動的環境で動作するエージェントのための新HTN プランニングフレームワーク

A New HTN Planning Framework for Agents in Dynamic Environments

林 久志
Hisashi Hayashi

株式会社 東芝 研究開発センター 知識メディアラボトリー
Knowledge Media Laboratory, Corporate Research and Development Center, TOSHIBA CORPORATION
hisashi3.hayashi@toshiba.co.jp

長 健太
Kenta Cho

(同 上)
kenta.cho@toshiba.co.jp

大須賀 昭彦
Akihiko Ohsuga

(同 上)
akihiko.ohsuga@toshiba.co.jp

keywords: agent, planning, replanning, logic programming, speculative computation.

Summary

In a dynamic environment, even if an agent makes a plan to obtain a goal, the environment might change while the agent is executing the plan. In that case, the plan, which was initially valid when it was made, might later become invalid. Furthermore, in the process of replanning, it is necessary to take into account the side effects of actions already executed. Nowadays, HTN planning is becoming popular among agent researchers because its task decomposition algorithm is efficient and suitable for joint planning in multi-agent systems. However, the dynamic replanning algorithm of HTN planning has not yet been established. In order to solve this problem, we have previously presented an agent life cycle that integrates HTN planning, action execution, knowledge updates, and plan modification. In that agent life cycle, the plans are always kept valid according to the most recent knowledge and situation. However, when one plan does not work and the agent starts using an alternative plan, it sometimes undoes some actions that do not have to be undone. The source of this problem lies in the need to have total-order plans. In other words, all the actions have to be sequentially ordered even when the order is not important in some parts of the plan. This paper extends our previous agent life cycle and presents new ways to avoid unnecessary action cancellation. This enables the agent to handle partial-order plans and to make use of two kinds of undoing actions: sequential undoing actions and concurrent undoing actions.

1. は じ め に

動的な環境への適応はエージェントの重要な課題である。動的に変化する環境では、昨日と同じ方法で今日同じ目標を達成できるとは限らない。動的な環境で、目標を達成する方法として、エージェントが推論をするという方法が挙げられる。実際、*Jini* [Tarau 99] や *MiLog* [Fukuta 00] などの移動エージェントは *Prolog* を使って推論をしながら行動する。*Plangent* [Ohsuga 97] の移動エージェントは、プランを作ってから行動を開始する。

一方、[desJardins 99] でサーベイされている通り、近年、マルチエージェントシステム等で、HTN プランニングが注目されている。HTN プラナーは抽象的なタスクをより具体的なサブプランに分解していくのだが、そのサブプランはプランライブラリーにあらかじめ用意されて

いるので一般的に高速である（つまり、サブプランを作る時間が浮く。）また、このプランニング方法により、エージェントは先に抽象的なプランだけ作っておいて、実行時に、そのエージェントが（または、そのエージェントからその抽象的なプランを受け取った別のエージェントが）抽象的なタスクの分解をすることができる。従って、より実行時の環境に適応しやすい。

実行時に推論してから行動すれば、動的な環境にある程度対応できるが、それだけでは不十分である。エージェントは、動的な環境の全てを把握しているとは限らないことが多く、推論した通りにアクションを実行できないことが多い。さらに、エージェントはプラン実行中に新しい情報を得て、推論に使った知識を更新するかもしれない。このような動的環境における問題は「再プランニング」の大きな課題である。代表的な人工知能の教科書 [Russel

95] に載っている標準的な半順序プランニングでは、あるアクションの「事前条件」とそれを満たす別のアクションの「事後条件(効果)」を繋いでプランを作成する。そして、そのようなアクション同士の因果関係を用いて、実行中のプランの正当性を確認し、必要に応じてプラン修正する。ところが、HTN プランニングのように、抽象的なゴールをより具体的なサブプランに置き換えていくプランニングでは、このような動的で漸次的なプラン修正方法は確立されていない。実際、HTN プラナーの *SHOP*[Nau 99] を使ったマルチエージェントシステム [Dix 03] でも、プラン実行中に動的にプランを修正することが課題となっている。

我々は、以前、これら問題 (HTN プランニングをするエージェントの動的なプラン修正) に対処するエージェントライフサイクル [Hayashi 02a] を提案し、*picoPlangent* [Cho 02] という移動エージェントシステムを使って実装 [Hayashi 02b] をした。このエージェントライフサイクルでは、エージェントは HTN プラナーまたは Prolog のように抽象的なタスクをより具体的なサブプランに分解して複数のプランを作成する。エージェントは複数のプランの中の 1 つを選択し、そのプランに従ってアクションを実行する。アクションの実行がうまくいかないときには、別の代替プランを用いる。

このエージェントライフサイクルでは、エージェントは、アクションを実行するたびに、実行したアクションの副作用を考慮し、必要に応じて、実行したアクションを取り消すアクションをプランの先頭に挿入し、各プランを修正する。これは、トランザクション論理プログラミング (TLP) [Bonner 93] のように、バックトラック時に、今まで実行したアクションをアンドゥするのに似ている。我々のエージェントは、複数のプランを保持しており、1 つのプランに従ってアクションを実行すると、別のプランに影響を及ぼす可能性があるため、このような実行済みアクションの取り消し作業は重要である。取り消しアクションの重要性は、Web サービス連携の分野でも認識されている。例えば、「あるスコープの実行中に不具合が生じた場合、そのスコープ内の処理を開始する前の状態に戻す」ことが、Web サービス連携用ワークフロー記述言語である *BPEL4WS* の仕様書 [BPEL4WS 03] に記述されている。

また、このエージェントライフサイクルでは、新情報を得たエージェントがプラン作成に使った知識 (プログラム) を更新するときに、無効なプランを消去し、逆に、新たに有効になるプランを作成する。このプラン修正方法は、*DSLDFN* [Hayashi 99, Hayashi 01] という論理プログラムの計算修正方法に基づいている。*DSLDFN* では、Prolog のように推論していくが、動的環境では、推論に使われた知識の一部が無効となり、探索木の一部の枝が無効になることがある。*DSLDFN* では、このように無効となる枝を除去し、推論を修正する。また、逆に、

新しい追加知識が有効になった場合には、探索木に新しい枝が伸びる場合がある。*DSLDFN* はこのような新しい枝を探索木に繋げて、推論を修正する。*DSLDFN* により、エージェントの最新知識に合わせてプランを更新することが可能になる。例えば、[Hayashi 99, Hayashi 01] では、ロボットの移動プランの作成・修正に *DSLDFN* が使われている。また、[Sato 02] では、*DSLDFN* に似た計算修正方法を用いて、マルチエージェントの投機的計算を行っている。マルチエージェントの投機的計算では、エージェントは、仮定に基づきプランを立てるが、外部からの情報により、その仮定が間違いであることに気づくと、エージェントはプランを修正する。[Sato 02] では、プラン実行中にプランを修正することはできないが、我々は、前述の取り消しアクションと *DSLDFN* を組み合わせることにより、プラン実行中に得られる新知識に合わせてプランを動的に修正できるようにした。新知識獲得後に推論を修正する別の論理プログラミングの方法が [Kowalski 99] に示されているが、この方法だと、知識を増やすことはできても、消すことはできない。我々は、知識の一部が訂正され、削除されても、プラン修正できるようにしている。

我々のエージェントライフサイクルにより、最新の知識 (プログラム) と最新のアクション実行状況に対して正しいプランを複数個エージェントに提供できるようになったが、エージェントが全順序プランしか扱えないことが原因による問題が残った。それは、不具合等のため、プランの実行を中断し、別の代替プランを実行するときに、実行済みのアクションを取り消すアクションを実行し、その後、その取り消したアクションを再び実行してしまうことがあるという問題である。例えば、部品を買って PC を組み立てるための以下の 2 つのプランを考えよう：

- [buy(a), buy(b), assemble(pc)]
- [buy(c), buy(a), assemble(pc)]

これらのプランは全順序プランである。前者のプランは、部品 A を買って部品 B を買って PC を組み立てるというプランで、後者のプランは、部品 C を買って部品 A を買って PC を組み立てるというプランである。仮に部品 A を買ったとすると、今までは、これらのプランを以下のように修正していた：

- [buy(b), assemble(pc)]
- [return(a), buy(c), buy(a), assemble(pc)]

注目すべきプランは後者で、この修正済みプランでは、部品 A を返品し、再度、部品 A を買うことになっている。後者のプランは無効ではないが、この取り消しアクションは明らかに無駄である。これは、後者の全順序プランが、部品 A と部品 C の買う順番を指定していることが原因である。このような無駄な取り消しアクションの実行をできるだけ避けるため、当論文では、部分的にアクションの実行順序を規定しないプラン (つまり、一種の半順序プラン) を扱える動的プランニングフレームワー

