ソフトウェア工学 86−14
（1992．7．21）

概念データモデルに基づく仕様記述言語コンパイラの処理ネック解決

横田 和久　橋本 正明　佐藤 正和
ATR 通信システム研究所

筆者らは，概念データモデルに基づくプログラムの仕様から手続き的な C プログラムを生成するコンパイラを提案した．このコンパイラは，グラフの閉路解析で生ずる処理ネックを持っている．そのため，本稿ではヒューリスティックなカットセット解析による処理ネックの解決法を示す．このコンパイラは自動的に構造不一致を発見し，解決する．構造不一致はジャクソンの構造化プログラミングの主たる問題である．コンパイラは，すべての閉路上のデッドロックのような構造不一致に関する矛盾を解決しなければならない．この閉路解析は，閉路の数が指数関数的に増加するので処理ネックをもたらす．それゆえ，筆者らは閉路解析をカットセット解析で置き換えた．さらに筆者らは，指数関数的に増大する閉路の数による処理ネックの解決のために，ヒューリスティックな方法をカットセット解析に適用した．実験によりこの処理ネックの解決が，構造不一致の検出精度を落すことなく行なわれたことを確認した．さらに，構造不一致の検出精度を上げるため，グラフの有向枝に持たせていた同期性を，二つの有向枝間の関係として持たせた解析法を適用した．

A Heuristic Cutset Analysis for Solving the Performance Bottleneck of a Conceptual Data Model-Based Language Compiler

Kazuhisa YOKOTA　Masaaki HASHIMOTO　Masakazu SATO

ATR Communication Systems Research Laboratories

2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan

The authors have already proposed a compiler for generating a procedural C program from a conceptual data model-based program specification. The compiler has a performance bottleneck caused by the circuit analysis of a graph. Therefore, this paper proposes a heuristic cutset analysis method for solving the bottleneck. The compiler automatically detects and solves structure clash, which is one of the main concerns of Jackson's Structured Programming. The compiler must solve a kind of inconsistency concerning the clash, like deadlock, on every circuit. This circuit analysis causes the bottleneck since the number of circuits increases exponentially. Therefore, the authors replace the circuit analysis by a cutset analysis. Moreover, the authors apply a heuristic method to the cutset analysis for solving the bottleneck since the number of cutsets also increases exponentially. An experiment has proved that the bottleneck is solved without decreasing clash detection accuracy.

1 はじめに

ソフトウェアの生産性と信頼性は，現在のソフトウェア工学の大きな問題である．ソフトウェアの再利用は，生産性と信頼性を改善する [1]．COBOL のような手続き的プログラミング言語のコードの再利用は実際それらを改善してきた．最近，領域知識や開発経験，仕様のようなソフトウェア開発知識の再利用の研究が増加している [2]．

筆者らも仕様の再利用を研究している [3]．我々が仕様を再利用するとき，その仕様を理解し，新しい要求に合わせるため拡張しなければならない．これは，再利用可能な仕様は理解性と拡張性を持っていなければならないことを意味する．それゆえ，筆者らは再利用可能な仕様を記述するために *PSDL (Program Specification Description Language)* [4, 5] を提案した．

PSDL は直接コンピュータで実行することはできない．そのため，筆者らは，バッチ処理環境 [4] やリアルタイム処理環境 [6] の PSDL プログラム仕様から手続き型言語 C のプログラムを生成する PSDL コンパイラの研究を続けている．前述の環境で最適なプログラムを生成するために，コンパイラはジャクソンの構造化プログラミング [7] の主な問題である構造不一致を発見し，解決する．しかしながら，構造不一致の自動的な発見と解決の研究はほとんど報告されていない [8]．PSDL においては，構造不一致の発見と解決のために，プログラム仕様から得られた有向グラフを解析する．そして，デッドロックのような構造不一致に関する矛盾をグラフ中のすべての閉路について発見し解決する．

しかし，最悪の場合，閉路の数は指数関数的に増加する [9] ので，処理ネックが生じる．実際，前述の PSDL コンパイラ [4] はすべての閉路を直接解析するので処理ネックをもつ．領域知識に基づく計算量を減少させるヒューリスティックな手法は，このような問題に対して有効である．しかし，すべての閉路の無矛盾性を保証しなければならないため，ヒューリスティックな手法はこの閉路解析には適用できない．ところで，閉路解析はカットセット解析で置き換えることができる．さらに，カットセット解析にはヒューリスティックな手法を適用できる．しかし，グラフのカットセットの数も指数関数的に増加する．

