TR-235

並列問題解決用言語 ANDOR-II

竹内彰一，高橋和子
清水広之(三菱電機)

March, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列問題解決用言語　ＡＮＤＯＲ－Ⅱ

ＡＮＤＯＲ－Ⅱ: A Language for Parallel Problem Solving

竹内彰一　　高橋和子　　清水広之
三菱電機　中央研究所

複数の構成要素が並列に動作する並列動作系を対象とする組合せ問題や協調型プラン生成などの問題を簡潔に表現できる言語ＡＮＤＯＲ－Ⅱについて述べ、さらにその表現をＧＨＣに変換して問題を並列に解く方法を示す。

1　はじめに

committed choice型並列論理型言語はアルゴリズムの明確なものについては十分な記述力をもっていることはほぼ確認されている。しかし一般の問題解決用言語として十分に使い易いものであるかどうかはPrologがそうであるほどには明らかでない。我々は複数の構成要素が並列に動作する並列動作系を対象とする問題解決システムをcommitted choice型並列論理型言語を用いて開発中である。本論文では我々のとったアプローチについて述べる。

まず、並列動作系を対象とする問題解決の問題点を２章で明確にする。その中でcommitted choice型並列論理型言語は問題を表現する言語としては低レベルすぎることを明らかにし、問題向き言語によるアプローチについて提案する。問題向き言語によるアプローチとは問題を簡潔に表現する言語と、その表現に基づいた探索・シミュレーションを含んだ推論メカニズムからなる。３章ではこの表現言語の言語仕様と計算モデルについて述べる。この言語はＡＮＤ並列とＯＲ並列の両並列性を含む。ＡＮＤ並列を並列事象の表現に、ＯＲ並列を複数の可能性をもったものの表現（可能世界の表現）に用いることにより、与えられた問題を宣言的に、かつ容易に表現することが可能となる。４章ではこの表現言語をcommitted choice型並列論理型言語の一つであるＦＧＨＣにコンパイルする方法について述べる。コンパイルの結果得られるＦＧＨＣプログラムは問題表現中のＯＲ並列記述が示唆するあらゆる可能性に沿った系のシミュレーションを行なうようなプログラムである。このプログラムに適当な条件を与えて実行すれば、その条件に合致するような可能世界における系のあらゆるデータを得ることができる。５章では一例として論理回路の故障診断を取り上げる。問題の表現は回路の構造記述等の他、故障の可能性のある素子の宣言、未知入力の指定からなる。この表現をＦＧＨＣにコンパイルし、それに回路の出力値についての条件を与えて計算するとその出力値に合致するような入力値、故障素子のあらゆる可能性が得られることが示される。６章では本節のアプローチを他のアプローチと比較する。

2　並列動作系を対象とする問題解決

2．1　動機と目標

Prologに代表される論理型プログラミング言語の特徴は従来より次のように言われている。

(1) 論理変数を扱えること
(2) バックトラッキングがあること
(3) 計算と証明の同一の過程で行なわれること

(1) はユニフィケーションがあることと言っても良いが、この特徴により論理型言語では部分的にしか定義されていないデータ構造を自然に扱えるようになっている。この特徴はしばしばd-listなどを例にとり、論理型言語が強力なリスト処理言語であるという主張の根拠となってきた。(2) もまたPrologに特有な特徴であり、この特徴により探索型アルゴリズムが非常に容易に書けるようになっており、これがしばしばPrologが人工知能向き言語であるという主張の根拠となっている。(3) は論理型言語の理論的基盤ならびに、実行のセマンティクスを与えるものとなっており、これを用いたメタ・プログラミングなどが可能となっている。

一方、一般に並列プログラミング言語の特徴は次のように考えられる。

(4) 計算の順序を半順序的に指定できること
(5) 複数の計算の間の同期を表現できること

並列論理型言語は論理型言語を基に並列プログラミングが可能なように拡張された言語とここではとらえる。並列論理型言語の１つの大きな流れはRelational Language, Concurrent Prolog, PARLOG, GHCなどのいわゆるcommitted choice型並列論理型言語族である。

committed choice型論理型言語の問題解決システム等への適用としては次の２つの可能性がある。第一の可能性は問題解決アルゴリズムの並列化である。これ自身は問題対象が並列性をもつものかどうかに依存しない。第二の可能性は問題対象の並列化である。これは問題解決の対象を複数の構成要素が互いに作用を及ぼし合いながら並列に動作する並列系に拡張するということである。具体的には論理回路ネットワーク、原子炉制御系、プラント・システムなどを対象とする問題解決が考えられる。

