを考えている。

参考文献

[1] Fikes, R. E., "STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol.2, pp.189-208, 1971.

[2] Sacerdoti, E. D., "A Structure for Plans and Behavior," Elsevier, New York. 1977.

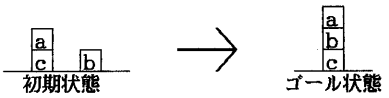
[3] Tate, A., "Generating Project Networks," Proc. IJCAI5, pp.888-893, 1977.

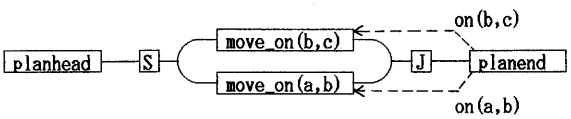
[4] Tate, A. and Whiter, A. M., "Planning with Multiple Constraints and an Application to a Naval planning probrem," Proc. 1st COnf. on AI Applications, pp.410-415, 1984.

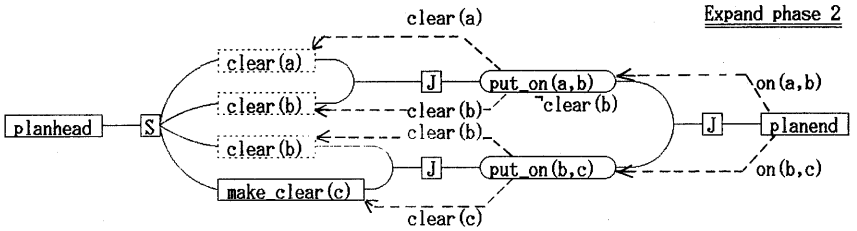
[5] Vere, S. A., "Planning in Time : Windows and Durations for Activities and Goals," IEEE trans. on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 3, pp. 246-267, May 1983.

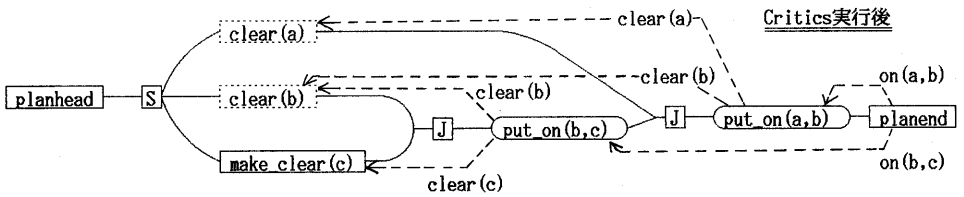
[6] Vere, S. A., "Splicing Plans to Achieve Misorderd Goals," Proc. IJCAI9, pp.1016-1021, 1985.

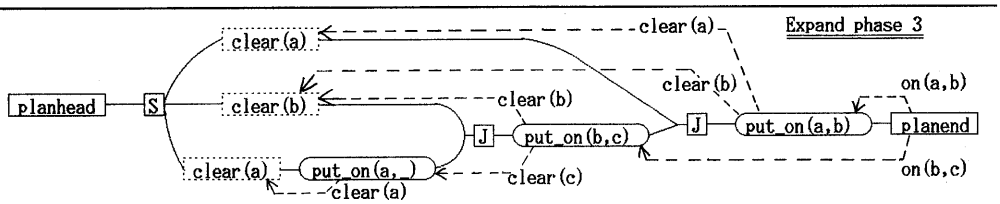
[7] Vere, S. A., "Temporal Scope of Assertions and Window Cutoff," Proc. IJCAI9, pp.1055-1058, 1985.











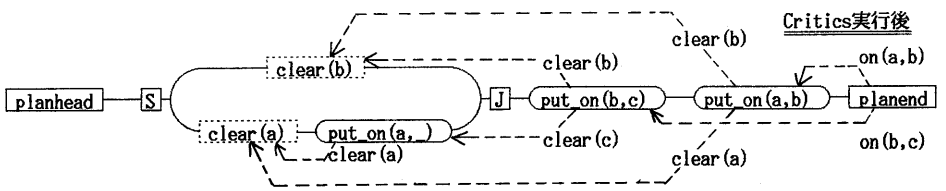


図5-1 ＰＬＡＭの実行例