社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2005－ＳＬＤＭ－119 (24)
2005／3／18

第一階述語論理の決定可能なサブクラスに対する同値制約を考慮した充足可能性判定

小澤 弘明† 浜口 清治†† 柏原 敏伸††

† 大阪大学基礎工学部 〒560-8531 大阪府豊中市待兼山町 1-3
†† 大阪大学大学院情報科学研究科 〒560-8531 豊中市待兼山町 1-3
E-mail: †h-kozawa@ics.es.osaka-u.ac.jp, ††{hama,kashi}@ist.osaka-u.ac.jp

あらまし 近年の大規模化, 複雑化した半導体回路の検証の効率化のために, 限量子を含まない等号付第一階述語論理の充足可能性判定を用いた形式的検証手法が提案されている. しかし, 限量子を含まない等号付第一階述語論理では, 関数記号や述語記号に特別な意味づけが行われないため, そのままでは意図した通りの結果が得られない事がある. 本報告では, 同値制約と呼ぶ式の集合を与える事により, 関数や述語の性質を考慮して充足可能性判定を行うアルゴリズムを示す. また, このアルゴリズムを実装し, 実験した結果について報告する.

キーワード 限量子を含まない等号付第一階述語論理, 充足可能性判定, 同値制約

Satisfiability Checking with Equivalence Constraints for Decidable Subclass of First-Order Logic

Hiroaki KOZAWA†, Kiyoharu HAMAGUCHI††, and Toshinobu KASHIWABARA††

† School of Engineering Science, Osaka University 1-3 Machikaneyama-cho, Toyonaka-shi, 560-8531, Japan
†† Graduate School of Infomation Science and Technology, Osaka University 1-3 Machikaneyama-cho, Toyonaka-shi, 560-8531, Japan
E-mail: †h-kozawa@ics.es.osaka-u.ac.jp, ††{hama,kashi}@ist.osaka-u.ac.jp

Abstract For formal verification of large-scale digital circuits, a method using satisfiability checking of logic with equality and uninterpreted functions has been proposed. This logic, however, does not consider the specific properties of functions and predicates, e.g. commutative property of addition. In order to ease this problem, we introduce "equivalence constraint" that is a set of formulas representing the properties of functions and predicates, and check the satisfiability of formulas under the constraints. In this report, we show an algorithm for checking satisfiability with equivalence constraints and some experimental results.

Key words logic with equality and uninterpreted function, satisfiability checking, equivalence constraints

1. はじめに

近年の半導体回路の大規模化, 複雑化により, 回路の検証にかかるコストが問題視されている. 回路設計の検証には, 従来はミュレーションが行われていたが, 近年ではより厳密な検証結果を得られる形式的な手法も用いられるようになってきている. 例えば, 組み合わせ回路や順序回路の検証では, 命題論理に対する充足可能性判定などが用いられている. しかし, 大規模な回路については論理回路レベルでの検証は依然として困難である. 形式的検証をより短時間でより容易に行う手法として, 論理回路レベルではなく, 機能レベルでの検証が考えられている. 具体的には, 等価性判定において算術演算回路をブラックボックス化して関数記号などによって表現し, 抽象化したまま扱う手法が考案されている.

関数記号を扱うことのできる論理体系として, 第一階述語論理があるが, 例えば２つの記述に違いがあるかを判定する充足可能性判定は決定不能である. これに対して, 機能レベルでの検証を行うために, 第一階述語論理のサブクラスである限量子を含まない等号付第一階述語論理 (Logic with Equality and Uninterpreted Functions, 以下 EUF と記す) を用いることが提案されている. EUF には充足可能性を判定するアルゴリズムが存在するため, 計算機による自動検証が可能であり, パイプラインプロセッサやアセンブリ言語で書かれたプログラムの検証に用いられている [7] [8]. EUF の充足可能性, または恒真性の

判定器としては, CVC [3] や ICS [4] などがある.

EUF では, 関数記号や述語記号に特別な意味づけがなされない. そのため, 既存のアルゴリズムで論理式の充足可能性を判定すると, 例えばある関数に交換則を仮定すれば充足可能であるような論理式も, 充足不能であると判定してしまうことがある.