この論文の主な目的は，ヒューリスティックなカットセット解析法を提案することである．また，構造不一致の検出精度を上げるため，グラフの有向枝に持たせていた同期性を，二つの有向枝間の関係として持たせた解析法も提案する．筆者らはこの手法を新しい PSDL コンパイラに実装し，評価を行なった．第 2 章と第 3 章で PSDL，構造不一致の発見とその解決法を概説する．第 4 章ではヒューリスティックなカットセット解析について述べる．第 5 章では同期性解析対象の改善について述べる．第 6 章では実験結果を示し，その手法について考察する．

2 PSDL

PSDL のプログラム仕様はそれぞれ図 1の INFORMATION と DATA, ACCESS で示される，情報層とデータ層，アクセス層から構成される．図は販売管理プログラムの例である．

(1) 情報層

情報層は対象世界の情報構造を記述する．product のような実体型は実体の集合である．各々の実体は対象世界のものやことを表す．E 文でそれぞれの実体型を指定する．EN 文は実体型の実体の数を決める．-50 は実体の数が 50 以下であることを指定する．それぞれの実体の性質は name や price のような属性値を集めたもので表される．それぞれの属性は A 文で記述される．STR や NUM は属性の文字と数の値域を区別する．K は主キー属性を指定する．

sold や buy のような関連型は，実体を関係付ける関連の集合である．関連型は R 文で定義される．C 文は関連で関係付けられる実体を記述する．たとえば，sold は product と sale を関係付ける．RN 文は，一つの実体につながる関連の数を決める．M はその数が 0 以上を意味する．1 はその数がちょうど 1 であることを意味する．たとえば，product の一つの実体は sold の関連を 0 または 1 以上持つことを意味する．

プログラム中の計算は，以下の従属性制約で指定される．AD(属性値従属性制約) は，関連によって関係付けられた実体から属性値を得る．RD(関連存在従属性制約) は，関連によって関係付けられなければならない実体から関連を得る．ED(実体存在従属性制約) は，他の実体から実体を得る．そのとき，それらの実体間の関連も同時に得られる．= 文は AD を定義する．例えば，sale の amount は同じ sale の quantity と，関連 sold によって関係付けられた product の price をかけることによって得られる．customer の total は，関連 buy によって customer と結びつけられた sale の amount を合計することによって得られる．図 1には RD と ED の例は無い．

(2) データ層とアクセス層

データ層では，種々のデータ型を記述することによって，product_data のような入出力のデータ構造を指定する．product_name のような基本データ型を記述し，それは，データの形式を C と同じ方法で指定する．product_data や product_record のような繰返し集団データ型や連接集団データ型は，図 1に示されるように I 文や O 文で定義される．S で定義される選択集団データ型は図 1には示されていない．集団データ型は他のデータ型から構成されるが，構成要素のデータ型の中に集団データ型があれば，それを G 文で指定する．文中の ON 句は集団データ型の終了条件を指定する．IX 文は繰返しデータの指標を指定する．= 文は，情報層とデータ層の情報制約を指定する．情報制約は，基本データ型が属性値，実体，関連のどれと結び付くかを決める．

