埋め込み型エージェントにおける行動とプランニング

大関

平成 16 年 3 月 23 日

埋め込みがタエージェントは、目標指向手段で、複雑で動的な状況を監視し、環境に影響を与えるエージェント周りの環境を知覚し、それに基づき行動するコンピュータシステムである。この論文は、エージェント設計のために、簡単に状況自動機械アプローチを復習し、状況自動機械の枠組みにおけるプランニングと行動の問題を探索する。

1　埋め込みエージェントの設計

埋め込みがタエージェントは、目標指向手段で、複雑で動的な状況を監視し、環境に影響を与えるエージェント周りの環境を知覚し、それに基づき行動するコンピュータシステムである。この種のシステムは、設計や構築が極端に難しく、明確な概念モデルや力のあるプログラミングツールがないため、実世界の複雑さは、簡単に圧倒的になってしまう。ある特別な場合では、設計は、古典的コントロール理論のようなよく知られている機械的なパラグラムに基づいている。しかし、より典型的に、このタイプのあつかいやすいモデルはあまり利用できないし、代替のアプローチが用いられなければならない。そのような代替の１つに、状況自動機械の枠組みであり、その枠組みは、質の点で埋め込みコントロールシステムと外部世界の間の関係をモデル化し、設計者のために抽象的概念をプログラミングする枠組みを提供する。この論文は、簡単に状況自動機械のアプローチを復習し、埋め込み型エージェントの行動部品の結成とよばれるアプローチのより詳細な一面について探索する。

1.1　状況自動機械モデル

状況自動機械アプローチの理論的基礎は、相互に作用するエージェントの組として世界をモデリングすることにある。その組の１つは、物理的環境に適応し、もう１つは埋め込みエージェントに該当する。それぞれは、もう一方から計画される信号の機能を変化させる局所的状態をもっている。設計処理の目的は、環境において望まれる結果を産み出す埋め込み状態機械の形式で、エージェントと同調することである。

興味の応用において、エージェントが辿っている環境と目標について利用できる情報において、エージェントを表記することはしばしば有用である。これらの表記が、すくなくとも設計の早い段階の間、明確な内部データ構造よりむしろ環境の状態に言及する言語によって表現されることも望まれている。加えて、状態機械の入力と出力、内部状態があまりにも多数あるので明示的に考えられないでしょう。これは、機械が振舞の複雑な型を発生させるためにたがいに作用する分岐部品の集合の外で構築されなければならないということを意味している。これらの要求は、意味のある単語で機械部品を簡潔に表記する、高レベルな耕成の言語の必要性を示している。

状況自動機械理論は、エージェントのデータを解釈する原理的な方法を、意味が設計者にとって明確である言語によって表現される世界についての事実をエンコードするものとして供給する。この種の解釈は表示された状態が、その環境で保持されることが保証されていて、データ好悪図が特定の値をもっている場合でないならあまり有用ではない。そのような考えが、外部現実における物体の関係からデータ構造の意味を定義しようとするきっかけを与えた。このアプローチにおいて x がもっている全ての世界の状態について、同じ値が s の中にあり、命題 p が真であるならば、$s \models K(x,p)$ とかき、機械変数 x は世界状態 s の中の p という状態を与えるという。このモデルの以前の特長と埋め込みシステムをプログラミングする有用性は

論文9、10、5、11のどこにでも表記されている。

エージェントにおいて、与えられた信号や状態を、環境に置ける物の関係による情報の中味を運んで来るものと物と見なす為の理論の基礎が設立されて、ひとは、この中味が表現されるかも知れない言語を考えることができる。一般的に、この情報を表現するための「最もよい」たった１つの言語なんてないでしょう。これらは、意味の解釈が関係する状態である歌のシステムと見なされうる。しかし、設計者は典型的に設計処理中に他の高レベルな言語を使うことを望む。このテーマは、目標表記言語との接続をこえて拡張されるでしょう。

1.2　認知と行動の分離

設計者の認識の容易さのための設計処理の構築方法の１つは、世界についての情報を獲得する問題とその情報に関連して適切に行動するという問題を分離することです。前者をわたしたちは認識と名付け、後者を行動となづけた。状態機械モデルにおいて図１に見れるように認識部分は更新巻数と初期状態に一致する。ところが一方、行動部分は出力写像に一致する。

認識ー行動分離は全体的に分りやすく、行動的システムをモデリングするための基礎となるかも知れないしならないかも知れない。認識と行動を駈け抜ける水平的分離は慣例的にブルックスが行なったエージェント設計アプローチが挙げられる。水平的なアプローチのおかげで設計者は明確な振舞を支援するために必要とされる認識と行動の限られた一面を同時に考えることができる。この方法で、それは、ロボットに置いて慣習的な処理をしばしば抑制する偽の一般性の追従をゆるさない。

しかし、これらの魅力的な特長は水平的分離が線形思考をはげます程度によってトレードオフの関係になる。慣習的に、情報の獲得を使用から分離しないという方法論は、複雑な振舞の型を自由に産み出す為に再結合することができる要素の特定というよりむしろとても明確な振舞の発展を支援しがちである。代替のものは、広く役に立つ情報をもとにもどし、多様資源とそのほかの多様目的のために開発するものを分離したシステムモジュールをもつことを基本とする垂直の戦略である。情報搾取と振舞生成の生まれながらの組み合わせは、プログラマーの努力を十分に活かす方法として垂直アプローチを魅力的にする。

認識と行動をもとにした分離への委任はまだ、発達戦略の疑問をのこしたままである。１つのアプローチは多かれすくなかれ停止ステップで認識ー行動の組を反射的にリセットすることである。入力信号や内部状態によって客観的に実行される情報は、行動部分の制約も含む、システムの他の部分についての制約に関連する。システムの残りの部分がより制約化されるにつれて、ますます設計者は世界について内部信号や状態を減らすことが出来るし、システムが含んでいる「情報」をより磨くことが出来る。設計者が設計を洗練するにつれて、システムが利用できる情報のモデルと、システムの反応がどうなるのかといったことがどんどん明確になる。