これに対して本稿では, 論理式 F の充足可能性判定を行う際に, 同値制約と呼ぶ式の集合を与え, そのパターンにマッチする式は常に真であると仮定する. 例えば, 同値制約として $f(X,Y) = f(Y,X)$ を与えると, 関数 f について交換則が成立しているという条件の下で論理式の充足可能性判定が行われる.

本稿では, EUF の論理式 F を和積標準形に変換し, 同値制約を考慮しつつ, 命題論理に対する充足可能性判定 (SAT) の手法を応用して充足可能性判定を行うアルゴリズムを示す. また, このアルゴリズムを実装し, 同値制約を与えた場合の実験結果を示す.

以下, 2 章では本稿で用いる用語の定義, 及び, 提案手法に用いる既存手法の説明を述べ, 3 章では提案する充足可能性判定アルゴリズムの説明をそれぞれ行い, 4 章では実験結果について述べる.

2. 準　　備

2.1 限量子を含まない等号付第一階述語論理

限量子を含まない等号付第一階述語論理 (EUF) とは, 一般の第一階述語論理から限量子を除き, 特別な記号として等号を含むものである. EUF の構文は, 項と論理式からなる.

[定義] 構文

項は, 変数, n 個の項を引数にとる関数, および ITE 項からなる. ITE 項 とは, C が論理式, t_1, t_2 が項の時の $ITE(C, t_1, t_2)$ を指す. ITE 項 は $if - then - else$ を表し, C が真の時は t_1 を, C が偽の時は t_2 を表す. **論理式**は, $true, false$, 論理変数, 項同士の等式, n 個の項を引数にとる述語, および, A,B が論理式, $\circ$ が論理演算子の時, $\neg A, A \circ B$ からなる.

□

[定義] 意味

空でない領域 D と, 関数記号, 述語記号, 及び変数記号の解釈 I が与えられると, 論理式の真偽が決定する.

解釈 I は, 引数を k 個持つ関数記号に, 関数 $D^k \to D$ を, 引数を k 個持つ述語記号に, 関数 $D^k \to \{true, false\}$ を, 変数記号に, D の要素を, 論理変数に, $\{true, false\}$ の要素をそれぞれ割り当てる.

□

[定義] 値の評価

項 t, 論理式 F の, 解釈 I に対する評価値 $I(t), I(F)$ は, 以下のように再帰的に定義される.

項 $t = f(t_1, t_2, \cdots, t_n)$ に対して, $I(t) = I(f)(I(t_1), I(t_2), \cdots, I(t_n))$. 項 $t = ITE(F, t_1, t_2)$ に対して, $I(F) = true$ ならば, $I(t) = I(t_1)$. $I(F) = false$ ならば, $I(t) = I(t_2)$. 論理式 $F = p(t_1, t_2, \cdots, t_n)$ に対して, $I(F) = I(p)(I(t_1), I(t_2), \cdots, I(t_n))$. 等式 $F = (t_1 = t_2)$ に対して, $I(t_1) = I(t_2)$ ならば, $I(F) = true$. $I(t_1) \neq I(t_2)$ ならば, $I(F) = false$. 論理演算子 $\circ$, 論理式 $F = F_1 \circ F_2$ に対して, $I(F) = I(F_1) \circ I(F_2)$.

□

[定義] 充足可能性と充足不能性

論理式 F が充足可能であるとは, $I(F)$ が真となるような解釈 I と領域 D が存在する事を意味する. 論理式 F が充足不能であるとは, $I(F)$ が真となるような解釈 I と領域 D が存在しない事を意味する.

□

次に, EUF における和積標準形について定義する.

[定義] 和積標準形 (Conjunctive Normal Form. 以下 CNF と記す)

原子式は, 等式, 述語, または論理変数からなる. **リテラル**は, 原子式, またはその否定からなる. **節**は, 1 つ以上のリテラルの論理和である. すなわち, $L_1, L_2, \cdots, L_n$ がリテラルのとき, $L_1 \vee L_2 \vee \cdots \vee L_n$ は節である. **和積標準形の論理式**は, 1 つ以上の節の論理積である. すなわち, $C_1, C_2, \cdots, C_n$ が節のとき, $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ は和積標準形の論理式である.

□

論理演算子の性質から, EUF の任意の論理式は, CNF に変換する事ができる. 以下, 論理式といえば, 一般的な形の論理式と明記しない限り, CNF であるとする.