アクセス層は，入出力ファイルへのアクセス方法を指定する．入力か出力かの区別，各々のファイルのレコー

```
INFORMATION
  E product
      EN -50
    A name  STR
     K
    A price NUM
  R sold
    C .product
      RN M
    C .sale
      RN 1
  E sale
      EN -100
    A number   NUM
     K
    A quantity NUM
    A amount   NUM
      = .sold..product.price * quantity
  R buy
    C .customer
      RN M
    C .sale
      RN 1
  E customer
      EN -50
    A name  STR
     K
    A total NUM
      = ASUM(.buy..sale.amount)
DATA
  I product_data
       IX product_id
      G product_record ON EndOfFile(product_data)
    O product_record
        %12s product_name
          = product.name
        %8d  product_price
          = product.price
  I sale_data
       IX sale_id
      G sale_record ON EndOfFile(sale_data)
    O sale_record
      %4d  sale_number
        = sold..sale.number
        = buy..sale.number
      %16s sale_customer
        = buy..customer.name
      %12s sale_product
        = sold..product.name
      %4d  sale_quantity
        = sale.quantity
  I account_data
       IX account_id
      G account_record ON PEntityNumber(sale)
    O account_record
      %4d  sale_number
        = sale.number
        = buy..sale.number
      %8d sale_amount
        = sale.amount
      %16s sale_customer
        = buy..customer.name
      %10d customer_total
        = buy..customer.total
ACCESS
  D product_file INPUT  20 product_data
  D sale_file    INPUT  36 sale_data
  D account_file OUTPUT 38 account_data
```

図 1: PSDL プログラム仕様

ドの長さ，各々のファイルが含むデータの名前はD文により指定する．

3 構造不一致の検出と解決

この節では構造不一致とその検出方法を述べる．また，以下のような制限のもとでの解決方法を述べる．1) 入出力ファイルは順アクセスファイルである．また入力レコードはソートされていない．2) 全てのレコードは同じ構造を持つ．3) 各々のレコードは，同じ型の中で高々一つの実体と一つの関連を表す．

3.1 構造不一致

第2章の中で触れた例は，Cのような手続き型言語では以下のようにプログラムされる．売上金額を計算するために，製品価格と売上数量がproductとsaleのレコードから参照される．このとき，productとsaleは同じ製品名を持つ．しかし，それらのレコードは，ソートされていないという仮定から，同期して入力することはできない．この非同期がproductとsalesデータ間の構造不一致と呼ばれる．この不一致は非同期性を見つけることによって検出できる．

もし，全てのproductレコードがプログラムで定義されたテーブルに含まれるとすると，productテーブルは例えば配列変数で定義され，productレコードはそのテーブルに格納される．それで，一つのsaleレコードが入力されると，売上金額は同じ製品名を持つproductレコードを取り出した後計算される．この操作は全てのsaleレコードについて繰り返される．よって，不一致は解決される．

もちろん，saleレコードは顧客名でソートされてないと仮定されているので，顧客合計はテーブルに格納されなければならない．さらに，顧客合計は売上レコードが全て入力された後得られるので，売上番号と金額は顧客合計と共に出力されるためにテーブルに格納されなければならない．しかし，売上数量は格納される必要がない．

3.2 構造不一致の検出

構造不一致は，PSDLプログラム仕様から作られる有向グラフを局所的，大局的に解析することにより，検出される．

(1) 有向グラフの作成

グラフの節点はPSDL文に従て集められる．節点は構造節点と制約節点に分けられる．実体型，非主キー属性，関係型，データ型，データセット型節点は構造節点である．第2章で述べた制約に相当する節点は制約節点である．以下の例外に注意しなければならない．実体型節点は，実体型だけでなく主キー属性も表す．上で述べた2)と3)の制限により，データ型節点は集団繰返し型によってのみ集められ，同じ集団繰り返しデータ型をもつ情報層とデータ層の情報制約は一つの節点で指定される．