たくさんの慣習的な設計状況に合う、反射的リセットにたいする代替案は、認識部分の設計が共通のインタフェースをのぞいて、行動部分の発展から完全に独立したところで実行されるところである、堅実な分割結合戦略である。情報を記号化するデータ構造は認識と行動のモジュールの間に共有される。ある信号や状態が、設計者が意図した意味の内容をもっているということを保証するためにエージェントがすることについてのいくつかの事実に設計者が頼る必要があるときに機会があるかも知れないが、もしこれらの状況が小さくされたり無視されたりしうるのならば、考えられる単純な結果が生まれるでしょう。

1.3　目標

私たちが見て来たように、エージェントの状態を意味的に性格づける１つの方法は、状態が具体化する情報である。認識部分が情報を運んできて、行動部分がこの情報を行動にマッピングする。しかし、多くの場合、行動を情報の関数としてだけでなく、エージェントがその瞬間に追跡する目標の関数として表記することがより自然である。

目標は二つの広いクラスに分けれる。それは静的なものと動的なものである。静的目標は、エージェントの振舞が真になるように単純に設計されるという記述である。現実に置いて、静的目標は、設計者がエージェントの行動戦略の概念を組織するのを手助けするのに実際用いられているけど、エージェントがこの目標に帰属することはある程

度余分であるような仕様書でしか無い。動的目標は、もう１つのものである。実行中に動的に変化するエージェントの目標に帰属する能力は、設計者がエージェントの振舞の表記することを劇的に単純化することが可能になる。

私たちはリアクティブシステムで情報ベースの意味に見をゆだねているので、私たちは情報の単語で明示的に定義された「客観的な」目標の意味を探す。私たちは p が「解脱」からとった N とよばれる固定されたトップレベル目標を暗示するという情報をもっているものとして目標 p をもっているという概念を再定式化する。形式的に、私たちは目標オペレータ G を以下のように定義する。

$$G(x,p) \equiv K(x, p \to N)$$

このモデルに置いて、x が p が解脱を暗示しているという情報を持って来たら x は目標 p をもっている。この定義は動的目標の概念をあつめる。何故ならば p は「今雨が降っている」といったような指示的な文であり、その信頼は時間によって異なるからである。そのモデルは、情報によって明確に目標を定義するので、情報を学ぶために使われる同じ堅実なツールは、同様に目標に適応され得る。実際、この定義のもと目標と情報は二重の概念である。

目標と情報の二重性を見るために、１つの変数の値 a を違う変数の値 b にマッピングする関数 f を考えて下さい。情報解釈のもとでは、そのような関数は、より明確な情報をもつ要素をより不明瞭な情報をもつ要素にする。これは、一般的に異なる入力を同じ出力にマッピングすることによる不明瞭さを導く。例えば、もし値が u_1 である変数 a が命題 p と関係し、もし値が u_2 である変数 b が命題 q と関係し、f が u_1, u_2 ともに値が v である変数 b にマッピングするならば、値 v は u_1, u_2 のどちらから起こったのか分らないし、それゆえにそれが含んでいる情報が、u_1 か u_2 のどちらかに含まれていた情報が、分けられない情報 $p \vee q$ になる。このように、関数的マッピングは忘却というものがある。

目標の解釈のもと、この画像は逆になる。「忘れる」という類似した事象は、含目標に身をゆだねていて、それは、状態を達成する他の方法があるということを「忘れる」というように考えられる。例えば、変数 a の個々の情報が、エージェントがお腹がすいているということと、右側にサンドイッチがあるということと、左側にリンゴがあるということであるとする。もし多対一の関数のアプリケーションが変数 b がもっている値とエージェントがお腹がすいているということが矛盾しないで、サンドイッチが右にあり、リンゴが左か右のどちらかにあることしかわからないのなら、私たちは、変数 b が左側で食べ物を見つける道を失ったということによって仕事の状態を表記することが出来る。言いかえれば、私たちは、変数 b は右の方にいくという副目標に身をゆだねるということをいえる。忘れるということと身を委ねるという現象は同じコインの表と裏のようなものである。

私たちは、この観察を情報と目標を表記する原則と関連づけることができる。K によって満たされる形式的な特長の１つは、演繹的な閉包原則であり、以下のようにかかれる。

$$K(x, p \to q) \to (K(x,p) \to K(x,q))$$

目標の類似した原則は

$$K(x, p \to q) \to (G(x,q) \to G(x,p))$$

となる。これは、正確に副目標の原則である。もしエージェントが q を目標としてもっていて、q が他のものに含まれるという情報を持って来たら、エージェントは p を目標として適応するようになる。この原則の正しさの証明は G の定義から直接的になされる。

これらのデータ構造の意味を見る二つの方法が与えられたとき、私たちは、上記で紹介したエージェントの状態機械モデルを再確認する。１つのパラメータをもつ純粋な「情報」単語で解釈する関数 f(i) として、機械の行動部分を明確化するよりむしろ、設計者が、二つのパラメータをもち、g がエージェントの動的目標を表現するものとして解釈されるでそれを定義する f'(g,i) のほうが便利である。g はどこから入力されるのだろうか？それは、はっきりと、エージェントの静的目標 g_0 と同様にエージェントの現在の情報の状態から究極的に計算されなければならない。そのようにして、それは、目標依存でないいくつかの仕様と整合性が取れなければならない。つまり f(i)=f'(extract(i,g_0),i) となる。にもかかわらず、目標抽出モジュールと目標指向行動モジュールの分解は、設計者に設計が意味的にグラウンドされたという知識のおいて、安

全に保つために、設計者が作製する認識器の負荷を大々的に軽減できる。

1.4 エージェント設計のためのソフトウェアツール

埋め込み型エージェントのデータ構造の意味の形式的な理解をもつことは、概念的に重要であるけれども、この理解は直接的にプログラマーの仕事を簡単化しない。このため、特定の基礎に基づき、埋め込み型エージェントのプログラミングをより簡単にするソフトウェアツールの設計と実装は、必要である。