クを提案する．この新しいフレームワークを使ったエージェントライフサイクルでも，HTN プラニングによりプランを複数作成し，最新の知識，最新のアクション実行状況を考慮し，漸次的にプラン作成・修正を行う．このためには，以下のような技術が必要となる：

- 将来のプラン修正を考慮したプラン分解（3 章）
- アクション実行後のプラン修正（4 章と 5 章）
- 知識（プログラム）更新後のプラン修正（6 章）
- 上の 3 つの技術の統合（7 章）

さらに，知識更新後のプラン修正の効率性に関する実験結果は 8 章で解析し，独立した意味論を使ったエージェントライフサイクルの健全性については 9 章で示す．その前に，次の章で基本的な定義をする．

2. 基本定義

この章では，プラン，節，動的プラニングフレームワーク，アクションを消費可能・不可能なプラン，アクションデータ，アクション実行履歴等，最も基本となる定義を行う．

まず，項とリテラルを定義する．ここで使う言語は，定数，変数，関数，述語を含む．Prolog と同様に，定数，関数，述語は，英小文字で始まる英数字記号列で表す．変数は英大文字で始まる英数字記号列で表す．ただし，定数，関数，述語を表すメタ変数は，英大文字で始まる英数字記号列で表す．他のメタ変数も，英大文字で始まる英数字記号列で表す．

【定義 2・1】 項は，定数，変数，複合項のいずれかである．ただし，複合項は，以下の形をしている：

$$F(T_1, \dots, T_n)$$

ここで， F は引数が n 個 ($n \geq 0$) の関数で，各 T_i ($1 \leq i \leq n$) は項である．

また，リテラルは，以下の形をしている：

$$P(T_1, \dots, T_n)$$

ここで， P は引数が n 個 ($n \geq 0$) の述語で，各 T_i ($1 \leq i \leq n$) は項である．述語 P の引数が 0 個である場合には，リテラル $P()$ を P と書いてもよいこととする．

次にプランを定義する．

【定義 2・2】 プランは以下のいずれかの形をしている：

- $[X_1, \dots, X_n]$
- $\{Y_1, \dots, Y_n\}$

ここで， $n \geq 0$ で，各 X_i と各 Y_i ($1 \leq i \leq n$) はリテラル，または，プランである．あるプラン PLAN の中に出現する PLAN 以外のプランのことを PLAN のサブプランと呼ぶ．また，プラン $[]$ 及びプラン $\{\}$ を空プランと呼び，同一視する．空プランを $\emptyset$ と書いてもよいこととする．

上の定義では，プランはいわゆる半順序プランを示す．（ただし，説明を簡単にするため，記述できる半順序プラ

図 1 プラン（直列）

図 2 プラン（並列）

ンは限定されている．) 直感的には，プラン $[X_1, X_2, X_3]$ とプラン $\{Y_1, Y_2, Y_3\}$ の実行順序は，それぞれ，図 1 と図 2 が示す通りである．

【例 2・1】 図 3 は以下のプランの実行順序を表す：

$$\{[a1, a2], a3, [a4, a5]\}$$

次に節を定義する．

【定義 2・3】 リテラル LIT を定義する節は，

$$LIT \Leftarrow PLAN$$

の形をしており，PLAN はプランである．PLAN が空プランであるとき，節 $LIT \Leftarrow PLAN$ を事実と呼び，この節を単に LIT と表してもよいこととする．

上の定義では，直感的には， $LIT \Leftarrow PLAN$ は，PLAN を実行すると，LIT が満たされるということを意味する．節の意味は，後に，定義 9・1 で定義される

次に動的プラニングフレームワークを定義する．動的プラニングフレームワークはプラン作成に使われる．

【定義 2・4】 動的プラニングフレームワークは，

$$\langle A, D, P \rangle$$

の形をしている． A は，アクションと呼ばれるリテラルの集合である．アクションのインスタンスもまたアクションである． D は，動的反リテラルと呼ばれるアクションではないリテラルの集合である．動的反リテラルのインスタンスもまた動的反リテラルである． P は A に出現するアクションを定義しない節の集合である． P をプログラムと呼ぶ．

上の定義でアクションはエージェントが実行することが可能なリテラルのことである．プログラム中の動的反リテラルを定義する節は，将来，更新される可能性があることとする．プログラムは，抽象的なリテラルを満たすプランを格納するプランライブラリーである．

図 3 サブプランが出現するプラン

次にアクションを消費可能、不可能なプランを定義する．直感的には、あるプラン P に従うと次に行うべきアクションの 1 つが A であるとき、 P は A を消費可能であるという．我々が扱うプランは、部分的にリテラルの実行順序を規定しないことがあるため、このようなアクションは複数存在することがある．

【定義 2・5】 動的プランニングフレームワークにおいて、アクション A を消費可能なプランとは、以下の形のプランである．

- $[X_1, \dots, X_n]$
 - $n \geq 1$ で、 X_1 は、 A と同一化できるアクションであるか、 A を消費可能なサブプランである．
- $\{Y_1, \dots, Y_n\}$
 - $n \geq 1$ で、ある $Y_i (1 \leq i \leq n)$ は、 A と同一化可能なアクションであるか、 A を消費可能なサブプランである．

また、動的プランニングフレームワークにおいて、アクション A を消費不可能なプランは、以下の形のプランである．

- $[X_1, \dots, X_n]$
 - $n = 0$ であるか、または、 $n \geq 1$ で、 X_1 は、 A と同一化不可能なアクションであるか、 A を消費不可能なサブプランである．
- $\{X_1, \dots, X_n\}$
 - $n = 0$ であるか、または、 $n \geq 1$ で、各 $X_i (1 \leq i \leq n)$ が、 A と同一化不可能なアクションであるか、 A を消費不可能なサブプランである．

上の定義は、プラン $[X_1, \dots, X_n]$ は、次に満たすべきリテラルまたはサブプランは X_1 であることを示し、プラン $\{Y_1, \dots, Y_n\}$ は、次に満たすべきリテラルまたはサブプランは $Y_1, \dots, Y_n$ のどれでもよいことを示す．

【例 2・2】 動的プランニングフレームワークにおいて、 a_1, a_2, a_3, a_4, a_5 をアクションとしたとき、図 3 に示されるプラン $\{[a_1, a_2], a_3, [a_4, a_5]\}$ は、 a_1 と a_3 と a_4 を消費可能であるが、 a_2 と a_5 は消費不可能である．

次に、アクションデータを定義する．このアクションデータには、アクションの副作用を取り消すアクションの情報を入れる．アクションデータにある情報は、アクションを実行した後に、プランを修正するのに用いられる．

【定義 2・6】 動的プランニングフレームワークにおいて、アクション A のアクションデータとは、次のいずれかである．

- $\text{dataA}(A, \text{noSideEffect})$
- $\text{dataA}(A, \text{undo}(\text{seq}, A^-))$
- $\text{dataA}(A, \text{undo}(\text{con}, A^-))$
- $\text{dataA}(A, \text{cannotUndo})$

上で定義されるアクションデータの意味は、定義 9・1 で定義される．アクションデータの直感的な意味は、4 章でアクション実行後のアクションデータを使ったプラン修正方法を示しながら説明する．

図 4 プランの探索木

最後に、アクションの実行履歴を定義する．アクション実行履歴には、実行したアクションの順番だけでなく、それらのアクションの副作用に関する情報も書き込まれる．

【定義 2・7】 アクション実行履歴は、

$$[H_1, \dots, H_n]$$

の形 ($n \geq 0$) をしている．ここで、各 $H_i (1 \leq i \leq n)$ は (実行済みの) アクションデータである． $n = 0$ のとき、アクション実行履歴 $[]$ は、空であるという．