グラフの有向枝は構造節点と制約節点の間に張る．枝の方向は，情報層では実体や，属性値，関連が制約から

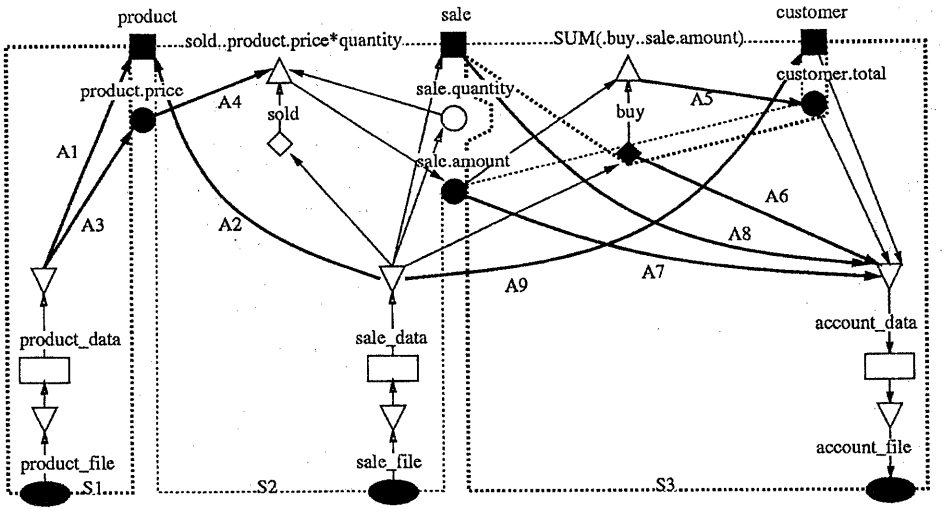

図 2: 有向グラフ

参照される方向と，制約から得られる方向に合わせ，データ層とアクセス層ではデータが入力用データセット型から流出する方向と，出力データセット型へ流入する方向に合わせる．

(3) 同期型有向枝

各々の入力ファイルは複数のレコードを含んでいる．例えば，product_file は TELEVISION レコード，VCR レコードのような複数の product_record を含んでいる．それらのレコードはプログラムによって同じ処理を受ける．それゆえ，同じ型の実体，属性値，関連，データが枝に沿って流れる．例えば，複数の属性値 product.price が図 2枝 *a4* 上を流れる．

この論文では，枝は同期型と非同期型に分けられる．非同期型枝上を流れる対象の順番は，非同期型有向枝と同じ構造節点につながるどの枝上の順番とも同じでないと考えられる．さらに，全ての対象がその節点に流入した後，その節点の対象は出力枝に沿って参照可能となる．しかし，同じ節点につながる同期型枝上を流れる対象の順番は同じと考えられる．それで，節点に対象が流入した後すぐに各々の対象は参照可能になる．

(4) 局所的解析

非同期型枝はまず局所的解析により発見される．もし，実体型節点や関連型節点が二つ以上の流入枝を持つなら，それらの入力枝は非同期型になる．実体型は非主キー属性を持つことが可能である．それらの非主キー属性節点の入力枝も非同期型になる．実体型の場合は次のようになる．

実体型節点に流入している 2 本以上の枝の各々から，同じ主キー値を持っている実体が 2 個以上流れて来ても，それらは 1 つの実体と見なされる．このため，各々の枝から得られた実体の集合に対して和集合をとらなければならない．和集合をとるには実体の集合の間で主キー値を参照しなければならないが，入力データはソートされていないので，同じ主キー値を持った実体が各々の流入枝から同期してながれてくるとは限らない．そこで，すべての実体を待ち合わせて参照しなければならないので，すべての実体が流れ込んで集合演算が終了した後，実体の属性値を参照できるものとする．このため，上記の実体型節点と非主キー属性節点へ流入している枝の上の流れは，それらの節点を介してつながった他の枝の上の流れと同期しない．図 2では実体型節点 product への 2 本の枝が流入しているので，枝 *a1* と *a2, a3* が非同期型になる．

実体型節点または非主キー属性節点と，属性値従属性制約節点または実体存在従属性制約節点との間にある枝は，実体相互の写像の数量関係を表す RN 文の関連数と対応している．その関連数が 2 以上か M であれば，その枝を非同期型にする．その理由を以下に説明する．なお，関連存在従属性制約については，関連型で対応づけられた実体型どうしの直積集合の全要素へ，関連存在条件を適用しなければならない．このため，同じ実体が 2 個以上の要素の中に出現しうるので関連数を仮想的に M と見なして，上記と同じ方法で同期性の解析をする．

属性値を参照する枝の関連数が 2 以上か M の場合，同じ属性値が何回も参照される．ところが，入力データはソートされていないので，同じ属性値が連続して参照されるとは限らない．そこで，属性値がいつ参照されてもよいように待ち合わせなければならないので，すべての属性値が流れ込んだ後，その枝から参照できるものとする．このため，その枝の上の流れは，実体型節点か主キー属性節点を介してつながった他の枝の上の流れと同期しない．たとえば，図 2では枝 *a4* が関連数 M を持つ

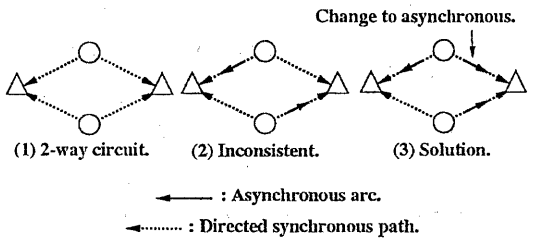

図 3: 閉路による矛盾の検出と解決

ので非同期型になる.