レックスはプログラマーが同調するデジタルの回路を明確化するためのコンパイル時にリスプの再帰機能をフルに使うことが出来るようにした言語である。計算の回路モデルは、状況自動機械理論の枠組みにおいて、意味的な分析を容易にする。しかし、レックスは今までの AI 言語よりもより類似した基本的プログラミング言語である低レベル処理言語しか提供しない。このため、私たちは、レックスによって供給された基礎を頭に、宣言されているプログラミング言語の組を設計し、実装した。ルーラーは「情報的な」意味に基づいており、エージェントの認識部分を明確化する為に使われるように意図されている。ギャップスは、「目標」意味に基づいており、エージェントの行動部分を明確化する為に使われるように意図されている。この論文の残りでは、私たちはギャップス言語について述べ、埋め込み型エージェントをプログラミングするのに有用であることと、プランニングのより古典的な仕事に関連したたくさんお拡張を述べる。

2 ギャップス

この章で、私たちはギャップスについて述べる。ギャップスは、コンピュータエージェントの振舞を明確化するための言語であり、宣言明確化の利点を保持するが、反射的で、並列行動をし、とても低レベルな行動を行なう戦略を実行する実時間型プログラムを作製する。

ギャップスはエージェントの行動部分明確にするために使われるという意図がある。ギャップスコンパイラはエージェントのトップレベル目標と目標遠ざかりルールの集合の宣言的明確化を入力としてとり、それらを認識部分の出力と、その出力全体としてのエージェントの出力をもつ回路の表記に変換する。エージェントの出力はたくさんの分岐したコントロールの行動に分割されるかもしれない。そのため、私たちは、エージェントが同時に移動したり話したりすることが出来るようにする手段を独立して明確化することが出来る。サンプル行動配列の宣言は以下の通り。

$$(declare - action - vector$$
$$(left - wheel - velocityint)$$
$$(right - wheel - velocityint)$$
$$(speechstring))$$

これはエージェントが三つの独立したコントロール要因を持ち、それ等をコントロールする出力値の型を宣言するということを述べる。

以下の節では、私たちはギャップスの今までの目標評価アルゴリズムと表現を示し、ギャップスの明確化が回路表記としてどのように実証されるのかを説明する。

2.1 目標とプログラム

ギャップスコンパイラはトップレベルの目標と目標遠ざけルールの集合をプログラムへ写像する。この節で、私たちは、目標と、目標引き下げルールとプログラムの概念を明らかにする。

三つの原始的な目標の型がある。それは、実行と、達成、維持の目標である。実行の目標は、形式的に、do(a) となり、a は実世界でエージェントによってとられうる瞬間的な行動を明確化するものである（エージェントの目標は単純にその行動を行なうこと)。エージェントが維持の目標をもっているなら、maint(p) と書き、もし命題 p が真であるならばエージェントは出来る限り p が真となり続けるようにする努力をするべきである。目標 ach(p) は達成の目標であり、その目標のためにエージェントは、出来る限り早く命題 p の真偽について議論できるようにするべきである。目標の集合は、ブーリアン処理で閉じられた原始的な目標の型で構成される。達成と維持の概念は、二重である。したがって、私たちは $\neg ach(p) \equiv maint(\neg p)$ と $\neg maint(p) \equiv ach(\neg p)$ をもっている。

目標を明確化する目標の点でプログラムの正しさを性格づけるために、私たちは、目標を導く行動の概念をもたなければならない。非形式的であるが、目標を満足させる正しいステップがあるなら、行動 a が目標 G を導くことを $(notateda \rightsquigarrow G)$ と表す。達成という目標のために、行動は、目標の状態が最終的に、真になる構成でなければならない。すなわち、機械の目標にとって、状態がもし、すでに真であるならば、行動は、すぐ次のステップで真になるということを暗に意味しなければならない。オペレータへの導きも以下の形式的な命題に従わなければならない。

$$a \rightsquigarrow do(a)$$
$$(a \rightsquigarrow G) \wedge (a \rightsquigarrow G') \Rightarrow a \rightsquigarrow (G \wedge G')$$
$$(a \rightsquigarrow G) \vee (a \rightsquigarrow G') \Rightarrow a \rightsquigarrow (G \vee G')$$
$$cond(p, a \rightsquigarrow G, a \rightsquigarrow G') \Rightarrow a \rightsquigarrow cond(p, G, G')$$
$$(a \rightsquigarrow G) \wedge (G \rightarrow G') \Rightarrow G'$$

この定義は、行動が目標を導くということは何を意味するのかという弱い直観を含んでいる。行動をすることの目標は行動をすることによって即刻に満足されなければならない。もし行動がそれぞれの二つの目標を導くなら、それは、二つの接続を導く。同時に、切断と状態を導くかも知れない。達成という目標を導くということの定義は、行動を行うということは、目標を達成することを構成し、行動は目標状態に対する過程を構成するといいたいと、いうことよりとても弱く見えるかも知れない。不幸にも、領域独立法でこの概念を形式化することは難しい。実際、この定義を満足する導出のどんな定義でも、ギャップスの目標引き下げアルゴリズムで構成されるから、その定義は、特定の領域に置いて強化されるかもしれない。

目標引き下げルールは (defgoalr G G') というように形式的に表し、目標 G が目標 G' に引き下げられるということを表す。すなわちその G' は G に特化されていてそれゆえに、G を含んでいる。「導出」の定義によって、G' を導くどんな行動も G を導くでしょう。

プログラムは、状態/行動ペアの有限集合である。プログラムの中で状態は実行表現であり、行動は実行表現の配列であり、それぞれの原子出力領域と一致する。これらの行動は、認識された入力から出力値への実行時のマッピングであり、戦略と見ることができる。その中に置いて、生成された特定の出力はエージェントの内部状態を経由した世界の外部状態に依存する。行動が全ての戦略になることを許すということは、とても柔軟であるが、出力領域の可能な値を数えあげることを不可能にする。行動配列の発話領域のみを支配するプログラムを明確化するために私達は、発話領域を要求しているプログラムに他の領域を制約しないある値を持たすことができなければならない。これを行う一つの方法は、発話の値の明確化とともに、行動配列の集合を数えることでしょう。また、それぞれは、他の行動配列部分と異なる値を持っています。これを行う代わりに、私達は、行動配列の要素が値 $\emptyset$ を含むことを許すがそれは、その領域の全ての可能な例示となる。