3. プラン分解と導出

この章では、プランの分解、導出方法を定義する．動的プランニングフレームワークでは、プラン導出後にもプログラムを更新する可能性があるため、プラン修正に必要な情報をプランに付加する．このようなプランの拡張をプランプラスという．また、プラン導出には、プランゴールが用いられる．プランゴールは 2 種類のプラン (プラス) を記録する：フロンティアプラン (プラス) と補助プラン (プラス) である．フロンティアプランは、探索木の探索済み部分木の先端のノードに対応する．例えば、図 4 の左側の探索木において、プラン 2 はプラン 1 から節 12 を用いて導出している．同様に、プラン 3 はプラン 1 から節 13 を用いて導出している．しかし、プラン 2 とプラン 3 からは、まだ他のプランを導出していない．この探索木において、フロンティアプランは、プラン 2 とプラン 3 である．プラン導出中は、フロンティアプランに出現するリテラルをプログラムの節を使ってサブプランに分解していく．例えば、図 4 では、フロンティアプランであるプラン 3 に出現するリテラルを分解してプラン 6 とプラン 7 を導出している．プラン 6 は節 36 を使って、プラン 7 は節 37 を使って導出している．プラン導出後、プラン 3 はもはやフロンティアプランではないが、そのかわりにプラン 6 とプラン 7 が新しいフロンティアプランとなる (分解されていないプラン 2 はフロンティアプランのままである)．分解したリテラルが動的リテラルである場合、リテラルを定義する節が将来変わる可能性がある．この場合、プラン 3 のようなフロンティアプランでないプランも補助プランとして記録する．将来、プログラムが更新され、新しい節が有効になると、補助プランから別のプランを導出できることがある．

【定義 3・1】 動的プランニングフレームワークにおけるプランプラスは、

(PLAN, CLAUSES, HISTORY)

の形をしている。PLAN はプランである。CLAUSES は動的リテラルを定義する節の集合である。CLAUSES に出現する各節は、プログラムに出現し、PLAN の前提節と呼ばれる。HISTORY はアクション実行履歴である。

プランの前提節とは、そのプランを作成するのに使われた節のうち、将来、無効になるかもしれない節である。前提節が無効になると、そのプランも無効になる。例えば、図 4 において、プラン 6 は動的リテラルを定義している節 3 6 を用いて作成されているので、節 3 6 をプラン 6 の前提節としてプランプラスに記録する。アクション実行履歴（定義 2・7）は、まだプランに反映されていない実行済みアクションのアクションデータ（定義 2・6）を記録する。

【定義 3・2】 動的プランニングフレームワークにおけるプランゴールは、

(PLANS⁺, SSP⁺)

の形をしており、PLANS⁺ と SSP⁺ はそれぞれプランプラスの集合である。PLANS⁺ に属する各プランプラスをフロンティアプランプラスと呼び、SSP⁺ に属する各プランプラスを補助プランプラスと呼ぶ。フロンティアプランプラスに出現するプランをフロンティアプランと呼び、補助プランプラスに出現する各プランを補助プランと呼ぶ。補助プランには、動的リテラルが 1 つ以上出現し、それらの動的リテラルの 1 つがマークされている。

この章で定義するプラン導出ルールでは、フロンティアプランに出現するリテラルをサブプランに分解していく。もし、分解されたリテラルが動的リテラルである場合には、分解されたプランを補助プランとして記録する。従って、補助プランには分解された動的リテラルが存在し、プラン導出ルールでは、この動的リテラルをマークしておく。補助プランは探索木の先端ではない中間ノードに相当し、将来、マークされた動的リテラルを定義する節がプログラムに追加されると、新しい枝がその中間ノードから伸びることになる。

次に節を使ってプランを分解する方法を定義する。Prolog と同様、同一化も行われる。

【定義 3・3】 PLAN2 は PLAN1 の新しいバリエーション^{*1}とし、PLAN2 のリテラル LIT2 の出現は PLAN1 の LIT1 に対応し、同じ位置に出現するとする。節 C の新しいバリエーションを LIT3 $\Leftarrow$ PLAN3 とし、LIT1, LIT2, LIT3 は同一化可能であるとするとする。PLAN1 に出現する LIT1 が選択されているとき、節 C による PLAN1 の分解プランとは、LIT2 と LIT3 を同一化し、PLAN2 から LIT2 の出現を

PLAN3 で置き換えることにより得られるプランである。

〔例 3・1〕 プラン $[x1(a), \underline{x2(b)}]$ に出現する $x2(b)$ が選択されているとき、節 $x2(V) \Leftarrow \{y1(V), y2(V)\}$ によるプラン $[x1(a), \underline{x2(b)}]$ の分解プランは、 $[x1(a), \{y1(b), y2(b)\}]$ である。

次にプランゴールの導出を定義する。直感的には、プラン導出とはプラン分解の繰り返しであるが、プラン導出中に、将来のプラン修正時に必要となる情報も記録する。

【定義 3・4】 動的プランニングフレームワークにおいて、プランゴール GOAL₁ からプランゴール GOAL_n ($n \geq 2$) までの導出とは、 n 個のプランゴールの列：

GOAL₁, ..., GOAL_n

で、各 GOAL_{k+1} は GOAL_k ($1 \leq k \leq n-1$) から以下のいずれかの導出ルールによって作成される。ただし、GOAL_k は以下の形をしているとする：

(PLANS⁺, SSP⁺)

p1 PLANS⁺ に属するフロンティアプランプラスの 1 つ：

(PLAN, CLAUSES, HISTORY)

を選択し、その内部のフロンティアプランの PLAN 内に出現するリテラル L を 1 つ選択する。ただし、L はアクションでも動的リテラルでもないこととする。L と同一化可能なリテラルを定義する全てのプログラムの節を C₁, ..., C_s ($s \geq 0$) とする。各 C_i ($1 \leq i \leq s$) による PLAN の分解プランを RPLAN_i とする。選択したフロンティアプランプラスを s 個のフロンティアプランプラス：

• (RPLAN₁, CLAUSES, HISTORY)

⋮

• (RPLAN_s, CLAUSES, HISTORY)

に置換することにより、GOAL_k から GOAL_{k+1} を作成する。

p2 PLANS⁺ に属するフロンティアプランプラスの 1 つ：

(PLAN, CLAUSES, HISTORY)

を選択し、その内部のフロンティアプランの PLAN 内に出現する動的リテラル L を 1 つ選択し、マークする。L と同一化可能な動的リテラルを定義する全てのプログラムの節を C₁, ..., C_s ($s \geq 0$) とする。各 C_i ($1 \leq i \leq s$) による PLAN の分解プランを RPLAN_i とする。選択したフロンティアプランプラスを s 個のフロンティアプランプラス：

• (RPLAN₁, {C₁} $\cup$ CLAUSES, HISTORY)

⋮

• (RPLAN_s, {C_s} $\cup$ CLAUSES, HISTORY)

に置換し、選択したフロンティアプランプラスを補助プランプラスとして SSP⁺ に加えることにより、

*1 新しいバリエーションは全ての変数を新しい変数に置き換えることによって得られる。

GOAL_k から GOAL_{k+1} を作成する．

導出ルール p1 は，節による定義が将来も変わらないリテラルを分解するのに対し，導出ルール p2 は，節による定義が将来変わるかもしれない動的反リテラルを分解する．これらの導出ルールでは，プラン内の選択されたリテラル L を定義する節が n 個あるとき，これらの節を使って同時に n 個のプランを作成する．例えば，図 4 では，節 3 6 と節 3 7 がプラン 3 に出現する同じリテラルを定義しているので，プラン 6 とプラン 7 は同時に作成する．

導出ルール p2 では，分解に使った節が将来無効になる可能性を考慮し，その分解に使った節をプランプラスに記録する．例えば，図 4 では，動的反リテラルを定義する節 3 6 はプラン 6 の前提節としてプランプラスに記録する．同様に，節 3 7 はプラン 7 の前提節として記録する．また，将来，動的反リテラルを定義する節が新たに有効になり，その新しい節を使って，選択したプランを分解できるかもしれない．そのために，選択されたプラン（プラス）を補助プラン（プラス）として記録する．例えば，図 4 では，プラン 3 を補助プランとして記録する．

〔例 3・2〕 動的プランニングフレームワークにおいて， $x2(V)$ は動的反リテラルであるとする．例 3・1 のように，プランを分解するとき，導出ルール p2 は，分解に使われた節 $x2(V) \leftarrow \{y1(V), y2(V)\}$ を前提節として分解後のプラン $[x1(a), \{y1(a), y2(a)\}]$ と共に同じフロンティアプランプラスに記録する．また，導出ルール p2 は，分解前のプラン $[x1(a), x2(b)]$ を補助プランとして記録し，選択した $x2(b)$ をマークする．