第一の可能性については、Prologのgenerate and test型アルゴリズムをcommitted choice型並列論理型言語のプログラムに変換する研究が最近活発に行われており、一応の成果を得ている。しかし、完全な変換法についてはまだ知られていない。これは探索に伴う多重環境の管理がPrologでは極めて柔軟であり、それに相当するものを並列環境下で実現することが容易ではないことによる。

第二の可能性についてはまだ本格的な研究はなされていない。今まで行われてきたことはcommitted choice型並列論理型言語をシミュレーション用記述言語として用いることに止どまっていた。すなわち、系の構造や各構成要素をcommitted choice型並列論理型言語で記述し、それに適当な入力データを与えて実行し結果を調べるという使い方である。シミュレーションで得られるものは系の動作の一例である。系の動作がいろいろな可能性を含んでいるときにそれらに関してシミュレーションは何の情報も提供しない。またそれらの可能性の集合に基づいた推論等も不可能である。シミュレーションを問題解決の一つと考えることはできるが、それは上に述べた意味で非常に限られたものである。

問題解決と言ったときの問題の意味をもっと明確にしよう。ここでは次のような並列動作系上の組み合わせ問題を考える。

以下の条件の集合を満足するように変数
Ｖ１，．．．．，Ｖｎの値を定めよ

条件１，．．．．，条件ｍ

問題対象が並列動作系の場合、条件の中には、

(1) 系の構造の記述、

(2) 各構成要素の可能な動作およびそれに伴う状態の変化に関する規則

(3) 系の入力値、出力値に関する条件、

等が含まれる。また求めるべき変数としては例えばある構成要素のある時点での状態、ある構成要素がある時点でとった動作などが考えられる。並列動作系における故障要素の検出や系の制御プログラム生成などもこの組み合わせ問題の一例と考えることができる。

組み合わせ問題を解くためにはV1，．．．．Vnの各々の取りうる値で構成されるn次元空間の探索が不可避である。このn次元空間中の一点は一つの動作例に相当する。すなわち、一つのシミュレーションが一点に対応する。このような問題を計算機を用いて容易に解けるようにするためには上の(1)～(3)を自然に記述できる言語と強力な探索機構が必要である。

committed choice型並列論理型言語は記述言語としては不適当である。なぜなら今まで提案されている committed choice 型並列論理型言語はすべて『commited choice』という思想のもとに、Prolog に代表される論理型言語の問題解決などへの適性である前述の特徴(2)、すなわち自動的な探索機能を失っているからである。この結果、ある状況の下でとるべき動作やその結果生じる状態の変化が一意に確定しないような系の構成要素の記述が不可能になっている。しかしながら、このような不確定要素は組み合わせ問題においては本質的なものなのである。

2．2 問題向き言語によるアプローチ

すでに述べたように並列動作系を対象とする問題解決においては問題を自然に記述できる言語と強力な探索機構が必要である。ここでとったアプローチは次のように要約できる。

(1) 並列動作系およびその構成要素の動作を自然に表現する言語を設計する（動作が一意に確定しない構成要素も簡潔に記述できる）

(2) この表現言語による記述に基づき、シミュレーションや探索などを交えた推論によって系の種々の性質を推定する方法を確立する

(3) この方法を具現化したソフトウェアを並列論理型言語を用いて開発する

表現言語に必要な機能は並列系の記述および不確定要素の記述である。並列系の記述に関してはcommitted choice型の並列論理型言語は十分な記述能力をもっていると考えられる。しかしこれだけでは不確定要素の記述ができないので committed choice という概念をもたない記述要素が別に必要である。

ここで提案する表現言語はＡＮＤＯＲ－Ⅱと呼ばれ、２種類の述語をもつ。第１の種類は AND関係と呼ばれ、これに属する述語はガード付き節だけから定義されてなくてはならない。第２の種類は OR 関係と呼ばれ、これに属する述語はコミット・オペレータをもたない節（無ガード節と以下呼ぶ）だけから定義されている。１つの述語がガード付き節、無ガード節の両者を用いて定義されることはない。しかしガード付き節および無ガード節ともに右辺に並ぶリテラルは AND関係に属するリテラル、OR 関係に属するリテラルの両者が混じってかまわない。AND 関係は確定要素および非決定的要素の記述に、OR関係は不確定要素の記述に基本的に用いられる。またこの言語はある意味で committed choice 型並列論理型言語とカット付き Prolog のスーパーセットと考えることもできよう。

