人　工　知　能　80－2
記　号　処　理　63－2
（1992. 1. 16）

等価変換による回路変換ルールの自動合成

馬淵　浩司　　赤間　清　　青木　由直

北海道大学　工学研究科　情報工学専攻

本論文では、論理回路の変換システムを一般化論理プログラム(GLP)の言葉で定式化し、等価変換を用いることによって、回路変換ルールを自動合成する方法について述べる。回路変換ルールは、GLPのクローズで表現する。ここでの等価変換とは、背景知識のもとで、あるクローズをそれと等価なクローズに変換するものである。等価変換による方法は、従来のHorn節のHeadをBodyに展開するだけの方法よりも処理が柔軟になり、枠組みの面で拡張可能である。さらに、本論文では、いくつかの等価変換ルールを示し、それを用いた合成例を与えることによって、本方法の有用性を明らかにする。

An Automatic Synthesis of Circuit Conversion Rules by using Equivalent Transformation

Hiroshi Mabuchi　　Kiyoshi Akama　　Yoshinao Aoki

Department of Information Engineering
Faculty of Engineering, Hokkaido University

Kita 13,Nishi 8,Kita-ku,Sapporo,060,Japan

In this paper,we describe a method for synthesizing circuit conversion rules automatically by using equivalent transformation.We form conversion system of logic circuits by GLP(Generalized Logic Program).Circuit conversion rules are expressed by clause of GLP.Equivalent transformation is to converse a clause to an equivalent clause on basis of background knowledge.A method for synthesizing by equivalent transformation prevents explosive increase compared to a method that develops Head of usual Horn clause to Body.In addition,we show some equivalent transformation rules and synthesized example by using them.

1 まえがき

LSI 設計に於いて、論理回路をより簡単な回路に変えることは基本的であり、また、複数の回路変換ルールを合成して、新たな回路変換ルールを求めることは、重要なことである。

論理回路を簡単化するときに、回路変換ルールを用いる。本論文では、論理回路を簡単化する変換系を論理の枠組みを用いて記述し、等価変換を用いることによって、回路変換ルールを自動合成する手法について述べる。

2 つの論理式が等価であるとは、それらの真理値がいかなる解釈のもとでも同じであることと定義する。本研究では、背景知識のもとでの等価性を扱う。等価変換とは、この背景知識のもとで、ある論理式をそれと等価な論理式に変換することである。

合成される回路変換ルールは、一般化論理プログラム (GLP) のクローズで表現されている。節の Body の中のいくつかのアトムの論理積が等価変換ルールの'変換前の論理積'とマッチし、しかも条件部を実行して成功したときに限り、マッチした部分が'変換後の論理積'に変換される。

回路変換ルールの自動合成は、部分計算、特に unfolding を用いても行える [1]。unfolding とは、Horn 節の Head を Body に展開する方法である。その時、多くの場合、1 つの Head が複数の節の Head と強制的にマッチすることになる。その複数の Head から複数の Body に展開される。このようにして、部分計算の結果、節の数が増大しやすい。

ところが、等価変換を用いると、1 つの節は、等価な 1 つの節に変換されるので、節の増大が抑制される。このように節の増大現象を起こさずに合成が行われることが、等価変換が unfolding よりも有効である理由である。

2 知識の表現

本論文では、論理回路を以下のように表現する。例えば、素子は次のように書く。

(GN out in connect)

ここで、GN は素子名であり、in は入力端子であり、out が出力端子であり、そして connect が結線先である。

ある素子の出力端子から結線できる結線先は複数あり集合になっている。この集合のことを結線先と呼ぶことにする。

例えば、下図 1 の AND 素子は、

(AND c {d,e,f,g} {a,b})

となる。

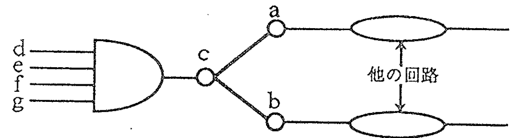


図1 知識の表現

3 等価変換ルール

3.1 等価性と等価変換の定義

論理式 $A \land K$ と $B \land K$ の真理値がいかなる解釈のもとでも同じであれば、またその時に限って、K のもとで A と B は等価であるという。