状態行動ペア $\{< c_1, a_1 >, \ldots, < c_n, a_n >\}$ を含む、プログラム Π は、もし全ての状態 c_i が真であり、一致する行動 a_i が G を導くなら、目標 G を弱く満足させるという。すなわち

$$\Pi weaklysatisfiesG \Leftrightarrow \forall i.c_i \rightarrow (a_i \rightsquigarrow G)$$

となる。プログラム中の状態が使い果たされる必要が無いということを注意してもらいたい。満足させるということは、一般的に不可能であるので全ての状況で、目標を導く行動があるということを求めるというものではない。私達は、プログラムが、自分の領域として行動を明確化する状況のクラスを言及する。私達は、プログラム Π の領域を次のように定義する。

$$dom(\Pi) = \bigvee_i c_i$$

目標 G はプログラム Pi に弱く満足されていて、$dom(\Pi) = true$ ならば、プログラム Π によって強く満足されているという。すなわち、全ての状況で、Π が G を導く行動を供給する。プログラム中の状態は、相互に締め出される必要は無い。プログラムの状態が一つ以上真であるとき、状態それぞれに関連する行動は目標を導き、プログラムの実行は、これらの行動の中で非決定的に選ぶかも知れない。

非決定的実行モデルが与えられたとき、私達は、プログラムに宣言の意味も与えることができる。プログラム $\Pi = \{< c_1, a_1 >, \ldots, < c_n, a_n >\}$ が

論理的解釈を持っていると考えられるとき、

$$(\bigwedge_i (a_i \rightarrow c_i) \wedge \bigvee_i a_i) \vee \neg \bigvee_i c_i$$

となる。プログラムのそれぞれの領域は、偽であるか、実行される行動とその行動が真であることに関連した状態があることである。

2.2 再帰的目標評価過程

ギャップスはレックスの上位言語として実装されていて、認識試験を与えるレックス言語の制約を利用する。レックス言語について述べる余白がここには無いので、他の論文の面白い読者を述べよう。ギャップスプログラムは、目標引き下げルールとトップレベルの目標表現の集合から作成される。目標引き下げルールの一般的な形式は、以下のように表される。

$$\begin{aligned}
&(detgoalrgoal - patgoal - expr),\\
&where\\
goal - pat ::= &(achpatrex - params)\\
&(maintpatrex - params)\\
goal - expr ::= &(doindexrex - expr)\\
&(andgoal - exprgoal - expr)\\
&(orgoal - exprgoal - expr)\\
&(notgoal - expr)\\
&(ifrex - exprgoal - exprgoal - expr)\\
&(archpatrex - expr)\\
&(maintpatrex - expr)
\end{aligned}$$

index はキーワードであり、*pat* は統一可能な変数を持つコンパイル時間の型であり、*rex-expr* は変数入力の実行中関数を明確化するレックス表現である。そして、*rex-params* は、*rex-expr* の結果に束縛される変数の構造である。これらの制約の詳細は、以下の節で述べる。

ギャップスコンパイラは、プログラマーが与えた目標引き下げルールを使って、目標をプログラムに写像する評価関数の実装である。この節で、私達は、それが正しいということを示す評価の過程を示しましょう。すなわち、目標 G と引き下げルールの集合 Γ が与えられたとき、$eval(G, \Gamma)$ は弱く G を満たす。

引き下げルールの集合 $Gamma$ が与えられたとき、私達は、評価過程を以下のように定義する。

do

$and : co$

$or :$

$if : disjoin - programs(conjoin - cond(second(G), eval(third(G))),$

私達は今、交互にこれらの場合を見るとしよう。

Do

関数 *make-primitive-program* は、インデックスとレックス表現をもってきて、プログラムを返す。インデックスは、行動配列のどの領域が割り当てられているかを指し示し、レックス表現は入力から行動領域の値への関数を表す。それは、形式的に以下のように定義される。

$$make - primitive - program(i, rex - expr) = \{< true, < \emptyset, \ldots, rex -$$

行動選択の i 番目の部分に *rex-expr* がある。このプログラムは、行動の部品 i が *rex-expr* によって表現される戦略であるかぎり、どんな行動も許す。

And

プログラムは、プログラムの状態/行動ペアの交差生産を行うことと、交差生産のそれぞれの要素を、一緒にマージすることによって、接続される。二つのプログラムを接続するとき、マージされた行動配列は二つの行動がマージ可能である、状態をともに持つオリジナルペアの状態の接続と関連づけられる。接続過程は単純に状態の接続と行動を共有するそれぞれのプログラムにおけるペアを発見する。私達は、処理を形式的に以下のように

定義する。

$$conjoin-programs(\Pi',\Pi'')$$
$$=\{<(c'_i \wedge c''_j \wedge mergeable(a'_i,a''_j)), merge(a'_i,a''_j)>\}$$
$$for 1 \geq i \geq m, 1 \geq j \geq nwhere$$
$$\Pi' = \{<c'_1,a'_1>,\ldots,<c'_m,a'_m>\}$$
$$\Pi'' = \{<c''_1,a''_1>,\ldots,<c''_n,a''_n>\}$$

接続処理は、プログラムの宣言された意味を保存する。すなわち、接続されたプログラムの意味的解釈はここのプログラムの意味的解釈の接続によって意味される。

二つの行動配列は、それぞれが部品であって、少なくとも一つが不明瞭か、両方が等しいとき、$mergeable$という。

$$mergeable(<a_1,\ldots,a_n>,<b_1,\ldots,b_n>)$$
$$\equiv \forall i.(a_i=\emptyset)\vee(b_i=\emptyset)\vee(a_i=b_i)$$