3 並列動作系表現言語 ＡＮＤＯＲ－Ⅱ

3．1 仕様

並列動作系表現言語ＡＮＤＯＲ－ⅡはＡＮＤ並列性を有するＦＧＨＣにＯＲ並列性を付加することにより、全解探索機能を導入した並列問題解決用言語である。

ＡＮＤＯＲ－Ⅱのシンタクスを図１に示す。ここで、Name,Integer,Head,Guard,BodyはＦＧＨＣと同様に定義される。ＡＮＤＯＲ－Ⅱでは、述語はＡＮＤ関係とＯＲ関係の２種類に分類され、すべての述語に対して、いずれの関係であるかを次のように宣言する。

```
:- and_relation 述語名／引数の個数,
:- or_relation 述語名／引数の個数.
```

ＡＮＤ関係の述語はＡＮＤ節（ガード付き節）のみから定義され、ＯＲ関係の述語はＯＲ節（無ガード節）のみから定義されなければならない。従って、ＡＮＤ関係，ＯＲ関係の両方で宣言することや、ＡＮＤ節，ＯＲ節の両方を用いて定義することはできない。しかし、各節のボディ部にはＡＮＤ関係，ＯＲ関係の両方のゴールが混在してもよい。

ＡＮＤＯＲ－Ⅱでは、すべての述語に対して、その引数の入出力モードを予め宣言しておかなければならない。このモード宣言では、各引数のモードは引数が入力引数の場合には“＋”，出力引数の場合には“－”とし、一つの述語に対して複数のモード宣言を行うことはできない。従って、入出力モードが明確でない述語や多モードの述語は定義できない。また、入力引数はゴール呼び出し時に既に基底項になっているものとし、出力引数はゴール呼び出し後に基底項に具体化されるものとする。つまり、ゴール呼び出しにおける入出力は共に基底項しか許されない。

ＡＮＤＯＲ－Ⅱプログラムはモード宣言，関係宣言，述語定義からなる。

3．2 計算モデル

ＡＮＤＯＲ－ⅡはＡＮＤ並列性とＯＲ並列性の両並列性を有する。つまり、論理積の関係にあるすべてのゴール・リテラルのリゾリューションを並列に実行するというＡＮＤ並列性と、ある述語の論理和の関係にある節のうち適用可能なすべての節によるゴール・リテラルのリゾリューションを並列に実行するというＯＲ

```
Sentence                ::= Mode_Declaration | Relation_Declaration | Clause
Mode_Declaration        ::= (:- mode Predicate_Modes)
Predicate_Modes         ::= Predicate_Mode | (Predicate_Mode , Predicate_Modes)
Predicate_Mode          ::= Predicate_Name | Predicate_Name ( Modes )
Modes                   ::= Mode | (Mode , Modes)
Mode                    ::= "+" | "-"
Relation_Declaration    ::= (:- Relation Predicate_Name / Arity)
Relation                ::= "and_relation" | "or_relation"
Predicate_Name          ::= Name
Arity                   ::= Integer
Clause                  ::= AND_Clause | OR_Clause
AND_Clause              ::= ( Head :- Guard "|" Body )
OR_Clause               ::= ( Head :- Body )
```

図　　1

並列性を有する。ＡＮＤＯＲ－Ⅱでは、このＯＲ並列性の導入による探索空間の増大をＡＮＤ関係，ＯＲ関係の２つに述語を分類することによって防いでいる。つまり、ＯＲ並列はＯＲ関係として定義された述語の節に対してのみ適用され、ＡＮＤ関係の述語に対しては適用されない。

ＡＮＤ関係，ＯＲ関係に属するゴール・リテラルのリゾリューションはそれぞれ次のようになる。ＡＮＤ関係に属するゴール・リテラルのリゾリューションは、ＧＨＣと同様にコミット・オペレータにより一つの節に限定され、ある節が選択されるとそれ以外の節を選ぶ可能性は棄却される。これに対して、ＯＲ関係に属する述語はコミット・オペレータをもたない無ガード節を用いて定義されており、そのゴール・リテラルのリゾリューションはヘッドとゴールとがユニフィケーション可能なすべての節について並列に実行される。つまり、リゾリューションが一つの節に限定されることはなく、リゾリューションのすべての可能性を保持する。従って、例えばゴール・リテラルＡとＯＲ関係の述語Ｈのｎ個の節，　Ｈｉ :- <Ｂｉ>．ｉ=1,...,ｎ　の各ヘッドＨｉとのユニフィケーションがすべて可能な場合には、リゾリューションの結果，ゴール・リテラルＡは <Ｂ1> or...or <Ｂn>　で置換えられる。元の論理世界が Ａ and Ｐ1 and...and Ｐm（ Ａ.Ｐ1,....,Ｐm　が論理ＡＮＤで結合）であったとすると、このＡのリゾリューションにより新しい世界は（<Ｂ1> or...or <Ｂn>）and Ｐ1 and...and Ｐmとなる。ところで、

```
        A and P1 and...and Pm
                ↓
(<B1> or...or <Bn>) and P1 and...and Pm
    = (<B1> and P1 and...and Pm)
    or(<B2> and P1 and...and Pm)
                :
    or(<Bn> and P1 and...and Pm)
```