例えば、今、次のような A_1, B_1, K_1 が与えられているとする。

$$A_1 = (P(x) \leftarrow h(x).)$$
$$B_1 = (P(a) \leftarrow .)$$
$$K_1 = (h(a) \leftarrow .)$$

このとき

$$A_1 \land K_1 \leftrightarrow B_1 \land K_1$$

が成り立つので A_1 と B_1 は等価である。

一方、

$$A_2 = (P(x) \leftarrow h(x).)$$
$$B_2 = (P(a) \leftarrow .)$$
$$K_2 = (h(a) \leftarrow .),(h(b) \leftarrow .)$$

のときには、

$$A_2 \land K_2 \rightarrow B_2 \land K_2$$

は成り立つが、この逆は成り立たない。故に、A_2 と B_2 は K のもとで等価ではない。

次に、等価変換を定義する。

背景知識 K のもとでの等価変換とは、ある論理式 A を K のもとでそれと等価な論理式 B に変換することである。

3.2 等価変換の有効性

従来の Horn 節の Head を Body に展開する方法では、1 つの Head が複数の節の Head と強制的にマッチすると、その複数の節の Head から複数の節の Body に展開されて、その複数の節の Body が再び展開されるという爆発的増大を生じる。

この様子を下の図 2 に示す。

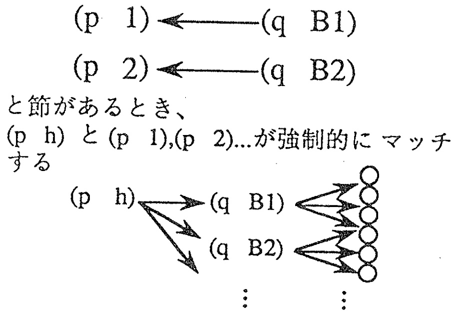


図2　従来のマッチング

ところが、等価変換を用いると、節の Body の中のいくつかのアトムの論理積が等価変換ルールの'変換前の論理積' とマッチし、しかも条件部を実行して成功したときに限り、マッチした部分が'変換後の論理積' に変換される。そのため爆発的増大現象を起こさずに、順当な合成が行われるのである。このことが等価変換が、従来の Horn 節の Head を Body に展開する方法よりも有効である理由である。

この様子を下の図 3 に示す。

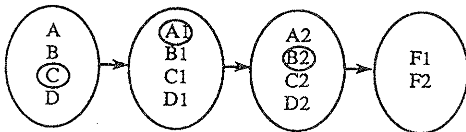
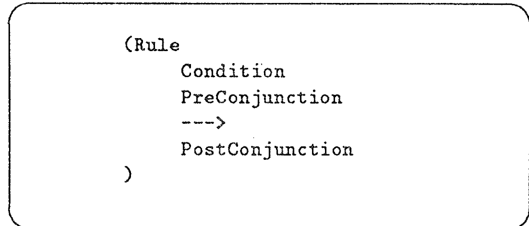


図3　等価変換

上の図 3 では、'変換前の論理積' から '変換後の論理積' に置き換えられるとき (上の図 3 では、丸で囲まれたクローズが置き換えられる) に、その置き換えに伴うユニフィケーション効果による影響 (上の図 3 では、丸で囲まれないクローズが変化することを意味する) が考えられる。

3.3 等価変換ルールの表現

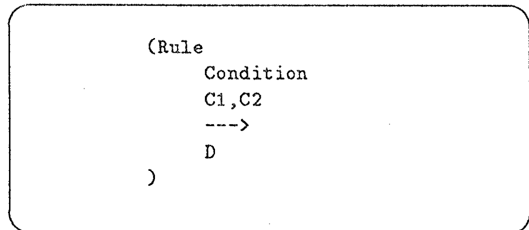
本論文では、3.1で述べた A,B をクローズに限定する。また、等価変換を次の様な形のルールを用いて行う。



```
(Rule
     Condition
     PreConjunction
     --->
     PostConjunction
)
```

ここで Condition は条件を表し、PreConjunction は'変換前の論理積' を表し、PostConjunction は'変換後の論理積' を表す。