属性値従属性制約によって非主キー属性値が得られる枝の関連数が 2 以上か M の場合，その制約には SUM(合計) のような集合関数が指定されている．集合関数には中間データが必要である．ところで，中間データは何回も更新されるが，入力データはソートされていないので，同じ中間データが連続して更新されるとは限らない．そこで，すべての中間データについて更新がすべて終了した後，その非主キー属性値を参照できるものとする．このため，その枝上のながれは，非主キー属性節点を介してつながった他の枝の上の流れと同期しない．たとえば，図 2では枝 *a5* が関連数 M を持つので非同期型になる.

(5) 大局的解析

局所的解析で得られた結果，他の非同期型枝はグラフの大局的解析により発見される．図 3 (1) に示すような方向混在閉路の上で，図 3 (2) に示すように非同期型有向枝がすべて同じ方向を持っていれば，制約の実行順序に矛盾が起きる．その理由は，非同期型有向枝を持った実体型節点か，非主キー属性節点，関連型節点があれば，その節点にすべての対象が流れ込んだ後でないと制約の実行は先へ進めないためである．この制約実行順序の矛盾を解消するには，図 3 (3) に示すように反対方向の枝を少なくとも 1 本は非同期型に変えればよい.

非同期型へ変える枝の選択は，生成されるプログラムの大局的最適化の問題である．以前の PSDL コンパイラの実験で以下に述べる方法を使った．まず，すべての方向混在閉路を解析して非同期型へ変わる枝の候補を検出する．そのうち，最も多くの閉路の候補になっている枝から順に非同期型へ変える．その結果，図 2では枝 *a5* が非同期型なので，枝 *a6* と *a7*, *a8*, *a9* が非同期型に変わった.

3.3 構造不一致の解決

構造不一致はデータと手続き構造を以下のように決めることにより解決される.

(1) データ構造

グラフの中で非同期型枝を持っている実体型節点と，非主キー属性節点，関連型節点とを非同期型とする．データセット型節点は常に非同期型である．その他の構造節点は同期型とする.

データセット型節点以外の非同期型節点に配列変数を割り当てる．同期型の構造節点と非主キー属性節点，関連型節点にスカラー変数を割り当てる．配列の要素数は EN 文と RN 文に基づいて決める．図 2では節点 `product` と `product.price`, `sale`, `sale.amount`, `buy`, `customer`, `customer.total` へ配列変数を割り当て，節点 `sale.quantity` と `sold` にスカラー変数を割り当てる.

(2) 手続き構造

グラフを，以下のような同期型連結部分グラフに分割する．まず，非同期型節点を部分グラフの境界点に置き，同期型節点は部分グラフの内部に置く．節点を介してつながっている同期型枝は相互に同期しているので，同じ部分グラフの中に置く．制約節点を介して同期型枝につながっている非同期型枝も，その同期型枝と同じ部分グラフに入れる．部分グラフの中では対象の流れがすべて同期する．たとえば，図 2のグラフは 3 つの部分グラフ *S1* と *S2*, *S3* とに分割された.

各々の部分グラフに手続きブロックを割り当てる．ところで大局的解析によって，隣接した部分グラフの境界上の非同期型節点は，その流入枝を一方の部分グラフ中に持ち，他方の部分グラフは流出枝のみを持つことが保証される．これは，隣接した部分グラフへ割り当てられたブロックの間に実行順序があることを示している．このため，ブロックの間に半順序ができるので，この半順序からブロック実行のための全順序を得る．各々の部分グラフの中で，各制約節点と各構造節点へ手続きを割り当てる．これらの節点は有向枝で結ばれているので，手続きの間に半順序がある.