もしそれぞれの部品が不明瞭だったら、テストはコンパイル時に完了され、増加する回路が生成されない。そうでなければ、実効時にテストされる状態とともに等式テストが結合される。

行動配列は、もし一つが利用可能であるなら、定義された要素をとって、部品レベルでマージされる。もし、配列が等しくなく部品で定義されていたら、結果は、未定義となる。

$$merge(<a_1,\ldots,a_n>,<b_1,\ldots,b_n>)$$
$$=<c_1,\ldots,c_n>, where$$
$$c_i = a_i if b_i=\emptyset or b_i=a_i,$$
$$b_i if a_i=\emptyset,$$
$$\perp otherwise$$

二つの行動配列をマージするものは、オリジナルのものに許可される行動の交差を許す行動配列を結果とする。

Or

二つのプログラムの切断は単純にプログラムの状態/行動ペアの集合の結合である。形式的に以下のようになる。

$$disjoin-programs(\Pi',\Pi'')=\Pi'\cup\Pi''$$

Not

ギャップスにおいて、否定は、*not*を含む表現が*(ach(not pat)), (maint(not —itpat)),(not(do index rex-expr))*だけの形式になるまで、可能な限りドモルガンの法則と ach と $maint$ の双対性を用いて表現される。最初の二つの場合、目標の明示的な引き下げルールがなければならないし、最後の場合は、私達は、単純に空のプログラムを返す。否定の扱いは、もし、私達が、その行動配列部品の数を数えることをしたり、コンパイル時に制約を知る必要があるなら、より強く扱われなければならない。このとき、*(not(do left-velocity 6))*は$\vee_{i\neq 6}$*make-primitive-program(left-velocity,i)*と同じになる。すなわち、*6*以外のどんな速さで行くことも許可される。私達が以前に述べたように、これらの限界は、可能な出力をたくさんもつ複雑なエージェントをコントロールするときにはとても厳しいものとなる。

処理*negate-goal-expression*は目標表現を以下のように再表記する。

$$(not(andG_1G_2)) \Rightarrow (or(notG_1)(notG_2))$$
$$(not(orG_1G_2)) \Rightarrow (and(notG_1)(notG_2))$$
$$(not(notG)) \Rightarrow G$$
$$(not(ifcG_1G_2)) \Rightarrow (ifc(notG_1)(notG_2))$$
$$(not(achp)) \Rightarrow (maint(notp))$$
$$(not(maintp)) \Rightarrow (ach(notp))$$

If

状態のプログラムの評価処理は状態オペレータ*cond(p,q,r)*を$(p\wedge q)\vee(\neg p\wedge r)$として定義することを決めつける。状態とプログラムを結合する処理は、以下のように定義される。

$$conjoin-cond(p,\Pi)$$
$$=\{<p\wedge c_1,a_1>,\ldots,<p\wedge c_n,a_n>\}$$

そして、

$$disjoin-programs(conjoin-cond(p,\Pi'),conjoin-cond(\neg$$
$$=\{<p\wedge c'_1,a'_1>,\ldots,<p\wedge c'_n,a'_n>,<\neg p\wedge c''_1,a''_1>,\ldots,<\neg p\wedge c''_m,$$

Ach と *Maint*

維持と達成の目標はルールベース Γ において、全ての適応的な引き下げルールの結果を切断することによって評価される。その先頭が表現$(achpat_1 rex-params)$である引き下げルールは、もし pat_1 と pat_2 が現在束縛されている環境において統一されるなら、目標表現$(achpat_2 rex-expr)$と合う。そのパターンは?を導くことによってマー

クされるコンパイル時の変数を伴う *s-expressions* である。レックスの表現とパラメータは $NULL$ であったら従わされるかも知れない。束縛する環境は、目標表現が評価されてるうちにコンパイル時の変数の他の束縛を構成する。このように、目標 *(and(ach(drive ?q?p))(ach(go ?p)))* の副目標 *(ach(go ?p))* を評価するとき、私達は、すでに *?* への束縛を持つかも知れない。序論で見たように、この目標の評価は $?p$ と $?q$ の全ての可能な束縛を通して後をたどられる。

一旦、パターンがマッチしたら、ギャップスはルールの具体を評価するために環境を束縛する新しいコンパイル時間を設定する。これは、具体的変数が発動によって束縛される場合に必要であり、以下のようになる。

$$8defgoalr(ach(at?p)[dist-errangle-err])$$
$$(if(not-facing?pangle-err)$$
$$(ach(facing?p)angle-err)$$
$$(ach(moved-toward?p)dist-err))$$

上記のルールで、*(at ?p)* はパターンであり、*?p* はコンパイル時のパラメータであり、*dist-err* と *angle-err* はレックス変数であり、いったん束縛が *?p* で代替されたら、*(not-facing ?p angl-err)* はレックス表現になるでしょう。このルールの発動は次のようになる。

$$(ach(at(office-ofstan))[*distance-eps*10])$$

ギャップスは、ルールが壊れたとき、環境を束縛する新しいレックス変数を生み出すこともする。そして、発動時に評価されていたレックス表現の先頭にあるレックス変数を束縛する。これらは、ルールの本体のレックス表現に現れるかも知れない。コンパイル時変数も、利用可能な実効関数のクラスの中からコンパイル時に選ぶためにレックス表現に用いられるかも知れない。

2.3 回路の生成

いったん目標表現が評価されてしまったら、プログラムが生成されたということを例示する、図 *2* に示すのと類似した回路のプログラムに負ける。関連する行動が真であるような行動はどんなものでも、十分正しいから、状態は、コンパイル時に選ばれた調停された順序においてテストされる。回路の出力は、真である最初の状態と一致する行動である。もしどの状態も満足されなかったら、プログラマーがエラーとした信号を示すエラー行動がとられる。もし回路生成の最終段階に置いて行動配列にまだ $\emptyset$ 部品があるならば、部品は調停値とともに例示されなければならない。回路の入力は、*if* と *do* 形式で与えられたレックス表現によって計算される。循環の出力はエージェントをコントロールするのに用いられる。