この等価変換ルールがどう働くかを例で説明する。そこで次のような等価変換ルールの具体例を考える。



```
(Rule
     Condition
     C1,C2
     --->
     D
)
```

$$\mathrm{H} \leftarrow \mathrm{B}_1, B_2, B_3, B_4 ...(1)$$

という節が与えられたとする。

上の図で C_1, C_2が (1) の節の B_2, B_3にマッチするとする。それから *Condition* の条件を実行し、成功したときに C_1, C_2が Dに置き換えられる。それによって (1) は (2) に変換される。

$$\mathrm{H} \leftarrow \mathrm{B}_1, D, B_4 \quad ...(2)$$

3.4 等価変換ルールの効果

Horn 節 ($\mathrm{H} \leftarrow \mathrm{B}_1, B_2, \cdots, \mathrm{B}_n$) を用いた部分計算では、$Head(H)$ を $Body(B_1, B_2, \cdots, \mathrm{B}_n)$ に展開する。この展開操作は、ルールで次のように表すことができる。

```
(Rule
     (true)
     H
     --->
     B1,B2,...,Bn
)
```

しかし、これは等価変換になるとは限らないことに注意する必要がある。

例えば、節が次のように定義されているとする。

```
(path (*rule . *path) *a *z) ←
(simplify *rule *a *b),
(path *path *b *z)
```

これによる展開操作は、等価変換ルールによって次のように表される。

```
(Rule
  (true)
  (path (*rule . *path) *a *z)
    --->
  (simplify *rule *a *b)
  (path *path *b *z))
```

これは大部分の場合、うまく働くが、(path *p *a *z) のように path の第 1 引数 (この場合は*p) が変数の場合には、等価変換にはならない。

これを完全な等価変換ルールにするには、次のようにするとよい。

```
(Rule
   (and (not (var *p))
        (= *p (*rule . *path)))
   (path *p *a *z)
    --->
   (simplify *rule *a *b)
   (path *path *b *z))
```

ここで (var *p) は、*p が変数であることを示す。

この変換ルールでは、(not (var *p)) によって*p は変数ではないという条件を課して上記の問題を避けている。

これとは対照的に、*z が変数のとき、強制的に (*a . *z1) に置き換える例が次の等価変換ルールである。

```
(Rule
   (and
       (not (var *y))
       (var *z)
       (= *y (*a . *y1))
       (= *z (*a . *z1)))
   (union *x *y *z)
    --->
   (union *x *y1 *z1))
```

(not (var *z)) ならば、*z が、相手を変えることなく自分が変化するというように自然に (*a . *z1) に変化するが、(var *z) ならば強制的に (*a . *z1) に変えてしまう。但し、この方法を使って論理回路を簡単化する場合もある。これは、上の等価変換ルールで言うと union は、$*x \cup *y = *z$ を意味することから、*y に*a が入っているときには、必ず*z にも*a が入っていることがわかるので、*z が変数のときには上述のように、強制的に変換してやる。

3.5 等価変換ルールの例

本論文で扱う 2 つの変換ルール andAnd ルールと absol ルールについて見てみる。

まず、andAnd ルールについて述べる。

集合オブジェクトによる andAnd 変換ルールのプログラムは、以下のようになる。

```
(Rule
   (== *rule andAnd)
   (simplify *rule
     {(AND *a *r1 {*aa . *rest})
      (AND *b {*aa . *r2} *out) . *rr}
     {(AND *a *r1 *rest)
      (AND *b *r3 *out) . *rr})
      --->
     (union *r1 *r2 *r3))
```

上のルールは*rule が andAnd に等しいときに限って等価変換されるものである。

述語 simplify は 3 つの引数を持っている。

第 1 引数は、*rule で、これはルール名を示す。

第 2 引数は、変形前の回路で、第 3 引数は、変形後の回路を示す。

第 2 引数は、2 つの AND 素子と残りの回路である。

最初の AND 素子は、入力端子が$*r_1$で出力端子が $*a$ で、結線先が (*aa . *rest) の端子分、結線可能であることを示す。

2 つめの AND 素子は、入力端子が (*aa . $*r_2$) で出力端子が *b で、結線先が*out である。ここで、最初の AND 素子の結線先が*aa であることがわかる。