4. アクション実行とプラン修正

我々が定義するエージェントは，プランを複数持つ．プランの実行がうまくいかないとき，もしくは，他のプランがより魅力的になったときには，プランの実行を中止し，別のプランに変更してもよい（ただし，複数あるプランのうち，どれが魅力的かを判断する基準の策定は，当論文の範囲外である．）しかしながら，あるプランに従って実行したアクションが別のプランの実行を妨げるかもしれない．このように，他のプランに影響を及ぼすアクションを「副作用のあるアクション」と呼ぶ．副作用のあるアクションを実行したときには，その副作用を取り消すアクションを用いてエージェントが保持している複数のプランを修正する．この章では，アクション実行後のプラン修正方法を中心に定義する．まず，実行済みアクションデータを踏まえてプランを修正する方法を定義する．

【定義 4・1】 動的プランニングフレームワークにおいて，アクション A 実行後のプラン修正ルールは，以下の通りである．ただし，PLAN を修正対象のプランとし，PLAN

は A を消費可能か消費不可能なプランとする．2 つ以上のプラン修正ルールが適用可能な時は，そのうちの 1 つだけを適用する．

PLAN が A を消費可能であるとき，

- a1 PLAN が $[A', X_1, \dots, X_n] (n \geq 0)$ で，A と A' が同一化可能ならば，A と A' を同一化し，PLAN を $[X_1, \dots, X_n]$ に修正する．
- a2 PLAN が $[SUBPLAN, X_1, \dots, X_n] (n \geq 0)$ ならば，SUBPLAN を A 実行後のプラン修正ルールに従って修正する．
- a3 PLAN が $\{X_1, \dots, X_n\}$ で，ある $X_i (1 \leq i \leq n)$ が A と同一化可能ならば，そのような X_i を 1 つだけ選択し， X_i と A を同一化し， X_i を PLAN から削除する．
- a4 PLAN が $\{X_1, \dots, X_n\}$ で，ある $X_i (1 \leq i \leq n)$ が A を消費可能なサブプランならば，そのような X_i を 1 つだけ選択し， X_i をアクション A 実行後のプラン修正ルールに従って修正する．

PLAN が A を消費不可能であるとき，DATA が A のアクションデータとすると，

- b1 DATA が $dataA(A, noSideEffect)$ であるならば，PLAN は修正しない．
- b2 DATA が $dataA(A, undo(seq, A^-))$ であるならば，PLAN を $[A^-, PLAN]$ に修正する．
- b3 DATA が $dataA(A, undo(con, A^-))$ であるならば，PLAN を $\{A^-, PLAN\}$ に修正する．
- b4 DATA が $dataA(A, cannotUndo)$ であるならば，PLAN を削除する．

プラン修正ルール a1 から a4 は，実行済みのアクションをプランから消去する．例えば，アクション a4 を実行した後，図 3 で示されるプラン $\{[a1, a2], a3, [a4, a5]\}$ は， $\{[a1, a2], a3, [a5]\}$ に修正される．

プラン修正ルール b1 から b4 は，実行したアクション A がプラン PLAN と関連がない場合の PLAN の修正方法である．A が副作用を持たなければ，PLAN を修正する必要はない（b1）が，実行済みアクションが副作用を持つ場合（b2, b3, b4）に PLAN を使うときには A を取り消すアクション A^- を実行する必要がある．取り消しアクション A^- を実行する場合には，図 5 のように A^- を PLAN を実行する前に行わなければならない場合（b2）と，図 6 のように， A^- をいつ実行してもよい場合（b3）に分けられる．A の副作用を取り消す必要があるのに，取り消すことができない場合（b4）には，PLAN を活用することはできないので，PLAN を破棄する．

例えば，PC を組み立てるプラン $[assemble(pc)]$ を実行する前に，PC の生産ラインを分解するアクション $scrap(productionLine)$ を実行してしまったら，プラン $[assemble(pc)]$ を実行する前に，PC の生産ラインを組み立てるアクション $build(productionLine)$ を実行すべきであることは，直感的に理解できる．この場合，プ

図 5 直列取り消しアクション

図 6 並列取り消しアクション

ランは $[\text{build}(\text{productionLine}), [\text{assemble}(\text{pc})]]$ に修正すればよい。 $\text{scrap}(\text{productionLine})$ のアクションデータは以下のように記述する：

- $\text{dataA}(\text{scrap}(\text{productionLine}), \text{undo}(\text{seq}, \text{build}(\text{productionLine})))$

一方、プラン $[\text{assemble}(\text{pc})]$ を実行しようとしているときに、PC を組み立てるのに必要でない部品 x を買うアクション $\text{buy}(x)$ を実行してしまった場合、部品 x を返品するアクション $\text{return}(x)$ を実行することが必要であるが、いつ返品するかはあまり重要ではない。このような場合、プランは $\{\text{return}(x), [\text{assemble}(\text{pc})]\}$ に修正すればよい。 $\text{buy}(x)$ のアクションデータは以下のように記述する：

- $\text{dataA}(\text{buy}(x), \text{undo}(\text{con}, \text{return}(x)))$

また、 x を返品できないときには、活用できないプラン $[\text{assemble}(\text{pc})]$ を破棄する。このような場合、 $\text{buy}(x)$ のアクションデータは以下のように記述する：

- $\text{dataA}(\text{buy}(x), \text{cannotUndo})$

なお、 x の価格を調べるといった副作用のないアクション $\text{checkPrice}(x)$ を実行しても、このアクションと関連のないプラン $[\text{assemble}(\text{pc})]$ を修正する必要はない。この場合、 $\text{checkPrice}(x)$ のアクションデータは以下のように記述する：

- $\text{dataA}(\text{checkPrice}(x), \text{noSideEffect})$

定義 4.1 のプラン修正方法は、PLAN は A を消費可能か消費不可能であると判断できるときにしか用いられない。しかしながら、例えば、リテラル lit がエージェントの実行できるアクションでない場合には、プラン $[\text{lit}, \dots]$ は、A を消費可能か消費不可能か判断できない。しかしながら、将来、この抽象的なリテラル lit が実行可能なアクションに分解される可能性がある。プラン分解後、このようなプランに修正ルールを適用するために、アクション実行履歴をプランプラスに記録する。

【定義 4.2】 動的プランニングフレームワークにおいて、アクション A 実行後のプランプラス修正ルールは、以下の通りである。ただし、修正対象のプランプラスを、

$$(\text{PLAN}, \text{CLAUSES}, [\text{H}_1, \dots, \text{H}_n])$$

($n \geq 0$) とする。

c1 $n = 0$ で、かつ、PLAN が A を消費可能か消費不可

能なプランであるならば、PLAN をアクション A 実行後のプラン修正ルール（定義 4.1）に従って修正する。

c2 $n \geq 1$ 、または、PLAN が A を消費可能でも消費不可能でもないならば、修正対象のプランプラスを

$$(\text{PLAN}, \text{CLAUSES}, [\text{H}_1, \dots, \text{H}_n, \text{H}_{n+1}])$$

に修正する。ただし、 H_{n+1} は A のアクションデータ*2 である。

最後にプランゴールの修正方法を定義する。

【定義 4.3】 動的プランニングフレームワークにおいて、アクション A 実行後のプランゴール修正ルールは、以下の通りである。ただし、GOAL を修正対象のプランゴールとする。

d1 GOAL に出現する各プランプラスをアクション A 実行後のプランプラス修正ルール（定義 4.2）に従って修正する。

あるプランがあるアクションを消化可能か消化不可能か判断できなくても、そのプランを分解したのちに、その判断ができる場合がある。このような場合には（下の定義の e3 では）プランを分解した後に、そのプランをアクション実行履歴に従って修正する。

【定義 4.4】 動的プランニングフレームワークにおいて、プラン分解後のプランプラス修正ルールは、以下の通りである。ただし、修正対象のプランプラスを、

$$(\text{PLAN}, \text{CLAUSES}, [\text{H}_1, \dots, \text{H}_n])$$

($n \geq 0$) とする。

e1 $n = 0$ ならば、修正対象のプランプラスを修正しない。

e2 $n \geq 1$ で、かつ、 H_1 がアクション A のアクションデータで、かつ、PLAN が A を消費可能でも消費不可能でもないならば、修正対象のプランプラスを修正しない。

e3 $n \geq 1$ で、かつ、 H_1 がアクション A のアクションデータで、かつ、PLAN が A を消費可能か消費不可能であるならば、修正対象のプランプラスを、

$$(\text{NEWPLAN}, \text{CLAUSES}, [\text{H}_2, \dots, \text{H}_n])$$

に修正する。ただし、NEWPLAN は、A 実行後のプラン修正ルール（定義 4.1）に従って PLAN を修正することにより得られるプランプラスとする。可能ならば、修正されたプランゴールを、同様にしてさらに修正する。