であるから、結果的に論理ＯＲで結合されたｎ個の世界が生じることになる（図２）。従って、一般に、ＯＲ関係に属するゴール・リテラルのリゾリューションでは適用可能な無ガード節が複数個あった場合には結果として元の世界が節の数だけ枝分かれし、互いに論理ＯＲの関係にある複数の世界が生じる。そして、その後のリゾリューションはそれぞれの世界において並列に実行されることになる。

以上述べたように、ＡＮＤＯＲ－ⅡではＯＲ関係に属するゴール・リテラルのリゾリューションにおいてはあらゆる可能性が保持される。従って、このＯＲ関係を用いることにより全解探索プログラムを容易に書くことができる。

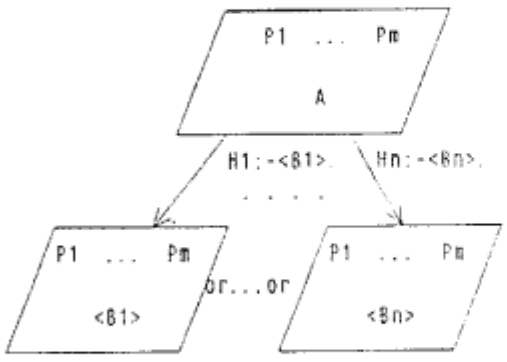


図2

4　実行系

4．1　ＦＧＨＣへの変換

ＡＮＤＯＲ－Ⅱで記述されたソース・プログラムはコンパイラによってＦＧＨＣのプログラムに変換される．ソース・プログラムにおけるボディ・ゴールは，そのままＡＮＤ並列に実行されるように変換される．ＯＲ関係で定義された述語（ＯＲ述語）については，各節を選択した場合のリゾリューションの結果が色付き値としてストリームの形にまとめられ，ストリームから各データを受け取ったプロセスが，その値に対する処理をＡＮＤ並列に行うように変換される．従って変換後のプログラムはＡＮＤ／ＯＲ両並列性を反映する．

4．2　変換系の概要

変換系は，ＤＦＡとＴＲＡの２つのモジュールから成る（図３）．ＤＦＡ（Data Flow Analysis）は解析部であり，ソース・プログラムを読み込んで静的解析により各節に対するＤＦＧ（Data Flow Graph ）をつくり，このグラフを使って動的解析を行ない，その結果を中間コードとして出力する．ＴＲＡ（TRAnslation ）は変換部であり，中間コードをＦＧＨＣのプログラムに変換する．

まず，基本的な概念を説明する．ＤＦＧは，各ノードがボディ・ゴールに，各エッジが変数にそれぞれ対応するグラフである．このグラフはループを持たず，同じ変数名に対応するエッジが異なるノードから同一のノードに引かれていることはない．エッジは各ゴール間の共有変数に相当し，ノードを結ぶチャネルと見なすこともできる．ＤＦＧにおいて，ＯＲ述語あるいはＯＲ述語をボディ・ゴールに持つ節を含む述語に対応するすべてのノードをＯＲノードと呼ぶ．ＯＲ述語によって世界が枝分かれした場合，変数にはストリーム型のデータが bind される．ＤＦＧにおいて，ストリーム型のデータが流れる可能性のあるチャネルをベクタ型，それ以外をスカラ型と呼ぶ．ＤＦＧはソース・プログラムの各節と１対１に対応しているので，以後ＤＦＧを中心に説明する．

次に，各モジュールについて詳細な手続きを説明する．

（１）ＤＦＡ

①ＤＦＧ作成

まず，ソース・ファイルを読み込んで各節に対応するＤＦＧを作成する．この際，引数に変数，定数以外の項を含むゴールは規則に従って，その様な項が現れない等価なゴールに書換える．従って，ＤＦＧにおける各ノードのチャネル数は対応するゴールの引数の数と一致する．