2.2 同値制約

[定義] 同値制約 Q

論理式 F, 項 t に含まれる変数の集合を, それぞれ $VAR(F), VAR(t)$ と表すとする. **同値制約** Q は等価な関数または述語を表した論理式の集合であり, 次のように定義する.

$$
\begin{aligned}
Q = \{ & t_1 = t_2 | t_1, t_2 \text{は関数記号で始まる項であり,} \\
& \text{かつ,} VAR(t_1) = VAR(t_2)\} \\
\cup \{ & p_1 \Leftrightarrow p_2 | p_1, p_2 \text{は述語記号で始まる論理式} \\
& \text{であり, かつ,} VAR(p_1) = VAR(p_2)\}
\end{aligned}
$$

□

同値制約中の各式は, 関数間, または述語間の同値関係を示しており, 左辺と右辺が常に同値であることを表す. 論理式 F の充足可能性判定の際には, 同値制約中の各式に含まれる変数は, F に含まれる部分項にそれぞれ置き換えられて判定される.

次に, 論理式 F に対して実際に同値制約 Q が表す式の集合を定義する. これを, Q の F に関するインスタンス と呼ぶ.

[定義] Q の F に関するインスタンス $Ins_F(Q)$

論理式 F に現れる全ての部分項の集合を $\mathcal{E}(F)$ とする. この時, Q の F に関するインスタンス $Ins_F(Q)$ は以下のように定義される.

$$
\begin{aligned}
Ins_F(Q) = & \{ t_1 = t_2 | t_1, t_2 \text{は次の 1, 2 を満たす} \} \\
& \cup \{ p_1 \Leftrightarrow p_2 | p_1, p_2 \text{は次の 3, 4 を満たす} \}
\end{aligned}
$$

(1) t_1 または t_2 が F 中に部分項として存在する.

(2) t_1, t_2 は, ある $s_1 = s_2 \in Q$ について, s_1 と s_2 の変数に $\mathcal{E}(F)$ の要素を代入したものである.

(3) p_1 または p_2 が F 中に原子式として存在する.

(4) p_1, p_2 は, ある $s_1 \Leftrightarrow s_2 \in Q$ について, s_1 と s_2 の変数に $\mathcal{E}(F)$ の要素を代入したものである.

□

[定義] $F|_Q$

同値制約 Q を仮定した上での論理式 F を $F|_Q$ と表すとする.

$F|_Q$ は次のように定義される.

$$F|_Q \equiv \left(\bigwedge_{G \in Ins_F(Q)} G \right) \Rightarrow F$$

□

同値制約 Q のもとで論理式 F の充足可能性判定を行う事は, $F|_Q$ の充足可能性判定を行う事を意味する. 例えば, $Q = \{f(X,Y) = f(Y,X)\}, F \equiv g(f(a,b)) = g(f(b,a))$ とすると, $Ins_F(Q) = \{f(a,b) = f(b,a)\}$ である. この時, $F|_Q$ の充足可能性判定は, $Ins_F(Q)$ 中の全ての式を真と仮定した上での F の充足可能性判定, すなわち, f が交換則の成り立つ関数記号であるという条件の下での充足可能性判定を意味する.

しかし, $F|_Q$ を定義通りの形で充足可能性判定を行うことは非現実的である. なぜなら, $\mathcal{E}(F)$ が, 構成の仕方から F の項数に対して指数乗の数の要素を含み得るため, $Ins_F(Q)$ の要素数が膨大になる可能性があるからである.

2.3 合同閉包

この節では, 関数, 述語, 変数, 及び論理変数の集合 S に対する合同閉包について述べる. まず, 集合 S に含まれる要素全てに対する解析木に対応するグラフ $G = (V, \Lambda, \delta)$ を導入する.

[定義] グラフ $G = (V, \Lambda, \delta)$

グラフ $G = (V, \Lambda, \delta)$ は, 関数, 述語, 変数, 及び論理変数の解析木に対応するグラフとする. この時, V は節点の集合, Λ は節点を引数にとり, 対応するラベルを返す関数, δ は節点と整数 i を引数にとり, 節点の i 番目の引数を返す関数である.

□

関数, 述語, 変数, 及び論理変数の集合 S を考える. S の各要素の解析木を全て含むグラフを G(S) とすると, 合同閉包は以下のように定義される.