2.4 引き下げられた接続的目標表現

接続的な目標表現は二つの形式を持っている。*(ach-or-maint(and p_1p_2))* と *(and(ach-or-maint p_1)(ach-or-maint p_2))* の二つである。維持の特性のため目標 *(maint(and(p_1p_2)))* と *(and(maint p_1)(mnaintp$_2$))* は意味的に等しい。しかしこれは達成の目標として正しくない。目標 *(achach(and p_1p_2))* は p_1 と p_2 が同時に真になることを求めるが、一方目標 $(and(achG_1)(achG_2))$ は未来のある時点でどちらかが真であることだけを求める。

形式 *(ach-or-maint(andp_1p_2))* の目標はこの接続パターンにマッチするパターンを持つ引き下げルールを使ってのみ引き下げられる。形式 *(and(ach-or-maint p_1)(ach-or-maint p_2))* の目標は二つの方法で引き落とされる。接続的な目標の基本的な評価過程を用いる方法と、特別な引き落としルールを用いる方法です。G_1 と G_2 を達成する効果的な行動が G_1 と G_2 をここに達成するというプログラムを結合することで単純に生成できない場合もしばしばある。目標 *(and(ach have hammer)(ach have saw))* のためのプログラムは、二つのツールが異なる余白にあるとき、ほとんど確実に終ることはできないでしょう。なぜならば、それぞれの副目標を達成する基本的なプログラムを構成する利用可能な行動が無いからである。このため、私達は、特別な振舞が結合的目標の直面で生成されるように、$(defgoalr(and(ach-or-maintpat_1rex-params_1)(ach-or-maintpat_2rex-params_2))goal-expr)$ という形式の引き下げルールを許す。

以下は、結合的目標の両方の種類を例示した例である。トップレベルに置いて、目標はハンマーを持つと同時に見ることであるが、これは、*ach and*

maint 目標の結合に引き落とすことができる。

$$(defgoalr(ach(and(haehammer)(havesaw))$$
$$(if(havehammer))(and(mainthavehammer)$$
$$(achhavesaw))$$
$$(if(havesaw)$$
$$(and(mainthavesaw)$$
$$(achhavehammer))$$
$$(if(closer-thanhammersaw)$$
$$(achhavehammer)$$
$$(achhavesaw))))))$$

エージェントは彼が物を持つまで近くのものを追いかける。そして、最初のものを保持している間次のものを追いかける。私達は、達成と維持の目標の結合を引き落とす類似のルールが必要かも知れない。上記の明確なルールの代わりに、私達は、以下のようなより一般的な一連のルールを書くことができる。

$$(defgoalr(ach(and?g1?g2)$$
$$[g1-paramsg2-params])$$
$$(if(holds?g1g1-params)$$
$$(ach?g2g2-params))$$
$$(if(holds?g2g2-params)$$
$$(and(maint?g2g2-params)$$
$$(ach?g1ga1-params))$$
$$(if(better-to-pursue?g1g1-params$$
$$?g2g2-params)$$
$$(ach?g1g1-params)$$
$$(ach?g2g2-params))))$$

ルールの一般的な形式はコンパイル時パラメータをとり、コンパイル時パラメータと実効時変数によって記号化される予測が世界で真になるかどうかをみることをテストする回路を生成する *holds* というレックス関数があることを仮定している。

2.5 優先順位をつけた目標リスト

目標の優先順位をつけたリストを明確化できることはしばしば便利である。ギャップスにおいて、私達は、$(priogoal-expr_1 \ldots goal-expr_n)$ 形式の目標表現と一緒にこれを行うことができる。この意味は、次のようになる。

$$cond(dom(\Pi_i),\Pi_1,$$
$$cond(dom(\Pi_2),\Pi_2,\ldots,$$
$$cond(dom(\Pi_{n-1},\Pi_{n-1},\Pi_n)\ldots)))$$

ただし、$\Pi_i = eval(goal-expr_i)$ である。プログラムの領域は、プログラムの状態の切断である。*prio* 目標のプログラムは、それが、適切な行動を持っていない限り、一番最初のプログラムを実行し、その後、二つ目、三つ目と行って行く。回路生成時には、この制約が優先順位のついた順序でプログラムを連接することによって、単純に実装することができる。そして、状態が満足されたものと一致する初めの行動を実行する。

prio 制約の利用の例が、特定を目標を達成する一つ以上の手段があって一つは何らかの理由二より他のものより好まれるが常に使えるわけではないというときにくる。私達は以下のようなルールを持っている。

$$(defgoalr(achin-roomr)$$
$$(prio(achfollow-planned-route-tor)$$
$$(achuse-local-navigation-tor)))$$

このルールは、後続のプランの道をにそって、部屋をさ迷うべきであるということを述べている。しかし、もし何らかの理由でそれが不可能であったら、局所的道案内にしたがって進むべきである。同じ効果が *if* 表現でも達成されるが、このルールはより優先順位の高い目標が失敗するであろう余分な状況を知るという高レベルな制約を必要としない。

2.6 優先順位つきの結合

目標の優先順位付の集合の面白い特別な場合は、優先順位付の目標の結合である。その場合、最も好まれる目標が全体の結合点であり、より好まれていない目標は目標の連なりのよりいっそう短い接続の先頭にいる。私達は $(prio-andG_1G_2\ldots G_n)$

を以下のようにする。

$$(prio(andG_1G_2 \ldots G_n)$$
$$(andG_1G_2 \ldots G_{n-1}) \ldots$$
$$(andG_1G_2)$$
$$G_1)$$

アイザックアシモフのロボット三原則はこのタイプの目標構造のよく知られた例である。もう一つの例として、話したりブロックを押したりするロボットを考えてください。それは、以下のようなトップレベルの目標を持っている。

$$(prio - and(maintnot - crashed)$$
$$(ach(inblock1room3))$$
$$(mainthumans - not - bothered))$$