②Shell の必要なゴール指摘

ベクタ型の入力チャネルを持つノードに対しては，shell をか

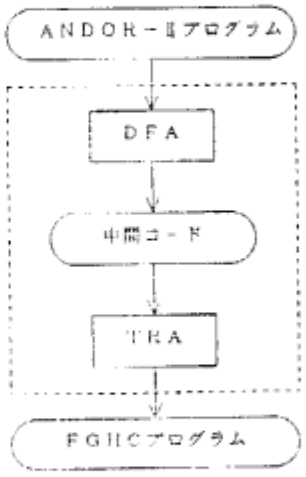


図3

けて処理をする必要がある．ベクタ型のチャネルは以下の規則に従って検出される

(a) ORノードを始点とするチャネルはベクタ型である．
(b) ベクタ型の入力チャネルを持つノードの出力チャネルはベクタ型である．
(c) それ以外のチャネルはスカラ型である．

③出力変数のタイプの判定

一般に述語Pは複数個の節で定義されるので，PのDFGも複数個存在する．その中でPの出力引数に対応するチャネルでベクタ型のものがあればその引数はストリーム型のデータを持つと判定する．

④中間コード生成

以上の手続きで得られた情報をもとに中間コードを生成する．中間コードは，ソース・プログラム及びDFG作成の際出現した全述語のリストからなるリスト部，各述語に対する引数の数，関係，出力変数のタイプ，shell の情報を含む情報部，情報を付加された定義節より成る定義部の3部で構成される．

（2）TRA

①ShellCreation 生成

まず，中間コードに含まれる情報に従って，対応するShellCreation 節を生成する．Shell は以下の機能を持つ．

(a) 各ストリームを分解して単一データの組を取り出す．
(b) 各データ組に対して，それらが同一世界に属するかどうかを判定し，同一世界に属していれば Core プロセスに送り，そうでなければその組は捨て去る．
(c) 結果を再びストリームの形にする．

②Check 生成

次に，Shell から呼び出されるプロセスCheck の定義節を生成する．Check は各データに付随する色の整合性を調べ，整合する（同じ世界に属する）ならば，Core プロセスにそのデータ組を送り，すべての色の和集合をその解の色として付随させる．そうでなければその組は捨て去る．

③OR－AND変換

非決定的なOR節を，決定的な節に変換する．即ち，述語の定義節をAND並列なゴールで呼び出す節を生成し，結果をマージして出力ストリームとする．従って，deadlockや無限ループに陥るものがあっても可能な解はすべて集めることができる．

④述語変換

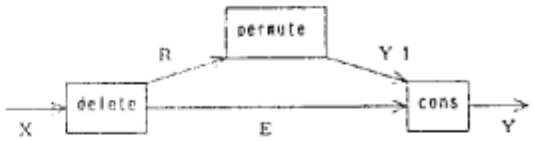
最後に，すべての述語に対する変換を行なう．この手続きでは述語名のつけ変え，色の情報を各プロセスに受け渡すための引数の付加を行なう．head unificationが失敗する可能性のある述語に関しては，失敗した場合の処理をする節を追加する．また，必要な節に対しては，出力のストリーム化を行なう．さらに，OR節に関しては，何番目の節を選択したかという情報を与えるためのゴールが付加される．

4．3　変換例

例として，以下のようにAND OR－Ⅱで記述されたpermutation のプログラムを考える．

```
:- mode permute(+,-), delete(+,-,-).
:- and-relation permute/2.
permute([],Y) :- true | Y=[].
permute(X,Y)  :- true |
   delete(X,E,R), permute(R,Y1), Y=[ E | Y1].
:- or-relation delete/3.
delete([ X | X1],E,R) :- E=X, R=X1.
delete([ X | X1],E,R) :-
   delete(X1,E,R1), R=[X | R1].
```

以下に，変換過程と変換後のプログラムの一部を示す．まず，DFAでは，このプログラムを読み込んで各節に対応するDFGを作成する．例えば，permute の第2節は以下のようなDFGになる．



次に，Shell の必要なゴールを抽出する．このDFGでは delete のみがORノードであり，ここからのデータ・フローを受ける2つのノード permute,cons は共にShell が必要になる．続いて，各述語の出力タイプを判定する．このDFGでは，出力チャネルYはベクタ型であるが，permute の第1節のDFGにおいてはスカラ型である．従って，permute の出力変数はストリーム型と判定される．DFAは最後にこれらの情報を持つ中間コードを生成する．