[定義] 合同閉包

関数, 述語, 変数, 及び論理変数からなる集合 S に対して定まるグラフ $G(S)$ について次の条件を満足する $V \times V$ 上の関係 R を $G(S)$ 合同な関係であるという.

1. $(\Lambda(u) = \Lambda(v) \wedge$ 全ての i について $\delta(u,i) = \delta(v,i)) \Rightarrow uRv$
2. R は同値関係

ある関係 E を含む, 最も細分化された合同関係 R を E の合同閉包という.

□

例えば, 集合 $S = \{y, f(x), w, g(y,z), w, g(f(x),z)\}$ に対する $E = \{(y, f(x)), (w, g(y,z))\}$ の合同閉包 R は次のようになる.

$$R = \{(y, f(x)), (w, g(y,z)), (g(y,z), g(f(x),z)), (w, g(f(x),z))\}$$

また, 合同閉包 R は同値関係であるので, 同値類が形成され, 各同値類の代表元を一意に決めることができる.

3. 同値制約を考慮した充足可能性判定アルゴリズム

提案するアルゴリズムは, 一般的な形の EUF の論理式を CNF に変換し, その CNF に対して同値制約を考慮して充足可能性を判定する. 以下に各部分の説明を述べる.

```
level : integer;
F : CNF;
E, D : V×V上の関係;
R : V×V上のの同値関係;

Check_Satisfiability(){
  status : integer;

  level = 0;
  R = V×V上で反射律のみを満たす関係;
  status = deduce();
  if(status == CONFLICT)
    return UNSATISFIABLE;
  else if(status == SATISFIABLE)
    return SATISFIABLE;
  while(true){
    level++;
    decide();
    while(true){
      status = deduce();
      if(status == CONFLICT){
        level = backtrack();
        if(level == 0)
          return UNSATISFIABLE;
      }else if(status == SATISFIABLE){
        return SATISFIABLE;
      }else
        break;
    }
}
```

図 1 充足可能性判定アルゴリズム

3.1 和積標準形への変換

論理演算に対しては中間変数を導入する一般的な方法を用いる. また, ITE 項に対しては中間変数を用いて以下のように処理する.

リテラル C, ITE 項でない項 t, s に対して, ITE 項 $T = ITE(C,t,s)$ を考える. 論理式 F が T を部分項として含むとき, F の中の T を新しい変数 x と取り替えた論理式を F' とすると, F は次のように表される.

$$F \equiv F' \wedge (\neg C \vee x = t) \wedge (C \vee x = s)$$

3.2 充足可能性判定アルゴリズム

充足可能性判定に際して, TRUE, FALSE という特別な定数を考え, すべてのリテラルを等式もしくはその否定で表し, 述語や論理変数を項のように扱う. 例えば, 述語 $p(t1, t2, ..., tn)$ を含むリテラル $\neg p(t_1, t_2, ..., t_n)$ は, 等式 $p(t_1, t_2, ..., t_n) = FALSE$ で表される. 以上を踏まえて充足可能性判定アルゴリズムを説明する.

図 1 に充足可能性判定アルゴリズムを示す. ブール式に対する SAT の解法と構造的にはほぼ同じであるが, 以下のような相違点がある.

- $x = x$ などの割り当てに関わらず真偽が決まるリテラルに真偽値を判定するため, 割り当てを行う前に deduce() を呼び出す. リテラルを一つしか含まない節はもともとユニット節であるため, この時に同時に導出が行われる.
- 等式を扱うために, 構文木に対応するグラフ上の合同閉包を使用する.

図 1 の変数の説明は次の通りである. $level$ は割り当てレベルを表す. 割り当てレベルとは, 割り当ての深さを表す値である. F は, 充足可能性を判定する CNF である. 関係 E は, 割り当て及びユニット節による導出によって同値であると判定された項の組の集合である. 関係 D は, E とは逆に, 割り当て及びユニット節による導出によって同値であってはならないと判定

```
decide(){
  C : 節;
  L : リテラル;
  status : integer;

  C = 真でない節;
  L = C中の真偽未決定なリテラル;
  Lは真であると印を付ける;
  割り当てをレベルと共にスタックに記憶;
  if(Lが肯定){
    E = E ∪ {(Lの左辺, Lの右辺)};
    status = MERGE(R, Lの左辺, Lの右辺);   /* Rを更新 */
    if(status == CONFLICT)     /* MERGE中に矛盾が発生 */
      backtrack();
  }else{
    D = D ∪ {(Lの左辺, Lの右辺)};
  }
}
```