上の定義の e1 では、プランに反映されていないアクションの実行履歴がないので、プランを修正する必要はない。e2 では、プランに反映されていないアクションの実行履歴はあるが、プランを修正するには更にこのプランを分解する必要がある。e3 では、アクション実行履歴

*2 A のアクションデータが入手可能であることを仮定している。その入手方法は規定しない。

に記録されているアクションの中で、最初に実行されたアクションをプランに反映させている。

【例 4・1】アクションを消費不可能な抽象的なプランを分解して、具体的なプラン $[buy(b), assemble(pc)]$ を導出したとする。このプランを導出する前に、アクション $buy(b)$ が実行されていると、そのアクションデータがプランと一緒に以下のようなプランプラスに記録されている：

```
([buy(b), assemble(pc)],  $\emptyset$ ,
 [dataA(buy(b), undo(con, return(b))))])
```

これをプラン分解後のプランプラス修正ルール e3 で修正すると、以下ようになる：

```
([assemble(pc)],  $\emptyset$ , [])
```

実行済みアクション $buy(b)$ はプランから削除され、 $buy(b)$ のアクションデータはアクション実行履歴から消去されている。このようにして、プランを具体化したときに、過去に実行したアクションをプランに反映させることができる。また、別のプランプラス：

```
([buy(c), assemble(pc)],  $\emptyset$ ,
 [dataA(buy(b), undo(con, return(b))))])
```

を、同様にして、プラン分解後のプランプラス修正ルール e3 で修正すると、以下ようになる：

```
({return(b), [buy(c), assemble(pc)]},  $\emptyset$ , [])
```

修正されたプランには、実行済みアクション $buy(b)$ のアクションデータに基づいて、 $return(b)$ が挿入されている。また、 $buy(b)$ のアクションデータはアクション実行履歴から消去されている。

プランを分解した後には、以下のようにプランゴール内の各フロンティアプランをチェックし、必要ならば修正する。

【定義 4・5】動的プランニングフレームワークにおいて、プラン分解後のプランゴール修正ルールは、以下の通りである。ただし、GOAL を修正対象のプランゴールとする。

f1 GOAL に出現する各フロンティアプランプラスをプラン分解後のプランプラス修正ルール（定義 4・4）に従って修正する。

5. 状況依存型取り消しアクション

当論文では、議論を簡単にするため、アクションの副作用の有無や取り消し方法が固定であり、それらの情報はアクションデータとして与えられていると仮定している。ところが、アクションの取り消し方は、状況に応じて変化するという問題がある。我々は、以前、全順序プランしか使わない旧動的プランニングフレームワークを用い、移動エージェントシステム [Hayashi 02b] を実装し

たが、ここでも、同様の問題があった。移動エージェントは、ネットワークに繋がれた複数のコンピュータを移動して仕事をするソフトウェアのことであるが、エージェントが現在いるコンピュータから別のコンピュータ $pc2$ に移動するアクション $goto(pc2)$ を取り消すアクションは、どのコンピュータから $pc2$ に移動するかによる。例えば、エージェントが $pc1$ から $pc2$ に移動するときには、 $goto(pc2)$ の取り消しアクションは $goto(pc1)$ だが、エージェントが $pc3$ から $pc2$ に移動するときには、 $goto(pc2)$ の取り消しアクションは $goto(pc3)$ である。我々は、[Hayashi 02b] で説明したとおり、Prolog のインタープリターを用いることにより、これらの問題を解決した。具体的には、新動的プランニングフレームワークでは、 $goto(X)$ のアクションデータは以下のように記述すればよい：

```
dataA(goto(X), undo(seq, goto(Y)))
:-location(Y).
```

ここで、アクションデータは、Prolog の文法で記述されていることに注意。 $location(Y)$ は、エージェントがいるコンピュータの場所を示すコンテキストである。エージェントは、移動するたびに、このコンテキストを更新する。例えば、現在エージェントがいるコンピュータが $pc1$ であるときには、コンテキストは $location(pc1)$ であるので、 $goto(X)$ のアクションデータは以下になる：

```
• dataA(goto(X), undo(seq, goto(pc1)))
```

このようにして、アクションデータの中身をアクション実行時のコンテキストに応じて変えることができる。なお、一般に、どのようなコンテキストを扱い、どのようにコンテキストを更新していくかについては、当論文の範囲外である。

6. プログラム更新とプラン修正

プログラム更新は、プログラムに新たに節を追加したり、あるいは、プログラムに属する節を削除したりすることにより行われる。このようなプログラム更新により、プラン導出の探索木に枝が追加されたり、逆に、枝が削除されたりする。従って、新しいプランが有効になったり、有効だったプランが無効になったりする。この章では、プログラム更新時のプラン修正方法を定義する。

【定義 6・1】動的プランニングフレームワークにおいて、プログラム更新後のプランゴール修正ルールは、以下の通りである。ただし、修正対象のプランゴールを、

```
(PLANS+, SSP+)
```

とする。

g1 プログラムに動的リテラル LIT を定義する節 C を挿入するならば、SSP⁺ に属する補助プランプラス

図 7 プランの探索木の修正

のうち、マークされたりテラルが LIT と同一化可能な全ての補助プランプラスを、

- $(PLAN_1, CLAUSES_1, HISTORY_1)$

⋮

- $(PLAN_n, CLAUSES_n, HISTORY_n)$

($n \geq 0$)とする。節 C によるプラン $PLAN_i$ ($1 \leq i \leq n$) の分解プラン (マークされた動的リテラルを選択) を $RPLAN_i$ とする。各プランプラス:

- $(RPLAN_1, \{C\} \cup CLAUSES_1, HISTORY_1)$

⋮

- $(RPLAN_n, \{C\} \cup CLAUSES_n, HISTORY_n)$

を $PLANS^+$ に挿入することにより、修正対象のプランゴールを修正する。

g2 プログラムから動的リテラルを定義する節 C を削除するならば、節 C が出現する全てのプランプラスを $PLANS^+$ と SSP^+ から削除することにより、修正対象のプランゴールを修正する。

プログラムに追加される可能性のある節は、動的リテラルを定義する。動的リテラルを選択してプランを分解するときには、導出ルール p2 (定義 3.4) に従い、そのプランを補助プランとして記録する。従って、その選択されたりテラルを定義する新しい節がプログラムに追加されても、プログラム更新後のプランゴール修正ルール g1 のように、その補助プランをその新しい節を使って分解することができる。これは、探索木の枝を追加することに相当する。図 7 では、新しく有効になった節 37 により補助プランとして記録されているプラン 3 を分解して新しくプラン 7 を作成している。

また、あるプランの導出に使われた節を削除すると、そのプランは無効になる。換言すれば、探索した探索木からある枝を枝打ちすると、その枝についている全てのプランが無効になる。図 7 では、削除された節 12 を使って作成されたプラン 2、プラン 4、プラン 5 が無効になる。プランプラスには、導出ルール p2 に従い、プランの作成に使った削除される可能性のある節を記録するので、プログラム更新後のプランゴール修正ルール g2 のように、無効になったプランを見つけて削除することができる。これは、探索木の枝を削除することに相当する。

7. エージェントライフサイクル

この章では、プラン作成、アクション実行、知識更新、プラン修正を統合するエージェントライフサイクルを導入する。エージェントは、リテラルを初期ゴールとして与えられ、それを満たす任意の個数のプランを作成し、実行を開始する。アクションを実行したり、プログラムを更新するときには、エージェントはプランを漸次的に修正する。エージェントは、保有しているプランの 1 つが空プランになり、その空プランに付随して記録されているアクション実行履歴が空であることを確認すると、初期ゴールが満たされたと判断し、エージェントライフサイクルを終了する。

アルゴリズム 7.1 動的プランニングフレームワークにおいて、初期ゴールがリテラル L のエージェントライフサイクルは、以下のようなアルゴリズムである。

- (1) 以下のように、初期プラン、初期プランプラス、初期プランゴールを設定し、最新プランゴールを初期プランゴールに設定する。

- $\{L\}$ を初期プラン $IPLAN$ とする。
- $(IPLAN, \{\}, \{\})$ を初期プランプラス $IPLAN^+$ とする。
- $(\{IPLAN^+\}, \{\})$ を初期プランゴールとする。

- (2) 空プランのフロンティアプランと空のアクション実行履歴が出現するフロンティアプランプラスが最新プランゴールに出現するまで、以下の手続きを繰り返す:

- a (プログラム更新) 以下の手続きを 0 回以上繰り返す:

- i. 動的リテラルを定義する節 C をプログラムに加える、または、プログラムから削除する。
- ii. プログラム更新後のプランゴール修正ルール (定義 6.1) に従って、最新プランゴールを修正する。