各々の手続きブロックは一つ以上の手続き繰り返しループを含む．そのループの制御要因は部分グラフの中で最初に実行される制約で決まる．たとえば，属性値従属性制約が最初に実行される場合，属性値従属性制約中に一つだけ記述されている関連型の非同期型節点がそれである．その関連型に一つのループを割り当て，その関連型に属する関連毎に実行を繰り返す.

このようにして構造不一致を解決するデータ構造と手続き構造が決定でき，プログラムを生成できた [4].

4 ヒューリスティックなカットセット解析

この章では，以前の PSDL コンパイラにあった閉路解析による処理ネックの問題，閉路解析法を置き換えるカットセット解析法，処理ネックを解決するためのヒューリスティクスの適用について述べる.

4.1 閉路解析の問題

グラフ中の閉路の数は，枝の数の指数で増加する [9]. それゆえ，筆者らは有向グラフ中の閉路を数えた．グラフは PSDL コンパイラを自己記述した PSDL プログラム仕様から得られたものを使った．閉路を数えるプログラムは容易に実装できる簡単なアルゴリズムを用いている．表 1に閉路の数と実行に要した時間を示す．この

表 1: グラフ中の閉路数実測と実測時間

Program	PSDL lines	Arcs	Vertices	Circuits	Count time
BGA	60	26	23	12	0 *sec.*
AGA	71	45	29	5,773	4 *sec.*
GL1	130	72	50	97,249	4 *min.*
MLA	217	109	76	371,682	23 *hr.*
PARA	176	149	73	Unable	> 24 *hr.*
GIC	356	167	124	Unable	> 24 *hr.*

SUN3/80 で実測.

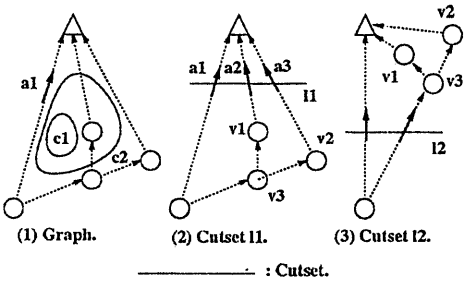

図 4: ヒューリスティックなカットセット解析

表 2: コンパイル時間の比較

Program	Previous compiler Total (T)	Previous compiler Global ana.(G)	Previous compiler G/T	New compiler Total (T)	New compiler Global ana.(G)	New compiler G/T
TP1	15.0	4.5	0.30	9.5	0.1	0.01
TP2	9.1	1.9	0.23	7.7	0.1	0.01
TP3	13.9	2.7	0.19	11.5	0.2	0.02
TP4	15.4	1.0	0.06	14.0	0.3	0.02

コンパイル時間 (*sec.*): SUN3/80 で実測.

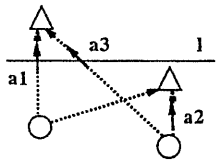

図 5: ねじれた閉路

表が示すように 100 以上の枝を持つグラフ中のすべての閉路を直接解析することは実際不可能である.

自己記述で作成された以前の PSDL コンパイラはたくさんの表を持つ. さらに, このコンパイラは直接すべての閉路を解析している. それゆえ, 膨大な閉路は致命的であり, 処理ネックをもたらす. 実際, このコンパイラは 100 行以上の PSDL プログラム仕様を処理することは不可能だった. もちろん, 大局的解析が表 2に示すコンパイル時間の大部分を占めていた.

4.2 カットセット解析

第 3.2節 (5) で言及した閉路解析法は以下のようにカットセット解析法で置き換えることができる. まずはじめに, 図 4 (1) に示すようにすべての枝を上向きにする. 枝 $a1$ は非同期型と仮定する. そこで, 非同期型枝を含むカットセットを選ぶ. 図 4 (2) において, カットセットは非同期枝を横切る水平線 $l1$ で示される. もし, $a1$ 以外の線を横切る枝 $a2$ と $a3$ も非同期と仮定されるなら, $a1$ を含むすべての閉路 $c1$ と $c2$ の矛盾は節 3.2 (5) で述べた意味でなくなる. その理由は $a2$ と $a3$ は $a1$ と個別の閉路で反対向きになるためである. そのため, 非同期枝を含むカットセットは非同期枝を含むすべての閉路上の矛盾を解決できる. 図 3 (3) に示すようなねじれた閉路は, 図 5において, 線 l 上のすべての非同期型枝 $a1$ と $a2$, $a3$ によって矛盾がなくなる. なぜなら, $a3$ は他の枝と反対向きだからである.