図 2　関数 decide

された項の組の集合である．また，関係 R は E の合同閉包である．以下に，図 1 で用いている各サブルーチンのアルゴリズムとその説明を示す．

関数 decide

図 2 に関数 decide のアルゴリズムを示す．関数 decide では，真でない節中の真偽未決定なリテラル L に真を割り当てる．バックトラックの際に必要となるので，真に割り当てたリテラルを *level* の値と共にスタックに積んでおく．

L が肯定である時，関係 E に (L の左辺, L の右辺) が加えられ，関数 MERGE によって E の合同閉包 R が更新される．関数 MERGE の詳細については後述する．R の更新時に，同値であってはならない項が同じ同値類に入る事がある．このようなときには矛盾が起こったとして関数 backtrack を呼び出す．

一方，L が否定である時，関係 D に (L の左辺, L の右辺) が加えられる．

関数 deduce

図 3 に関数 deduce のアルゴリズムを示す．関数 deduce では，主に次の 3 つの処理を行う．

（1）　リテラルの真偽の判定

真でない節 C に含まれる真偽の未決定なリテラルについて，関係 R 及び関係 D の下で真偽を判定する．C が偽になれば，矛盾が起こったとして CONFLICT を返す．

（2）　ユニット節の処理

ユニット節とは，リテラルが真偽未決定なもの一つ以外全て偽であるような節を指す．C がユニット節になれば，真偽未決定で残っているリテラルに真を割り当て，そのリテラルの肯定・否定に応じて R または D を更新する．R の更新時に矛盾が起これば CONFLICT を返す．

（3）　同値制約との矛盾判定

(1) と (2) の処理を全ての節に対して行い，全ての節が真となった時，関数 check_constraint で現在の R と D が同値制約と矛盾するかを判定し，矛盾するならば CONFLICT を，そうでなければ，充足可能であるので SATISFIABLE を返す．

以上の処理を，矛盾が発生する，もしくは充足解が発見される，もしくはユニット節がなくなるまで繰り返す．

関数 backtrack

図 4 に関数 backtrack のアルゴリズムを示す．関数 backtrack では，次の 2 つの処理を行う．

（1）　割り当てレベルを戻す

```
deduce(){
  C : 節;
  L, unit : リテラル;
  left, right : 項;
  status : integer;

  while(true){
    for(真でない節C){
      for(真偽未決定なリテラルL){
        left = Lの左辺;
        right = Lの右辺;
        if( (left,right)∈R ){
          if(Lが肯定){
            Lは真であると印を付ける;
          }else
            Lは偽であると印を付ける;
        }else if( (left,right)∈D ){
          if(Lが肯定)
            Lは偽であると印を付ける;
          else{
            Lは真であると印を付ける;
          }
          導出結果をlevelと共にスタックに記憶
        }
        if(Cが偽){
          return CONFLICT;
        }else if(Cがユニット節){
          unit = 真偽未決定で残っているリテラル;
          unitは真であると印を付ける;
          if(unitが肯定){
            E = E ∪ {(unitの左辺, unitの右辺)};
            status = MERGE(R,unitの左辺, unitの右辺);
            if(status == CONFLICT)
              return CONFLICT;
          }else{
            D = D ∪ {(unitの左辺, unitの右辺)};
          }
          導出結果をlevelと共にスタックに記憶
        }
      }
      if(全ての節が真)
        /* 同値制約に反するか調べる */
        if(check_constraint() == CONFLICT)
          return CONFLICT;
        else
          return SATISFIABLE;
      else if(ユニット節が存在しなかった)
        return UNDECIDABLE;
    }
  }
```