第 3 引数も同様である。

union は、$*r_1 \cup *r_2 = *r_3$を意味する。

次の図 4 は、PAL を用いたときのプログラムに対応して andAnd ルールを図示している。

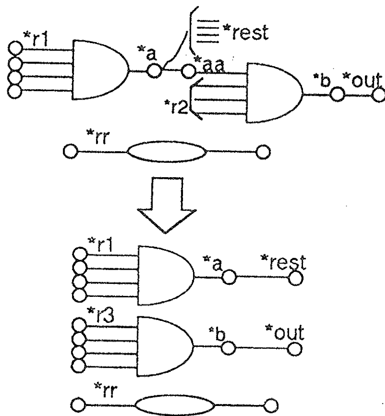


図4　andAndルール

次に、absol 変換ルールについて述べる。

absol 変換ルールは、式で表すと、次のようになる。

$$(A \wedge B) \vee A = A$$

次に、集合オブジェクトによる absol 変換ルールのプログラムは、以下のようになる。

```
(Rule
    (== *rule absol)
    (simplify *rule
      {(AND *d {*a . *r1}
                {*dd . *rest})
       (OR *m {*a *dd} *out) . *rr}
      {(AND *d {*a . *r1} *rest)
       (EQ *m *a *out) . *rr})
       --->)
```

上のルールは*rule が absol に等しいときに限って等価変換されるものである。

(EQ *m *a *out)

の EQ は、イコール素子名であり、入力端子が*a で、出力端子が*m で、一本の線でつながっていることを示す。結線先は*out である。

次の図 5 は、PAL を用いたときのプログラムに対応して absol ルールを図示している。

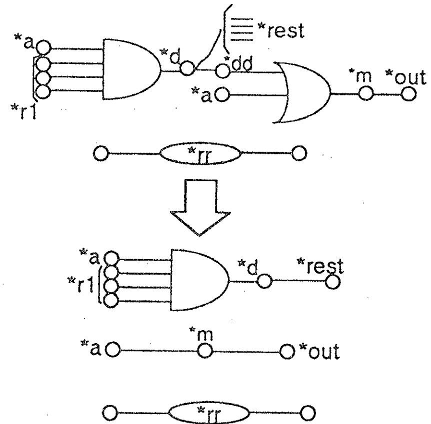


図5　absolルール

4　等価変換による回路変換ルールの合成

4.1　等価変換による回路変換ルールの合成法

r1,r2 をそれぞれ変換ルールとするとき、それら 2 つの変換ルールを合成するには、Head(H_1), $Body(B_1)$を次のように定める。

$$H_1 = B_1 = (path\ (r_1\ r_2)\ *a\ *z)$$

そして、$H_1 \leftarrow B_1$を等価変換すればよい。

それによって*a,*z が具体化され、初期回路と最終回路になる。

4.2　等価変換による回路変換ルールの合成のアルゴリズム

等価変換による回路変換ルールの合成ルールを求めるアルゴリズムについて述べる。

入力として (path (r_1 r_2) *a *z) が与えられる。

入力として与えられた clause が、複数の等価変換ルールの '変換前の論理積'(PreConjunction) と照合され、マッチした場合は、ユニフィケーションが行われ、マッチした等価変換ルールの '変換後の論理積'(PostConjunction) が次の入力として与えられる。

この操作が繰り返されて、入力とする clause が無くなった時点で処理を終了する。

このアルゴリズムをプログラム化すると次のようになる。

```
(as (reduce * *Z)
    (init)
    (Reduce * *Z)
    (pprint (head= * body= *Z)))
```

上のプログラムは、入力として変数*に (path (r_1 r_2) *a *z) が与えられて、等価変換によって回路変換ルールが自動合成されていくおおもとのプログラムであり、次に示す述語 Reduce に処理が移る。

```
(as (Reduce *prog *progz)
    (append *head (*before . *tail)
                        *prog)...(1)
    (rule *cond (*before1) --->
                       *afters)...(2)
    (matchOK? *before *before1 *cond)
                               ...(3)
    (cut)
    (append *head *tail *progb1)
    (append *progb1 *afters *progb)
    (cut)
    (if (not (= (MEMBER . ?) *before))
                 (set memberFlag on))
    (Reduce *progb *progz))
(as (Reduce *prog *prog))
```