- b (アクション実行) 以下の手続きを 0 回以上繰り返す:

- i. もし可能ならば、最新プランゴールに出現するあるフロンティアプランが消費可能なアクション A を 1 つ選択し、実行する。
- ii. A の実行に成功したら、A 実行後のプランゴール修正ルール (定義 4.3) に従って、最新プランゴールを修正する。

- c (プラン分解) もし可能ならば、最新プランゴールからプランゴール GOAL までの導出を作成し、最新プランゴールを GOAL に更新する。

- d (アクション実行履歴) プラン分解後のプランゴール修正ルール (定義 4.5) に従い、最新プランゴールを修正する。

- e (プランの単純化) 可能な限り、以下のよう

なプランの単純化を繰り返す：

- i. 最新プランゴールに出現するプランに出現する空のサブプラン^{*3}を削除する．
- ii. 最新プランゴールに出現するプランに {E} または [E] の形のサブプラン (E はリテラルかプラン) が出現するならば、そのサブプラン ({E} または [E]) の括弧をはずし、E にする．

〔例 7・1〕 動的プランニングフレームワークにおいて、buy(X) と assemble(X) をアクションとし、good(X) を動物的リテラルとする．プログラムは以下の通りとする．

```
make(X) ← [getPartsFor(X), assemble(X)].
getPartsFor(X) ← {parts(X, Y), get(Y)}.
get([]).
get([H|T]) ← {good(H), buy(H), get(T)}.
parts(pc, [a, b]). parts(pc, [b, c]).
parts(pc, [c, a]).
good(a). good(b). good(c).
```

make(pc) を初期ゴールとすると、部品を買って PC を組み立てる以下の 3 つのプランが得られる．

```
[{buy(a), buy(b)}, assemble(pc)]
[{buy(b), buy(c)}, assemble(pc)]
[{buy(c), buy(a)}, assemble(pc)]
```

ここで、部品 a, b, c の買う順序が決まっていないことに注意．dataA(buy(X), undo(con, return(X))) が buy(X) のアクションデータであるとする．buy(a) を実行すると、プランは以下のように修正される．

```
[buy(b), assemble(pc)]
{return(a), [{buy(b), buy(c)}, assemble(pc)]}
[buy(c), assemble(pc)]
```

1 番目と 3 番目のプランからは実行済みのアクションである buy(a) が消去された．2 番目のプランで、新たに挿入されたアクション return(a) と残りのサブプランの実行順序は指定されていないことに注意．

次のアクションを実行する前に、部品 a が不良品であるという情報を得て、節 good(a) をプログラムから消去したとする．1 番目と 3 番目のプラン^{*4}は、消去した good(a) を前提節として記録しているので、消去する．その結果、2 番目の有効なプランのみが残る：

```
{return(a), [{buy(b), buy(c)}, assemble(pc)]}
```

このプランに従って buy(b) を実行すると、実行した buy(b) がプランから消去される：

```
{return(a), [buy(c), assemble(pc)]}
```

ここで、部品 a が不良品でなかったことにエージェントが気づき、いったん消去した good(a) を再びプログラムに加えたとする．すると、good(a) を無効であるとして消去したときに消した 2 つのプランが再び有効になり、これらが再生される：

```
[assemble(pc)]
{return(a), [buy(c), assemble(pc)]}
{return(b), [buy(c), assemble(pc)]}
```

なお、good(a) をプログラムから消去してから再びプログラムに加える間に、buy(b) を実行していることに注意．この buy(b) の実行結果は、再生された上の 2 つのプランに反映されている．これらのプランの導出には、補助プランと、それに付随して記録されているアクション実行履歴が用いられる．

8. プラン修正方法 (プログラム更新時) の効率性に関する実験評価

この章では、プランニング用知識を更新した後の 6 章のプラン修正方法と、最初から単純にプランを作成しなおす再プランニング方法とを、実験により比較する．プラン作成及び修正アルゴリズムは Prolog で実装した．使用した Prolog は SWI-Prolog (Version 5.0.3) である．

まず、実験に用いられるプランニング用知識を示す．アクションの宣言は以下の通りである：

```
action(pound(mochi)).
action(spread(_)).
```

ここでは、pound(mochi) と spread(_) の 2 つのアクションが宣言されているが、これらに同一化できるリテラルは全てアクションとみなされる (例えば、spread(_) と同一化できる spread(anko) や spread(kinako) はアクションである) 同様に、以下のように動物的リテラル (prefer(_)) の宣言をする：

```
dy(prefer(_)).
```

今回の実験では、アクションは実行しないので、アクションの副作用に関して記述する必要はない．プログラムは以下のように記述する：

```
axiom(seq, make(mochi, N), [
    repeatPound(N), giveTaste]).
axiom(con, repeatPound(0), []).
axiom(con, repeatPound(N), [
    pound(mochi), repeatPound(M)]):-
    N>0, M is N-1.
axiom(con, giveTaste, [
    prefer(X), spread(X)]).
axiom(con, prefer(anko), []).
```

^{*3} サブプランでない空プランは削除しない．

^{*4} 正確には、プランプラスが前提節を記録している．

上のプログラムの記述において、

- $\text{axiom}(\text{seq}, H, [B_1, \dots, B_n])$
- $\text{axiom}(\text{con}, H, [B_1, \dots, B_n])$

は、それぞれ、以下の節を表す：

- $H \leftarrow [B_1, \dots, B_n]$
- $H \leftarrow \{B_1, \dots, B_n\}$

これらの節は、Prolog で実装されたインタープリタで解釈されるため、 $\text{repeatPush}(N)$ の定義節のように、節が適用されるための条件（“ $N > 0$ ” と “ $M \text{ is } N-1$ ”）を記述することができる（同様に、取り消しアクションが適用される条件を記述できることは、5 章で既に紹介した。）

上のプログラムで、 $\text{make}(\text{mochi}, N)$ を初期ゴールとすると、「 N 回、餅をつき（ $\text{pound}(\text{mochi})$ ）、最後にあんこ味をつける（ $\text{spread}(\text{anko})$ ）」というプランができる。このプランを作った後に、 $\text{prefer}(\text{anko})$ をプログラムから削除して、 $\text{prefer}(\text{kinako})$ をプログラムから追加すると、6 章のプラン修正方法により、「 N 回、餅をつき（ $\text{pound}(\text{mochi})$ ）、最後にきなこをつける（ $\text{spread}(\text{kinako})$ ）」というプランに修正できる。プラン修正前にアクションは実行しないので、最初から単純にプランを作成しても同じプランを作成できる。このシナリオを用い「最初からプランを作り直す方法」と「6 章のプラン修正方法」の 2 つの方法で「プログラムを更新し、再プランニングするのに要する時間」（ランタイムで単位はミリセカンド）を測定し、比較した。

実験結果は、表 1 で示されるとおりである（実験は 3 回行い、その平均値を実験結果として載せた。）初期ゴール $\text{make}(\text{mochi}, N)$ の N を増加させると、最初からプランを作り直す方法では、再プランニングにかかる時間は単調に増加するのに対し、6 章の方法でプランを修正する時間はほとんど 0 ミリセカンドである。6 章の方法でプランを修正すると、餅を N 回つく部分のサブプランを再利用できる。 N の値が大きいほど、このサブプランを作るのに時間がかかるので、6 章のプラン修正方法は、最初からプランを作り直す方法に比べて、再プランニングの時間をより節約できることが想像できるが、この実験により、それが確認できた。従って、一般的に「6 章のプラン修正方法をつかうと、探索した探索空間のうち再利用できる部分が大きければ大きいほど、再プランニングの時間を節約できる」という直感的予測をこの実験は裏付ける。なお、実験で使われた例題のように、プランのごく一部だけの変更が必要となるケースは多い。例えば、6 章のプラン修正方法は DSLDNF という証明手続きを基礎としているが、DSLDNF[Hayashi 99, Hayashi 01] では、ロボットの移動経路の一部だけが変更となる場合で、再経路計算の時間を節約することに成功している。

表 1 プログラム更新と再プランニングに要する時間の各方法の比較

方法 \ N(回数)	200	400	600	800	1000
再度プラン作成 (msec)	97	327	688	1215	1790
プラン修正 (msec)	0	0	3	0	3