図 3　関数 deduce

```
backtrack(){
  L : リテラル;

  L = 最もレベルの高い真の割り当て
  level = Lのレベル
  if(level == 0)
    return 0;

  割り当て, 導出, E, R, Dをlevelの状態まで戻す;
  Lは偽であると印を付ける;
  push_to_Assign_stack(L,level);
  if(Lが肯定)
    D = D ∪ {(Lの左辺, Lの右辺)};
  else{
    E = E ∪ {(Lの左辺, Lの右辺)};
    status = MERGE(R, Lの左辺, Lの右辺);
    if(status == CONFLICT)
      backtrack();
  }
  return level;
}
```

図 4　関数 backtrack

割り当てが真である最も新しいレベルまで割り当てレベルを戻し，そのレベル以降の割り当て，導出の結果を取り消す．

（2）　割り当てを反転させる

割り当てレベルを戻したら，そのレベルの割り当てを反転させ，この割り当てを *level* と共にスタックに積む．さらに，この割り当てと，リテラルの肯定・否定から，E, R, D を更新する．もし R の更新時に矛盾が起こったら，さらに関数 backtrack を呼び

```
/* Rに関係(u,v)を追加し，その合同閉包を新たなRとする */
MERGE(R : 合同閉包; u,v : 構文木の節点){
  X,Y : 節点の集合;
  x,y : 節点;
  status : integer;

  if( (u,v)∈D )
    return CONFLICT;

  if( !(FIND(u)=FIDN(v)) ){
    X = { xを引数に持つ関数・述語の節点 | x∈uの同値類 };
    Y = { yを引数に持つ関数・述語の節点 | y∈uの同値類 };
    UNION(R,u,v);
    for( (x,y) | x∈X, y∈Y )
      if( !(FIND(u)=FIND(v))&&CONGRUENT(R,x,y) ){
        status = MERGE(R,x,y);
        if( status == CONFLICT)
          return CONFLICT;
      }
  }
  return OK;
}
```

図 5　関数 MERGE

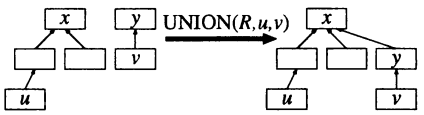

図 6　関数 UNION の処理

出す.

関数 MERGE

合同閉包を求める関数 MERGE を図 5 に示す. このアルゴリズムは, [5] に示されているアルゴリズムに関係 D に関する処理を付け加えたものである. 関数 MERGE は以下のように処理を行う. なお, 同値類に対する処理は UNION-FIND アルゴリズム [10] を用いる.

（1） $(u,v) \in D$ であるかを調べる.

$(u,v) \in D$ である場合, u と v は同値であってはならないので, 矛盾が生じる. よって CONFLICT を返す.

（2） $(u,v) \in R$ であるかを調べる.

uRv が成り立つ場合, R を更新する必要がないので, OK を返す. 関数 FIND は, 引数にとった節点の含まれる同値類の代表元を返す関数であり,

$$\mathrm{FIND}(u) = \mathrm{FIND}(v) \Leftrightarrow (u,v) \in R$$

である.

（3） ある節点 x を引数にとる関数または述語の節点の事を, x の親と呼ぶものとする. u の同値類それぞれに対して親の集合を求め, それらの和集合を X とする. 同様に, v に対する上記の集合を Y とする.

（4） 関数 UNION(R,u,v) によって, uRv が成り立つように R を更新する.

図 6 に関数 UNION の処理を示す. 関数 UNION は, 同値類を併合させる関数である.

（5） $x \in X, y \in Y$ である組 (x,y) それぞれに対して, x と y が R の下で同値でなく, かつ, CONGRUENT(R,x,y) が true を返すとき, MERGE(R,x,y) を呼び出す.

関数 CONGRUENT(R,x,y) は, xRy が成り立つなら true を返す関数である. もし MERGE の返値が CONFLICT だった場合, R の更新時に矛盾が起こったことになるため, CONFLICT を返す.

```
Q : 同値制約;
F : 入力CNF;
R : 合同閉包;

check_constraint(){
  flag, status : integer;
  q : Qの要素;
  varset : 変数の集合;
  left,left',right,right' : 関数または述語の構文木;
  A : 変数の集合に対するε(F)の割り当て;
  t, s : 構文木;

  while(true){
    flag = 0;
    for(q ∈ Q){
      varset = qに含まれる変数の集合;
      left = qの左辺;
      right = qの右辺;
      for( 割り当てA．ただし，割り当てAに従って変数を
           ε(F)の要素と置き換えたleftをleft'とすると，
           Fのある部分項tについてleft' R t が成り立つ){
        right' = 割り当てを適用したright;
        if(Fのある部分項sについて，right' R s が成り立つ){
          if( FIND(t) != FIND(s) ){
            status = MERGE(R, t, s);
            if(status == CONFLICT) return CONFLICT;
            flag = 1;
          }
        }
      }
    }
    if(flag == 0)
      break;
  }
  return OK;
}
```