次に，TRAが中間コードの情報部を見ながら定義部をFGHCのプログラムに変換する．上図のノード cons は2つのベクタ型入力チャネルを持つので，各々を分解するShellCreation 節を生成する．

```
cons-Shell-2-1( [v(X,Cx) | Xs],Y,Z) :-true |
    cons-Shell-2-2(v(X,Cx),Y,Z1),
    cons-Shell-2-1(Xs,Y,Z2),
    out-Merge(Z1,Z2,Z).          % 結果の収集
cons-Shell-2-1([],Y,Z) :-true | Z=[].

cons-Shell-2-2(v(X,Cx),[v(Y,Cy) | Ys],Z) :-true |
    cons-Check-2 1(v(X,Cx),v(Y,Cy),Z1),
                                 % 色の無矛盾性チェック
    cons-Shell-2-2(v(X,Cx),Ys,Z2),
    out-Merge(Z1,Z2,Z).
cons-Shell-2-2(X,[],Z) :-true | Z=[].
```

分解されたデータ組の色の無矛盾性を調べるCheck 節を生成する．

```
cons-Check-2-1(v(X,Cx),v(Y,Cy),Z) :- true |
    same-Color([Cx,Cy],R), cons-Check-2-2(R,X,Y,Z).
cons-Check-2-2(success(C),X,Y,Z) :- true |
    cons-Core(X,Y,Z0,w(C)), Z=[v(Z0,C)].
cons-Check-2-2(fail,X,Y,Z) :- true | Z=[].
```

OR述語 delete は各定義節をAND並列なプロセスとして呼び出すように変換される．

```
delete-Core(X,Y,Z,w(C)) :- true |
    get-Counter(Ct),       % Ct: この分岐点を示すID
    delete-Core-1(X,Y1,Z1,w(C),Ct),   % deleteの第1節
    delete-Core-2(X,Y2,Z2,w(C),Ct),   % deleteの第2節
    out-Merge(Y1,Y2,Y),
    out-Merge(Z1,Z2,Z).
```

最後に，すべての述語に対する変換を行なう．

```
permute-Core([],Y,w(C)) :-
     true | VY=[], Y=[v(VY,C)].
permute-Core([X | Xs],Y,w(C)) :- true |
                                 % C:現在の世界の色
    delete-Core([X , Xs],E,R,w(C)),
    permute Shell(R,Z),
```

```
    cons-Shell-2-1(E,Z,Y).

delete-Core-1([X|Xs],E,Y,w(C),Ct) :- true  |
    append(C,[(c1,Ct)],NewC),        % 色の情報の追加
    VE=X, E=[v(VE,NewC)],
    VY=Xs, Y=[v(VY,NewC)].
delete-Core-1(X,E,Y,w(C),Ct) :- true | E=[], Y=[].
                                  % 失敗した場合.
```

この結果得られたＦＧＨＣのプログラムを実行すると，permute の全解が得られる．

5　応用例

ＡＮＤＯＲ－Ⅱの応用例として論理回路の故障診断について述べる．ここでの論理回路の故障診断とは、論理回路の入出力データを観測データとして与え、論理回路内の故障素子及び未知入力値に対するすべての解を求めるものである．ここで、回路の出力はすべて既知であり、入力は未知入力があってもよいものとする．例えば、図4のような場合、入力 In を未知入力，素子 E1，E2 を故障可能素子とすると、

In = "0"　のとき　E1が故障
In = "1"　のとき　E2が故障

というような解を求めるものである．

この回路故障診断の処理は次の三つの部分に分けられる．

(1)　入力生成部
(2)　回路解析部
(3)　出力チェック部

入力生成部では、未知入力値を "0" か "1" かに場合分けして、すべての可能な入力を生成する．回路解析部では、故障可能素子の動作状態を正常か故障かに場合分けして、すべての入力に対して出力を求める．出力チェック部では、得られたすべての可能な出力と観測された出力とをチェックし、それらが一致するものだけについてその入力及び素子の動作状態を返す．以上のように、この例では generate and testの方法で処理を行っている．上記(1),(2)の部分が generator，(3) の部分が tester に相当する．

実際のプログラムのトップレベルの述語 test と出力チェック部の述語checkOutput の定義を次に示す．

```
:- and_relation test/2.
test([DataIn,DataOut],Answer) :- true |
        setInput(DataIn,Input),
        circuit(Input,Output),
        checkOutput(Input,Output,DataOut,Answer).

:- and_relation checkOutput/4.
checkOutput(Input,[Output,State],DataOut,Answer) :-
        DataOut = Output | Answer = [Input,Output,State].
```