上のプログラムを簡単に説明する。

(2) で、Rule とのマッチングをさせており、*before が*before1(PreConjunction) とマッチして、しかも条件部*cond を実行して成功したときに限り、*afters(PostConjunction) に変換される。

(1) の*before と (2) の*before1 とが (3) の下に示す述語 matchOK? のところでユニフィケーションが行われる。

次に matchOK? のプログラムを示す。

```
(as (matchOK? *before *before1 *cond)
    (copy (*before *before1))
    (= *before *before1)
    *cond
    (pprint (before= *before))
    (pprint (before1= *before1))
    (rind "matchOK? (y/n) " y))
```

上のプログラムで、rind は y を入力したときに限って処理を実行することを意味する。

4.3 等価変換による回路変換ルールの合成例

このセクションでは等価変換によるルールの合成例を述べる。

ここでは、andAnd ルールと absol ルールとを合成する。

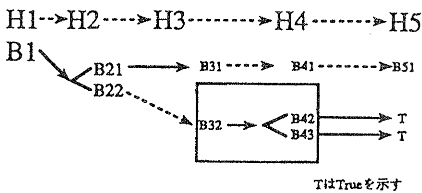
等価変換によって Head,Body は次のように推移する。

まず、H1,B1 からスタートする。

```
H1=[(path (*rule . *path) *1 *2)]
B1=[(path (*rule . *path) *1 *2)]


H2=[(path (andAnd . *path) *1 *2)]
B2=[B21=(simplify andAnd
         {(AND *3 *4 {*5 . *6})
          (AND *7 {*5 . *8} *9) . *10}
         {(AND *3 *4 *6)
          (AND *7 *11 *9) . *10})
    B22=(path (absol)
         {(AND *3 *4 *6)
          (AND *7 *11 *9) . *10} *2)]


H3=[(path (andAnd . *path)
         {(AND *3 {*12 . *13}
                  {*20 *21 . *15})
          (AND *7 {*20 . *19} *9)
          (OR  *16 {*12 *14} *17)
                            . *18} *2)]
B3=[B31=(union {*12 . *13} *19 *11)
    B32=(path (absol)
         {(AND *3 *4 *6)
```

```
            (AND *7 *11 *9) . *10} *2)]


H4=[(path (andAnd absol)
          {(AND *3 {*12 . *13}
                      {*20 *21 . *15})
           (AND *7 {*20 . *19} *9)
           (OR  *16 {*12 *14} *17)
                             . *18} *2)]
B4=[B41=(union {*12 . *13} *19 *11)
    B42=(simplify absol
          {(AND *3 {*12 . *13}
                          {*14 . *15})
           (OR  *16 {*12 *14} *17)
           (AND *7 *11 *9) . *18})
    B43=(path ()
          {(AND *3 {*12 . *13} *15)
           (EQ *16 *12 *17)
           (AND *7 *11 *9) . *18}]


H5=[(path (andAnd absol)
          {(AND *3 {*12 . *13}
                      {*20 *21 . *15})
           (AND *7 {*20 . *19} *9)
           (OR  *16 {*12 *14} *17)
                                 . *18}
          {(AND *3 {*12 . *13} *15)
           (EQ *16 *12 *17)
           (AND *7 *11 *9) . *18})]
B5=[(union {*12 . *13} *19 *11)]
```