9. 意 味 論

プランニングアルゴリズムが定義された意味どおりに正しく動作すること（プランニングアルゴリズムの健全性）を示すことは重要なプランニングの研究課題である。代表的な HTN プラナーである SHOP のプランニングアルゴリズムに関しては健全性が証明されているが、我々のエージェントライフサイクルでは、プラン実行中に動的にプランが修正されるため、エージェントがエージェントサイクルを終えた後に、最新の知識（プログラム）に対して初期ゴールが満たされていること（エージェントライフサイクルの健全性）を示す必要がある。この章では、独立した意味論を定義し、その意味論に対してエージェントライフサイクルの健全性を示す。なお、SHOP では、存在するプランを必ず見つけること（プランニングアルゴリズムの完全性）が証明されているが、エージェントライフサイクルの完全性の研究は将来の我々の課題である。

以下の意味論では、 $\text{hold}(L, [A_1, \dots, A_n], \text{PLAN})$ という古典論理の述語を用いる。直感的には、この述語は、アクションを $A_1, \dots, A_n$ の順に実行した後にプラン PLAN を実行すると初期ゴール L が満たされるということを意味する。エージェントが一連のアクション： $A_1, \dots, A_n$ を実行して 1 つのプランの実行を無事に終了した後に、 $\text{hold}(L, [A_1, \dots, A_n], \emptyset)$ がいえれば、空プラン $\emptyset$ が示すとおり、エージェントがこれ以上何も行動しなくても初期ゴール L が成り立つことがわかり、エージェントの一連の行動 $A_1, \dots, A_n$ の正当性（つまり、健全性）を示せる。なお、以下の意味論で使われる $\leftarrow$ は、古典論理で通常使われる含意を表し、節で使われる $\Leftarrow$ とは異なる。

【定義 9.1】動的プランニングフレームワーク DPF において、 $\text{axiom}(\text{DPF})$ は以下のように定義される：

ax1 任意のリテラル L に対し、以下が成り立つとする：

- $\forall \text{hold}(L, [], \{L\})$
- $\forall \text{hold}(L, [], [L])$

ax2 L を任意のリテラルとし、 A と $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし、PLAN を任意のプランとする。PLAN が A を消費可能ならば、以下が成り立つとする：

- $\forall (\text{hold}(L, [A_1, \dots, A_n, A], \text{PLAN}^{\text{del}(A)}) \Leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ここで、 $\text{PLAN}^{\text{del}(A)}$ は、PLAN から消費可能な A の 1 つの出現を削除してできるプランである。

ax3 L を任意のリテラルとし、 A と $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし、PLAN を任意のプランとする。 A のアクションデータが $\text{dataA}(A, \text{noSideEffect})$

であるならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n, A], \text{PLAN}) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ax4 L を任意のリテラルとし, A と $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし, PLAN を任意のプランとする. A のアクションデータが $\text{dataA}(A, \text{undo}(\text{seq}, A^-))$ であるならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n, A], [A^-, \text{PLAN}]) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ax5 L を任意のリテラルとし, A と $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし, PLAN を任意のプランとする. A のアクションデータが $\text{dataA}(A, \text{undo}(\text{con}, A^-))$ であるならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n, A], \{A^-, \text{PLAN}\}) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ax6 プログラムの各節: $L \leftarrow \text{PLAN}$ に対し、以下が成り立つとする：

- $\forall \text{hold}(L, [], \text{PLAN})$

ax7 L と B を任意のリテラルとし, $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし, PLAN と SPLAN を任意のプランとする. PLAN に B が出現するならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n], \text{PLAN}^{\text{SPLAN}/B}) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}) \wedge \text{hold}(B, [], \text{SPLAN}))$

ここで, $\text{PLAN}^{\text{SPLAN}/B}$ は, SPLAN から, 1 つの B の出現を SPLAN に置き換えてできるプランである.

ax8 L を任意のリテラルとし, $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし, PLAN を任意の空でないプランとする. PLAN に空のサブプランが出現するならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n], \text{PLAN}^{\text{del}(\emptyset)}) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ここで, $\text{PLAN}^{\text{del}(\emptyset)}$ は, PLAN から 1 つの空のサブプラン $\emptyset$ の出現を削除してできるプランである.

ax9 L を任意のリテラルとし, $A_1, \dots, A_n (n \geq 0)$ を任意のアクションとし, E を任意のリテラルまたはプランとする. PLAN に {E} または [E] がサブプランとして出現するならば、以下が成り立つとする：

- $\forall(\text{hold}(L, [A_1, \dots, A_n], \text{PLAN}^{\text{nobracket}(E)}) \leftarrow \text{hold}(L, [A_1, \dots, A_n], \text{PLAN}))$

ここで, $\text{PLAN}^{\text{nobracket}(E)}$ は, PLAN から, サブプランとして出現する [E] または {E} の 1 つを選択し, その両端の括弧をはずしてできるプランである.

上の定義で, ax1 により, 初期プランの正当性を示せる. ax2 から ax5 により, アクション実行後のプラン修正方法の正当性を示せる. ax2 は, PLAN に従ってアクション A を実行した後の残りのプランは $\text{PLAN}^{\text{del}(A)}$ であることを示す. ax3 は, 副作用のないアクション A を実行しても, PLAN に影響がないことを示す. ax4 と ax5

は, アクション A を実行しても, A^- で取り消せることを意味する. ax4 では, A^- は PLAN を実行する前に実行しなければならない. ax5 では, A^- と PLAN の実行順序は問わない. ax6 はプログラムの節の意味を表す. ax7 により, リテラルをサブプランに分解するときの正当性を示せる. ax8 と ax9 により, プランの単純化 (余分な空のサブプランと括弧の削除) の正当性を示せる.

健全性の証明には, 以下の補題が用いられる：

[補題 9・1] 動的プランニングフレームワーク DPF において, L をエージェントに与えられた初期ゴールとし, エージェントライフサイクルの (1) (初期プランゴールの生成) を終了し, (2) の k サイクル目 ($k \geq 0$) を終了した時点で, エージェントはアクションを $A_1, \dots, A_n (n \geq 0)$ の順に実行していたとする (ただし, エージェントが実行したアクションは他にないとする.) また, 最新プランゴールに出現する各プラン PLAN に対し, PLAN と共にプランプラスに出現するアクション実行履歴に $A_{m+1}, \dots, A_n (0 \leq m \leq n)$ のアクションデータが記録されているとする. このとき, ある代入 θ が存在して, 以下が成り立つ：

(*1) $\text{axiom}(\text{DPF}) \models$

$$\forall(\theta(\text{hold}(L, [A_1, \dots, A_m], \text{PLAN})))$$

《証明》 k に関する帰納法による証明の概要を示す.

- $k = 0$ のとき, エージェントライフサイクルの (1) (初期プランゴールの生成) を終了しているので, 最新プランゴールに出現するプランは初期プラン {L} のみである. エージェントはアクションを実行していない. $\text{axiom}(\text{DPF})$ の ax1 により, 以下が成り立つ:

$$\circ \text{axiom}(\text{DPF}) \models \forall \text{hold}(L, [], \{L\})$$

故に $k = 0$ のときに補題 9・1 が成り立つ.

- k サイクルを終了した時点で, 補題 9・1 が成立していると仮定する.

○ $k + 1$ サイクル目でプログラムから節を削除したときには, その節を使ってできるプランは最新プランゴールから削除される. 従って, 最新プランゴールから削除されていない各プラン PLAN に関して, (*1) が成り立つ.

○ $k + 1$ サイクル目でプログラムに節 C を追加したときには, 新しいプラン PLAN が最新プランゴールに追加される場合がある. この場合, ある補助プラン SPLAN が最新プランゴールに存在し, SPLAN を C で分解することにより PLAN が作られる. 帰納法の仮定と $\text{axiom}(\text{DPF})$ の ax6 と ax7 により, 新しく作成されたこの PLAN に対し, (*1) が成り立つ.

○ $k + 1$ サイクル目でアクション A を実行したとき, 最新プランゴールの各プランプラスが修正の対象となる. プランプラスが修正されるときは, その内部のプランが修正されるか, アクション実行履歴に A のアクションデータが記録されるかのいずれかである.