述語 setInput，circuit，checkOutput がそれぞれ上述の (1),(2),(3)の部分に相当する．ここで、変数 DataIn,DataOut はそれぞれ入力データリスト、出力データリストであり、既知入出力は "0" または "1" で、未知入力は "?" で与えられる．また、最終的に得られる解は、入力リスト，出力リスト，故障可能素子の動作状態のリストである．すべての述語のうちでＯＲ関係として定義される述語は、述語 setInput の下の未知入力の場合分けを行う述語 setUnknownin ，及び述語 circuitの下の故障可能素子の動作の場合分けを行う述語 elementDoubt のみであり、その他の述語はすべてＡＮＤ関係として定義される．この二つのＯＲ関係の述語の定義を次に示す．

```
:- or_relation setUnknownIn/1.
setUnknownIn(In) :- In := 0.
setUnknownIn(In) :- In := 1.

:- or_relation elementDoubt/5.
elementDoubt(Name,Type,Input,Output,State) :-
        elementi(Name,Type,correct,Input,Output,State).
elementDoubt(Name,Type,Input,Output,State) :-
        elementi(Name,Type,error,Input,Output,State).
```

次に、述語 circuitの定義，つまり回路記述の方法を半加算器回路 (half adder) を例に説明する．半加算器回路を図5に示す．ここで、A,B は入力，S,C は出力であり、C がケタ上げ(carry)である．この回路は次のように定義される．

```
:- and_relation circuit/2.
circuit([In1,In2],Output) :- true |
        element(inv1,inverter,correct,[In1],Out1,_),
        element(inv2,inverter,correct,[In2],Out2,_),
        element(and1,and2,doubt,[In1,Out2],Out3,State1),
        element(and2,and2,doubt,[Out1,In2],Out4,State2),
        element(and3,and2,doubt,[In1,In2],Out5,State3),
        element(or1,or2,correct,[Out3,Out4],Out6,_),
        Output=[[Out6,Out5],[State1,State2,State3]].
```

述語 circuitの最初の引数は入力引数であり、回路への入力のリストである．この場合、変数In1,In2 がそれぞれ入力A,B に対応する．また２番目の引数 Output は出力引数であり、回路の出力と故障可能素子について検出された動作状態のリストを返す．Outputへの代入はボディ部の最後のゴールで行われているが、この場合変数Out6,Out5 がそれぞれ出力S,C に対応し、変数State1,State2,State3が故障可能素子について検出された動作状態であることを示している．

各素子に対する述語 elementの引数は次のような意味を持つ．

element(Name,Type,State,In,Out,State1)

- Name　：　素子名
- Type　：　素子の種類
  inverter,and,or など、
  この例の and2,or2 は２入力 and,or を示す．
- State　：　与えられた動作状態
  doubt,correct,error のいずれかであり、それぞれ故障可能素子，正常素子，故障素子を示す．
- In　：　入力リスト
- Out　：　出力リスト

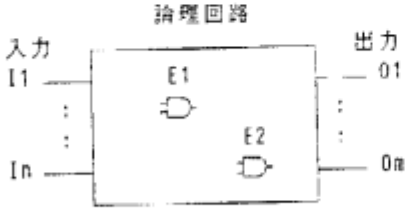


図4

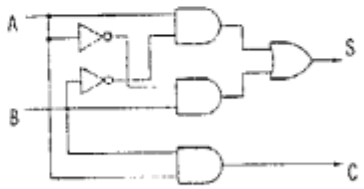


図5

2出力以上の場合にリストとなる。

State1： 求められた動作状態
correct またはerror

ここで、Name,Type,State,Inは入力引数、Out,State1は出力引数である。Typeで用いられる基本的な素子の動作は次のように定義される。

```
:- and relation element2/4.
element2(inverter,State,Input,Output) :- true |
      inverter(State,Input,Output).
element2(and2,State,Input,Output) :- true |
      and2(State,Input,Output).
element2(or2,State,Input,Output) :- true|
      or2(State,Input,Output).

:- and relation inverter/3.
inverter(correct,[0],Out) :- true | Out:=1.
inverter(error,[0],Out) :- true | Out:=0.

:- and relation and2/3.
and2(correct,[0,0],Out) :- true | Out:=0.
and2(error,[0,0],Out) :- true | Out:=1.
```

各素子は以上のように記述されるが、各素子間の結合は入出力変数In,Outの共有関係で示される。

次に、この半加算器回路での故障診断の実行例を示す。例えば、観測データとして入力 A= "1", B = "?", 出力 S = "0", C = "0" が与えられたとすると、

```
?- andor test([[1,'?'],[0,0]],Answer).
```

として故障診断が実行され、

```
Answer : [ Input, Output, State ] =

[[1,0],[0,0],
      [(and1,error),(and2,correct),(and3,correct)]]
[[1,1],[0,0],
      [(and1,correct),(and2,correct),(and3,error)]]
```