それは、話す領域において $NULL$ 文字をもつどんな行動でも *humans-not-bothered* を維持するということをいうルールも持っている。それはまず、*(in ?x ?y)* が *?x* をプッシュするか、人に持ち上げて移動することを頼むことによって達成されることができる。そして、ロボットが壁と接触するのを避ける行動は *not-crashed* を維持する。ロボットがブロックを押すことができる限り、それは、全ての *3* つの状態を満足することができる。しかし、もしブロックが壁にあってそれを押そうとする状態にあれば、壁との距離をとろうとするので、最初の副目標と敵対することになる。最も好まれる目標は達成されない、したがって、接続から最後の状態を落とすことによって獲得される次に最も好まれる目標を考える。現在、人を悩ませることが許されているので、ロボットは、自分のためにブロックを移動するように誰かに頼むことによって自分の目標を満足させることができる。人間がコンパイルするとブロックをコーナーから移動し、ロボットは自動的に以前の押すという振舞に戻る。これは、プログラマーが、世界の状態の全ての組み合わせを明示的に想像する必要無く、柔軟で反応的な振舞をプログラミングするために便利で高レベルな制約である。全ての目標の記号的操作がコンパイル時に起こるということを思い出すことは重要である。実行時に、エージェントは単純に一番最初に真となる状態と関連がある行動を行う。

3 ギャップスの拡張

ギャップスは、エージェントのコンパイル時に配線され得る行動マップを明確化するための適切な言語である。この章では、私達は、コンパイル時に限界までプランニングするため、実行時にプランニングするため、実行中の目標引き落としを行うためにギャップスを拡張し増大する方法を考える。

3.1 目標引き落とし階層に伴う世界的なプランニング

スコッパーは世界的プランの概念を導入した。世界的プランは与えられた目標のために、目標へ導く行動へエージェントのすべての可能な入力状況をマップする関数である。目標のギャップス評価から得られる結果を出すプログラムはトップレベルの目標のサービスにおいて状況を行動へマッピングする世界的プランとして考えられる。

スコッパーのアプローチは、ギャップスのものと、オペレータ表記言語においてエージェントの能力をユーザが明確化するという点で異なる。この言語のおかげで、ユーザはオペレータと呼ばれるエージェントの原子的な能力を明確化することができ、それぞれのオペレータを実行する期待された影響が、世界上にあり、おそらくオペレータが実行される世界の状態に依存する。

オペレータを特徴づけるもうひとつの方法は、認識関数を使うことである。もし qが世界に保持されているときはいつでもエージェントのおこなう行動 α が結果として p が世界に保持されるならば、*q=regress(α,p)* という関係が保持される。一般的に退化関数は最も弱い q のようなものを返す。退化は普通目標状況 p からまきもどして見られる。命題 q は p を満足させる状況の集合から離れるたった１つのステップやオペレータである状況の集合を表記する。私たちは、もしエージェントが q を満足させる状況にたどり着くことができるなら、簡単に p を満足させる情教にたどり着くことができる。

以下の階層的ギャップスルールのおかげで、それは、世界的プランを構築するために、プランナーによって典型的になされる限界までの後ろむき連鎖探索をすることができる。ギャップスコンパイ

ラは、ルールが欠点によって無限の後ろむき連鎖を引き起こしてしまうので、深さに制限を与えた後ろむき連鎖によって僅かに増大されなければならない。

$$
\begin{aligned}
&(defgoalr(ach(before?p?q))\\
&\quad(if(holds?q)\\
&\quad\quad fail\\
&\quad(if(holds?p)\\
&\quad\quad(doanything)\\
&\quad(if(holds(regress?a?p))\\
&\quad\quad(do?a)\\
&\quad(ach(before(regress?a?p)\\
&\quad\quad(regress?a?q)))))))
\end{aligned}
$$

(ach (before ?p ?q)) の形式の目標のために引き落としルールがある。すなわち、目標はほかの状態 q が獲得する前に状態*?p* を達成することである。目標を達成するこの形式は典型的であると私たちは考える。すなわち、エージェントがどれだけ長くかかってもなにかを達成するという目標をもっているということはあまりない。ルールは次のように働く。もし世界において*?q* が真であるならエージェントは失敗する。もし*?p* が世界において保持されていたら、エージェントは目標を達成したから、どんなこともできる。それ以外でもし、どんな行動*?a* にとっても *(regress ?a ?p)* が保持されていたら、この目標は目標 *(do ?a)* へ引き下げる。最後に、どんな行動*?a* の *(before (regress ?a ?p)(regress ?a ?q))* という目標を達成することを引き下げられることができる。最後の引き落としはエージェントが行動*?a* が開放される状態*?q* を達成する状態に移行する前に行動*?a* が目標*?p* を達成する状態へエージェントが移行することはよいということをいう。なぜならば、いったんそれがおこなわれたら、エージェントがしなければならない全てのことは行動*?a* である。

基本的な *3* ブロックとブロックワールド問題の課程のアプリケーションを考えましょう。*pab* のような行動は原子と呼ばれる。原子は、*put a on b* のように記される。世界は、*clear a* を記す *ca* や、*on b table* を記す、*obt* のような叙述によって表記される。*time(i)* のような追加的な叙述は *0* から始まるグローバル時間が i ならば、真となる。私達は、*time(i)* を表すための省略形 t_i を用いる。目標 *(ach(before(and oab obc)(time 2)))* が与えられたとき、評価過程は、以下で書かれているプログラムを返す。

$$
\begin{aligned}
\{&<(\neg t_2 \wedge obc \wedge ca \wedge cb), pab>,\\
&<(\neg t_2 \wedge \neg t_1 \wedge obc \wedge ca \wedge cb), pat>,\\
&<(\neg t_2 \wedge \neg t_1 \wedge obc \wedge oab \wedge ca), pat>,\\
&<(\neg t_2 \wedge \neg t_1 \wedge ca \wedge cb \wedge cc), pbc>,\\
&<(\neg t_2 \wedge \neg t_1 \wedge oba \wedge cb \wedge cc), pbc>\}
\end{aligned}
$$