もし、プランプラスの中のプランが PLAN に修正されたならば、PLAN と共に記録されているアクション実行履歴は空であり、帰納法の仮定と $axiom(DPF)$ の ax2 と ax3 と ax4 と ax5 により、ある代入 θ が存在して、以下が成り立つ：

$$*axiom(DPF) \models \forall(\theta(hold(L, [A_1, \dots, A_n, A], PLAN)))$$

従って、PLAN に関して、(*1) が成り立つ。

そうでなく、もし、プランプラスの中のアクション実行履歴に A のアクションデータが記録された場合は、同じプランプラス内のプラン PLAN は修正されていないので、帰納法の仮定から、PLAN に対して (*1) が成り立つ。

- $k+1$ サイクル目で最新プランゴールに出現するあるプランを分解したとき、新たに分解プランが作成される。帰納法の仮定と $axiom(DPF)$ の ax6 と ax7 により、新たに作成された各分解プラン PLAN に関して、(*1) が成り立つ。
- $k+1$ サイクル目で、アクション実行履歴に記録されているアクション A_{m+1} のアクションデータを用い、最新プランゴールのあるプランを修正し、アクション実行履歴から A_{m+1} のアクションデータを消去することがある。このような場合、修正されたプランを PLAN とすると、帰納法の仮定と $axiom(DPF)$ の ax2 と ax3 と ax4 と ax5 により、ある代入 θ が存在して、以下が成り立つ：

$$*axiom(DPF) \models \forall(\theta(hold(L, [A_1, \dots, A_m, A_{m+1}], PLAN)))$$

従って、PLAN に関して、(*1) が成り立つ。

- $k+1$ サイクル目で最新プランゴールに出現するあるプランを単純化（空のサブプランの除去と 1 つの要素からなるサブプラン $\{E\}$ または $\{E\}$ の括弧の除去）したとする。帰納法の仮定と $axiom(DPF)$ の ax8 と ax9 により、単純化された各分解プラン PLAN に関して、(*1) が成り立つ。

以上から、 $k+1$ サイクル終了時に補題 9・1 が成り立つ。 □

上の補題から以下の定理が成り立つ。

〔定理 9・1〕（エージェントライフサイクルの健全性）動的プラニングフレームワーク DPF において、 L をエージェントに与えられた初期ゴールとし、エージェントライフサイクルを終了した時点で、エージェントはアクションを $A_1, \dots, A_n$ ($n \geq 0$) の順に実行していたとする（ただし、エージェントが実行したアクションは他にないとする。）このとき、ある代入 θ が存在して、以下が成り立つ：

$$\bullet axiom(DPF) \models \forall(\theta(hold(L, [A_1, \dots, A_n], \emptyset)))$$

《証明》 エージェントライフサイクルを終了した時点で、最新プランゴールに、ある空プラン $\emptyset$ ($\{\}$ または $\{\}$) が存在し、この空プランと共にプランプラスに記録されているアクション実行履歴は空である。故に、補題 9・1

から、定理 9・1 が成立する。 □

10. お わ り に

我々は、Prolog や HTN プラナーのようなりテラル（タスク）分解をベースに「半順序プラン」の一種を使って、タスク詳細化（プラン作成）、アクション実行、プログラム更新、プラン修正を統合する新エージェントライフサイクルを提案した。また、独立した意味論を定義し、その意味論に対し、エージェントライフサイクル終了後に初期ゴールが達成されることを示した。

我々は、アクションの事前条件と事後条件（効果）に関する知識を用いずにプランを作成しているが、実行済みアクションの副作用を考慮してプラン修正を行っている。「はじめに」の章の例で示した通り、アクションの実行順序を全て決めてしまうと、プランの実行を中断して他の代替プランを実行するときに、あるアクション A を取り消すアクション A^- を実行して、その後、再び A を実行するという無駄な動きが生じることがあると指摘したが、部分的にリテラルの実行順序を規定しないプランを扱えるようにし、さらに、「いつ実行してもよい取り消しアクション」を導入し、より一層、必要のない取り消しアクションの実行を避けている。

プログラムを更新するときも我々はプラン修正を行う。節をプログラムから消去するとき、無効になるプランを消す。新しい節をプログラムに加えるとき、新たに有効になるプランを加える。これにより、最新のプログラムに対し有効なプランを保持することが可能である。このプラン修正方法は DSLDNF [Hayashi 99, Hayashi 01] に基づいているが、再プログラミングの際に、SLDNF（Prolog で使われている証明手続き）で始めからプランを作り直すよりは DSLDNF で保持しているプランを修正したほうが一般に高速であることが実験で確認されている。当論文においても、実験により同様の結果を確認できた。

今後の課題としては、「プランやアクションを選択する方法」に関する研究と、「同時に実行できるサブプラン間の干渉」に関する研究と、「エージェントライフサイクルの完全性」に関する研究と、「実用アプリケーション」の作成と、が挙げられる。

◇ 参 考 文 献 ◇

- [Bonner 93] A. J. Bonner and M. Kifer. Transaction Logic Programming. International Conference on Logic Programming, pp. 257-279, 1993.
- [BPEL4WS 03] BPEL4WS v1.1 Specification, 2003.
- [Cho 02] 長健太, 林久志, 大須賀昭彦. picoPlangent: 携帯機器向け知的移動エージェント. エージェント合同シンポジウム, 2002.
- [desJardins 99] M. E. desJardins, E. H. Durfee, C. L. Ortiz, Jr., and M. J. Wolverton. A Survey of Research in Distributed, Continual Planning. AI Magazine, pp. 12-22, Winter 1999.

- [Dix 03] J. Dix, H. Munoz-Avila, and D. Nau. IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment, *Annals of Mathematics and AI*, Vol. 4, No. 37, pp. 381-407, 2003.
- [Fukuta 00] N. Fukuta, T. Ito, and T. Shintani. MiLog: A Mobile Agent Framework for Implementing Intelligent Information Agents with Logic Programming. *Pacific Rim International Workshop on Intelligent Information Agents*, 2000.
- [Hayashi 99] H. Hayashi. Replanning in Robotics by Dynamic SLDF. *IJCAI Workshop "Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World"*, 1999.
- [Hayashi 01] H. Hayashi. Computing with Changing Logic Programs. PhD Thesis, Imperial College of Science, Technology and Medicine, University of London, 2001.
- [Hayashi 02a] H. Hayashi, K. Cho, A. Ohsuga. Speculative Computation and Action Execution in Multi-Agent Systems. *ICLP Workshop on Computational Logic and Multi-Agent Systems*, *Electronic Notes in Theoretical Computer Science*, Vol. 70, No. 5, 2002. 2002.
- [Hayashi 02b] H. Hayashi, K. Cho, and A. Ohsuga. Mobile Agents and Logic Programming. *IEEE International Conference on Mobile Agents*, pp. 32-46, 2002.
- [Kowalski 99] R. A. Kowalski and F. Sadri. From Logic Programming to Multi-Agent Systems. *Annals of Mathematics and Artificial Intelligence*, Vol. 25, pp. 391-419, 1999.
- [Nau 99] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. *International Joint Conference on Artificial Intelligence*, pp. 968-975, 1999.
- [Ohsuga 97] A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. PLANGENT: An Approach to Making Mobile Agents Intelligent. *IEEE Internet Computing*, Vol. 1, No. 4, pp. 50-57, 1997.
- [Russell 95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
- [Sato 02] K. Sato. Speculative Computation and Abduction for an Autonomous Agent, *International Workshop on Non-Monotonic Reasoning*, pp. 191-199, 2002.
- [Tarau 99] P. Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog, *International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1999.

〔担当委員：北村泰彦〕

2003 年 10 月 31 日 受理

著者紹介

林 久志 (正会員)

1995 年早稲田大学理工学部情報学科卒業。1996 年ロンドン大学インペリアルカレッジコンピューティング学科修士課程修了。1999 年ロンドン大学インペリアルカレッジ電気電子工学科博士課程修了。Ph.D.(ロンドン大学インペリアルカレッジ)。2000 年 (株) 東芝入社。現在同社研究開発センター知識メディアラボラトリー所属。主としてエージェント、プランニング、論理プログラミングの研究に従事。

長 健太

1995 早稲田大学理工学部情報学科卒業。1997 同大学院理工学研究科修士課程修了。同年 (株) 東芝入社。現在同社研究開発センター知識メディアラボラトリー所属。知的移動エージェントの研究に従事。情報処理学会、ソフトウェア科学会各会員。

大須賀 昭彦

1981 年上智大学理工学部数学科卒業。同年 (株) 東芝入社。1985～1989 年 (財) 新世代コンピュータ技術開発機構 (ICOT) に出向。現在 (株) 東芝研究開発センター知識メディアラボラトリー主任研究員。工学博士 (早稲田大学)。電気通信大学大学院客員教授ならびに大阪大学大学院非常勤講師兼任。主としてソフトウェアのためのフォーマルメソッド、エージェント技術の研究に従事。1986 年度情報処理学会論文賞受賞。情報処理学会、電子情報通信学会、日本ソフトウェア科学会、IEEE CS 各会員。