図 7　関数 check_constraint

関数 check_constraint

関数 check_constraint は, 関数 deduce において充足解が求まった時に, その解を導いた関係 R および D が同値制約に矛盾しないかを調べる. 図 7 にアルゴリズムを示す. 関数 check_constraint は, 同値制約 Q に含まれる各式 q に対して以下のような処理を行う.

（1） q に含まれる変数の集合 $varset$ を求める.

（2） $varset$ に含まれる各変数に対して $\mathcal{E}(F)$ の要素を割り当てる. ただし, この割り当ては次の条件を満たすものとする.

- 割り当てに従って q の左辺に含まれる変数を $\mathcal{E}(F)$ の要素で置き換えた関数 (または述語)$left'$ に対応する構文木を新たに考えるとすると, 合同閉包 R について $left'Rt$ が成り立つような F の部分項 t が存在する.

（3） 上の条件を満たすような割り当てを全て求め, 各割り当てに対して次の処理を行う.

- 割り当てに従って q の右辺に含まれる変数を $\mathcal{E}(F)$ の要素で置き換えた関数 (または述語)$right'$ に対応する構文木を新たに考える時, 合同閉包 R について $right'Rs$ が成り立つような F の部分項 s が存在するならば, 同値制約の定義より $left' = right'$ であるから, $t = s$ でなければならない. よって, $(t,s) \notin R$ ならば, 関数 MERGE(R,t,s) を呼び出して R を更新する. MERGE の返値が CONFLICT であった場合, 充足解を導いた 関係 R および D が同値制約 Q に矛盾しているので, CONFLICT を返す.

同値制約 Q に含まれる全ての式 q に対して, 1. から 3. の処理を R の更新がなくなるまで繰り返す.

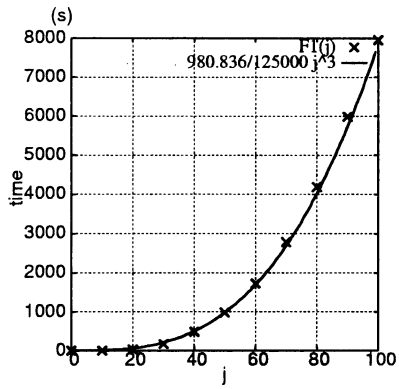

図 8　同値制約を与えた $F_1(j)$ の処理時間

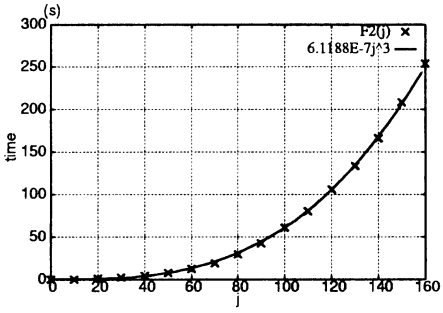

図 9　同値制約を与えた $F_2(j)$ の処理時間

4. 実験結果

3. で述べたアルゴリズムを C 言語で実装し, 実験を行った. 実験環境は以下のようになっている.

- CPU : Pentium 4, 2.53GHz, 512KB(キャッシュ)
- 物理メモリ 249.82MB
- OS : Linux 2.6.0-7
- コンパイラ : GNU C コンパイラ 2.95.3