プログラムによれば、もし b が c の上にあり、a と b の上に何もなく、時間が *2* でなかったら、エージェントは b の上に a をおくことができる。そうでなく、時間が *1* でも *2* でもなければ、エージェントはたくさんの他のことをすることができる。例えば、b が c の上にあり、a と b の上に何もなければ、エージェントは a をテーブルの上に置くことができる。これは、プログラムの一般性を例証している。それがまだ時間 *1* でなくて、時間 *2* の前に、a を b の後ろに置く時間があるから、過程をもとに戻すことが許される。このプログラムが完全でないことに注意してください。二つの行動で、目標を満足することができないブロック状態があるから、何の行動もとれなくなるような状況がある。これはギャップスに用いられている形式をもとにしたプログラムであるので、グローバル維持目標のような他の目標から起こるプログラムが連接され得るということも注意してください。二つ目の状態を達成する前に一つ目の状態を達成する一連の行動を許すことに置いて、その一般性は、他の制約を表現するプログラムとの連接が $NULL$ でないプログラムに帰着するでしょう。

3.2 分析プランナーを伴う並列作業

状態空間があまりに大きくて網羅的なプランニングが、コンパイル時には非現実できであるとき、実行時プランニングシステムとギャップスの枠組を統合することによるプランニング問題として表記される問題に答えることが可能である。

私達は、プランニング過程を、*1* ステップずつ刻まれる増加的計算として表現できる。それぞれのステップに置いて、過程は、出力を行うが、それは、「私は、まだ答えがでていません」という意味の出力一つだけかも知れない。何ステップか後

にプランニング問題の大きさに依存して、プランナーは現実の結果を生み出す。この結果は、伝統的なシステムに置いて貯められて、実行される。そして、エージェントは初めの行動を行い、プランナーが新しい計画を発生するのを待つ。

プランナーが仕事を始めてから時間がたっても良いので、私達は、プランナーが作ったプランが、プランナーが終了するときに、エージェントが見る状況にとって、適切であるということに注意しなければならない。もし、プランナーがプランの正しさが依存している世界の状態を監視しているなら、これは発生される。もしこれらの状況のいくつかが、偽になったら、プランナーは再び始まることができる。この振舞はいつも正しいわけではないが、正しいでしょう。悪い場合、プランナーは続けて「わからない」という信号を送る。そして、エージェントは反射的にプランの利益なしで環境に反応する。

上記で議論したプランナーの種類は「いつでも」アルゴリズムの退化した形式である。「いつでも」アルゴリズムはいつも答えを持っているが、その答えは事項中に改定される。上記で挙げた例に置いて、答えは、しばらくの間や区に立たないし、一つのステップで劇的に改定される。もっと段階的に改定するプランニングアルゴリズムを持つことは有用であるかも知れない。そのようなアルゴリズムはある種の経路プランニングにおいて存在する。例えば、最初にいくつかの経路が返されるが、そのアルゴリズムは経路をより短く効果的にするように働く。しかし、最適でないと知られているプランをとるか、プランニングにより多くの時間を使うかについて決定することはまだ難しい。たくさんある日課にとって、最適性は一番求められているものではないし、もしプランが完全に要求されたなら、単純なプランに基づいて行動するだけで十分である。

ギャップスの展望から、「いつでも」プランナーは状態を持っている見通しがきく過程である。それは、「もし私が行動γに続く行動βに続く行動αを行ったら、私の目標は達成されるでしょうという状態に世界がいる」という形式の状態を「認識する」ことである。以下のギャップスプログラムはそのようなプランナーの役に立つが、緊急の状態に反応するための潜在能力も持っている。

$$
\begin{aligned}
&(defgoalr(ach(inroom)[rt])\\
&(if(know-plan-for-getting-to-roomrt)\\
&(achexecute-first-step\\
&(plan-for-getting-to-roomrt))\\
&(if(time-is-critical-for-getting-to-roomrt)\\
&(achdrive-in-the-direction-of-roomr)\\
&(maintsit-still))))
\end{aligned}
$$

もしエージェントが時間tに部屋rにいるという目標をもっていて、エージェントがそこへ行くプランを知っているならば、彼は、そのプランの最初のステップを行うでしょう。そうでなくて、もし実行中のようだったら、エージェントはその瞬間に考えられるもっとも良い行動をするべきでる。もしそのときに何の問題もなければ、彼のもっとも良い行動のコースは、認識部分がプランを返すまでじっとすわって待っていることである。プランニングと反射的行動を組み合わせるこれらの問題は*Kelbling*によってより詳しく研究されている。

3.3 実行中目標

今まで私達はエージェントのトップレベル目標がコンパイル時に明確化されている場合にのみ焦点をしぼってきた。実行中に目標を獲得するものとしてエージェントを考えることはしばしば有効的である。

3.3.1 タスク指名

実行時間目標に反応する最も簡単な場合は、それらを認識された情報の他の型であると考え、与えられた目標のにおいて条件付けられる目標引き下げルールを書くことである。これの例として、エージェントは以下の形式の引き下げルールと順

序に従う静的なコンパイル時の目標を与えられる。

$$
\begin{aligned}
&(defgoalr(maintfollow-orders)\\
&(if(current-request-pending)\\
&(achgoal-encoded-by\\
&(perceived-command))\\
&(dotwiddle-thumbs)))\\
&(defgoalr(achgoal-encoded-byparams)\\
&(if(move-commandparams)\\
&(achdo-move-command\\
&(get-destinationparams))\\
&(if(stop-commandparams)\\
&(achstopped)...)))
\end{aligned}
$$

エージェントは自然なままの要求をもとにした正しい目標引き落としをタスク指名することによってエージェントが要求を認知するような要求を実行するでしょう。この方法は、たくさんの場合を満足するが、少なく限られた型となる実行時の目標を必要とする。なぜならば、異なる型は、テストされなければならないし、直接的にタスク指名しなければならない。

3.3.2 実行時目標引き下げ