この推移は、次の図6のようになる。



但し,ここで ──▶ は,置き換えを示し,----▶ はそれに伴うユニフィケーション効果による影響を示す

図6　推移変化

ここで実線の矢印は、節の置き換えを示し、破線の矢印は、それに伴うユニフィケーション効果による影響を示す。この破線の矢印の動きを path の第2引数 (初期回路) と、第3引数 (最終回路) で見てみると、次のようになる。

```
*1 (H1) → *1 (H2) →
{(AND *3 {*12 . *13} {*20 *21 . *15}) →
 (AND *7 {*20 . *19} *9)
 (OR  *16 {*12 *14} *17) . *18} (H3,H4,H5)


*2 (H1) → *2 (H2) → *2 (H3) → *2 (H4) →
{(AND *3 {*12 . *13} *15)
 (EQ *16 *12 *17)
 (AND *7 *11 *9) . *18} (H5)
```

明らかに、節の置き換えに伴うユニフィケーション効果による影響が現れていることがわかる。

そして、andAnd ルールと absol ルールとの合成結果は、

$$H5 \leftarrow B5$$

となる。

この合成結果を図で表すと、次の図7のようになる。

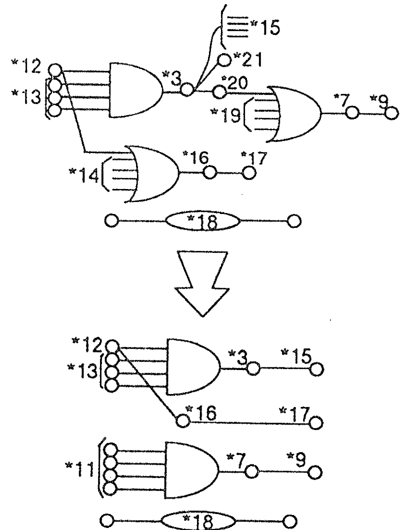


図7　合成結果

5　むすび

本論文では、いくつかの等価変換ルールを示し、それを用いることによって、回路変換ルールを自動合成した。等価変換は、従来の Horn 節の Head を Body に展開する方法に比べると、1つの節は等価な1つの節に変換されるので、節の増大が抑制され

る。このことから合成するうえで、効率よく処理が行われた。

今後の課題は、次のようなことである。

回路変換列が与えられたときに、この具体例に対応した一般的な合成ルールを求める。このときに、論理回路が素子の集合で表現されることより、複数の素子からなる回路どうしのユニフィケーションは、複数考えられる。このことから適用ルール名の列は複数現れ、これから求められる合成ルールも複数になる。その中から回路変換列の具体例に対応した合成ルールを選ぶ。

さらに、今回の研究では、'変換前の論理積'を構成する論理式は、1つで取り上げたが、今後、複数の論理式を扱うケースを考える。

文 献

[1] Hiroshi Mabuchi,Kiyoshi Akama,Yoshinao Aoki:一般化論理プログラムの部分計算による論理回路の変換ルールの合成法, 情報処理学会, 知識のリフォーメーションシンポジウム論文集,pp.129-136 (1991.11)

[2] kiyoshi Akama:問題解決知識の学習, 知識プログラミングシンポジウム, 日本ソフトウェア科学会,(1989)

[3] Dejong,G.and Mooney,R.:Explanation-based Learning:An Alternative View,MachineLearning vol.1,No.2,pp.145-176(1986)

[4] Mitchell,T.M.,Keller,R.and Kedar-Cabelli,S. Explanation-Based Generalization:A unifying view,Machine Learning,vol.1,No.1,pp.47-80 (1986)

[5] Shavlik,J.W.and DeJong,G.F.:BAGGER An EBL Systemthat Extendsand Generalizes Explanations,AAAI-87pp.516-520 (1987)

[6] Michalski,R.S.,Carbonell,J.G.and Mitchell T.M.(eds.):Machine Learning An artificial intelligence approach MorganKaufmann, LosAltos (1983), 邦訳; 知識獲得と学習シリーズ第1-3巻 共立出版 (1987)

[7] Masayuki Numao:説明に基づく学習, 一領域固有の知識を用いたアプローチー, 人工知能学会誌,vol.3,No.6,pp.704-711(1988)

[8] Kiyoshi Akama:意味計算 2,-制約付変数とユーザ定義オブジェクト-自然言語処理研究会資料,74-1, (1989)

[9] Kiyoshi Akama:SufficientConditions of Two Inference Rulesfor Generalized Logic Programs-The Logic Programming Conference '91pp.161-170, (1991)