という解が得られる。つまり、診断の結果、与えられたデータに対して、

- 未知入力B が "0"のとき、素子and1が故障で素子and2,and3 は正常
- 未知入力B が "1"のとき、素子and3が故障で素子and1,and2 は正常

という二つの解が存在するということが分かる。

6 議論

本節で述べたことは並列動作系を対象とした探索問題のcommitted choice 型並列論理型言語へのコンパイル法と位置付けることができる。種々のクラスの探索問題のcommitted choice型並列論理型言語へのコンパイル法について最近いくつかの研究が報告されている。

上田は制限付き Prolog で書かれた全解探索問題をＦＧＨＣにコンパイルする方法について報告している[Ueda86a]。制限とは各述語の入力、出力がそれぞれグラウンド（変数を含まない）でなくてはならないというものである。PrologはＡＮＤ逐次、ＯＲ逐次の言語であるが、コンパイル・コードではこれらはそれぞれＡＮＤ逐次、ＯＲ並列に評価される。すなわちコンパイルによりＡＮＤ逐次はＡＮＤ逐次へと、ＯＲ逐次はＯＲ並列へと変換される。ＦＧＨＣ上ではＡＮＤ逐次をコンティニュエーション、ＯＲ並列をＡＮＤ並列で実現している。上田はこのアプローチに基づいて、同様の制限をもつコルーチン付き Prolog のＦＧＨＣへのコンパイル法についても報告している[Ueda86b]。これはコルーチンを複数のコンティニュエーションをもったＦＧＨＣプログラムへと変換するアイディアに基づいている。コルーチン間の制御の授受（スケジューリング）は静的にコンパイルするという方式をとる。この方式は静的にスケジューリングを決定できるようなクラスの問題を効率良くコンパイルする優れた方式といえる。しかし、コルーチン記述は並列事象の表現力が不十分であること、また並列事象のシミュレーションでは動的スケジューリングが不可欠であるということから、並列動作系を対象とするような問題にはこの方式は適さないといえる。

玉木はＡＮＤ、ＯＲ両並列性をもったPrologで書かれた全解探索問題をcommitted choice型並列論理型言語にコンパイルする方法について報告している[Tamaki86]。この表現言語も上田の場合と同様グラウンド入出力という制限をもつ。またＡＮＤ並列記述は並列に実行されるゴール間の共有変数による通信を許さないという制限をもっている。この方式ではコンティニュエーションのかわりにストリームを用いることによってＡＮＤ並列のコンパイルを実現している。上田の方法に比べ、実行時のオーバヘッドが増える可能性はあるが、表現言語中でＡＮＤ並列表現を許したこと、またその結果コンパイル・コードの並列度が大きくなるという利点をもつ。

本節で提案した方式とこの玉木の方式とは共通のアイディアをもつ。すなわち、表現言語にＡＮＤ並列を許したこと、ストリームによるコンパイルの２点である。しかし両者はＡＮＤ並列で表現可能な問題のクラスという点で決定的に異なる。すなわち玉木の場合ＡＮＤ並列記述では共有変数による通信を許さないが、本方式では許される。この通信機能は並列動作系の記述という観点からは欠くことのできないものである。しかしこれを許すことによって本方式では玉木の場合のストリームよりは複雑な色付けされたストリームの処理が必要となった。

これらの３方式はそれぞれ異なったクラスの探索問題のcommitted choice型並列論理型言語へのコンパイル法を定義している。本方式はこの中で他の２つのクラスを含む最も大きなクラスを表現可能である。そしこのクラスだけに並列動作系を対象とする探索問題が含まれる。しかしながら現状では複雑度は他の２つより大きく、その結果実行効率も同じ問題を対象とすれば一番大きい。

謝辞

本研究は第５世代コンピュータ・プロジェクトの一環として行われた。日頃御指導をいただいているICOT古川研究所次長および伊藤第１研究室室長に感謝します。

文献

[Tamaki86] H.Tamaki, "Stream-based Compilation of Ground I/O Prolog intoCommitted Choice Languages", Tech. Rep. No.86-5, Dept. of Information Science, Ibaraki Univ. (1986).

[Ueda86a] K.Ueda, "Making Exhaustive Search Programs Deterministic", Proc. of 3rd Int. Conf. on Logic Programming, LNCS 225, Springer-Verlag (1986).

[Ueda86b] K.Ueda, "Making Exhaustive Search Programs Deterministic(II)", Proc. of 3rd Conf. of Japan Society of Software Science and Technology (1986).