入力データとして, PARCOR フィルタの計算を行うプログラム parcor.c, および, ADPCM 変換を行うプログラム adpcm.c を用いる [9]. parcor.c, adpcm.c はそれぞれ 40 行と 100 行程度の C 言語のプログラムである. parcor.c, および adpcm.c のそれぞれの for 文を j の値を与えて展開し, それを EUF 論理式で表したものをそれぞれ $P_1(j), A_1(j)$ とする. さらに, $P_1(j)$ 中の+の引数を入れ替えた論理式 $P_2(j)$, $A_1(j)$ 中の $t > s$ を $s < t$ とした論理式 $A_2(j)$ を用意し, $F_1(j) \equiv P_1(j) \oplus P_2(j), F_2(j) \equiv A_1(j) \oplus A_2(j)$ に対して j の値を変えながら実験を行い, 実行に要した時間を測定した. $F_1(j)$ の原子式の数と変数の数はおよそ $1000 \times j, 73 \times j$, $F_2(j)$ の原子式の数と変数の数はおよそ $420 \times j, 27 \times j$ で表される. 入力式 $F_1(j)$ に対して同値制約 $\{X + Y = Y + X\}$ を, $F_2(j)$ に対して同値制約 $\{X > Y \Leftrightarrow Y < X\}$ をそれぞれ与えた場合の提案アルゴリズムの処理時間を図 8, 図 9 に示す. 図の横軸は j の値であり, 縦軸は処理時間である. また, 図 8 の実線曲線は, $j = 50$ の時に処理時間の値 980.836 をとるような j の 3 次関数であり, 図 9 の実線曲線は, $j = 100$ の時に処理時間の値 61.188 をとるような j の 3 次関数である.

同値制約を与えたため, 提案アルゴリズムは $F_1(j), F_2(j)$ を充足不能と判定した. また, 図 8, 図 9 より, 処理時間は j^3 に比例すると考えられる. また, $F_1(j), F_2(j)$ 共に, 式の長さは j に比例するので, 式の長さが n の時, 提案アルゴリズムを実装したプログラムの処理時間は $O(n^3)$ で抑えられると考えられる.

○既存の判定器との比較

入力式 $F_1(j), F_2(j)$ に対して, 同値制約を与えない場合の処理時間を, 既存の判定機である ICS [4] と比較した. j の値は $F_1(j)$ に対しては 0〜100, $F_2(j)$ に対しては 0〜150 の範囲で実験を行った. ICS に対して提案アルゴリズムを実装したプログラムは 10〜100 倍の処理時間を要した. この原因としては, 矛盾が起こった場合の割り当てを保存していないために, 同じ割り当てを原因とする矛盾が繰り返し起こっていることが考えられる. これに対しては, GRASP [1] や Chaff [2] に用いられている矛盾解析の手法を取り入れる事が考えられる.

5. あとがき

本稿では, 第一階述語論理の決定可能なサブクラスである, 限量子を含まない等号付第一階述語論理に対して同値制約を導入し, 関数や述語の性質を考慮した充足可能性判定アルゴリズムを考え, それを C 言語で実装し, 実験を行った.

今後の課題として, 矛盾解析の導入による実行時間の短縮, および, より多くの同値制約を含んだ例へのアルゴリズムの適用が挙げられる. また, 本稿の同値制約では, 結合則を考慮した充足可能性判定を行うことができない. 結合則については, 別途異なるアルゴリズムが必要になると考えられる.

文　献

[1] João P. Marques-Silva, and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Transactions on Computers*, 1999

[2] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik, "Chaff: Engineering an Efficient SAT Solver", *Design Automation Conference*, 2001

[3] Clark W. Barrett, "Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories", *PhD Dissertation. Stanford University*, 2003

[4] Jean-Christophe Filliâtr and Sam Owre and Harald Rueβ and N. Shankar, "ICS: Integrated Canonizer and Solver", *Conputer Science Laboratry*, 2001

[5] Jean H. Gallier, "Logic for Computer Science", *Harper & Row Publishers*, 1986

[6] Pei-Hsin Ho, "Assertion-Based Verification Part B", *ATVA 2004 Tutorials*, 2004

[7] Randal E. Brant, Steven German, and Miroslav N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions", *Computer Aided Verification 1999*, pp.4-10, 1999

[8] D. W. Currie and A. J. Hu and S. Rajan and M. Fujita, "Automatic Formal Verification of DSP Software", *Proceedings of the 37th Design Automation Conference*, pp.130-135, 2000

[9] M. Imai, Y. Takeuchi, N. Ohtsuki, and N. HIkichi, "Compiler Generation Techniques for Embedded Processors and Their Application to HW/SW Codesign", *System Level Synthesis*, pp.293-320, 1999

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "INTRODUCTION TO ALGORITHMS SECOND EDITION", *The MIT Press*, 2001