4.3 ヒューリスティクスの応用

グラフのカットセットの数は枝の数の指数で増加する [10]. しかし, 重要なことは非同期型枝を含むカットセットは, 4.2節で述べたように非同期型枝を含むすべての閉路の矛盾を解決できる. 矛盾の解決だけならすべてのカットセットを調べる必要はない. それゆえ, 節 4.1で述べた処理ネックを解決するには, 各々の非同期型枝による矛盾を解決する, より好ましいカットセットを許容される時間内で見つけるだけで十分である.

この論文では, もし vi が aj を含む有向パス上にないなら, 節点 vi は枝 aj の並行節点と呼ぶ. 節点 vj からの流出有向パス上に節点 vi があるなら, vi は vj の上にある節点と呼ばれる. たとえば, 図 4(2) では $v1$ と $v2$, $v3$ は枝 $a1$ の並行節点である. $v1$ と $v2$ は $v3$ の上にある節点である. 処理ネックを解決するためのヒューリスティクスは次のように上で述べたカットセット解析法に導入される.

1. 節 3.2 (4) で述べた局所的解析で見つけた非同期型枝に処理順序をつける. そして, 最初の枝を選ぶ. たとえば, 図 4 (1) において, $a1$ を選んだとする.

2. 非同期枝を横切る水平線を引く. たとえば, l が引かれたとする.

3. 非同期型枝のすべての並行節点を一列に並べる. 並行枝を水平線の下に配置する. これで並行線がカットセットを与える. たとえば, 図 4 (2) では $v1$ と $v2$, $v3$ が $l1$ の下に配置される. $l1$ は $a1$ と $a2$, $a3$ を含むカットセットを与える.

4. カットセット中の同期型枝を数える．たとえば，$l1$ には2つの同期型枝 $a2$ と $a3$ がある．

5. 次の並行節点を選び，それを水平線の上におく．もちろん，並行節点の上の節点は，並行節点より上に配置されなければならない．この操作は新しいカットセットを与える．たとえば，図4(3)において，$v3$ を選べば，カットセット $l2$ が与えられる．ここで，4へ行く．もし，すべての節点がすでに水平線の上に置かれていたら，次に進む．

6. 最も同期型枝が少ないカットセットを選び，それらを非同期型枝に変える．たとえば，図4(3)において，カットセット $l2$ が最も少ない同期型枝を持つので，$a4$ は非同期型に変更される．そこで，局所的解析で得られた次の非同期型枝を選び，2へ行く．もしすべての非同期型枝をすでに処理していたら，この手続きを止める．

ところで，水平線の上位に節点を1つずつ移動させるのではなく，同時に $n(n = 1, 2, 3, \cdots)$ 対ずつすべて移動させれば，水平線上にすべてのカットセットが現れる．しかし，グラフのカットセットの数は指数で増加する．そのため，小さな n に関して，筆者らは既存の PSDL プログラム仕様にツールを適用し，検出精度を調査した．表3で，大局的解析で見つかった非同期型枝の数を，閉路解析とヒューリスティックなカットセット解析で比較した．表では，両方の解析結果は $n = 1$ のときでもほぼ同じ精度を示している．一方，コンパイラの処理時間は大幅に改善された．

表 3: 非同期型枝の数の比較

Program	Arcs	Asyn. arcs after local ana.	Asyn. arcs after global ana.				
			Circuit ana.	Cutset ana.			
				n=1	n=2	n=3	n=4
BGA	26	4	*6*	*6*	6	6	6
AV	31	6	*11*	*9*	9	9	9
GIC	167	58	*58*	*61*	61	61	59
GAI	180	74	*89*	*85*	86	85	85
GII	199	77	*90*	*90*	89	90	90

Count time (SUN3/80): $1sec.$(n=1〜2), $1〜10min.$(n=4).

5 同期性解析対象の改善

以前の PSDL コンパイラはグラフの有向枝へ同期性を持たせて構造不一致を検出したが，この検出法では，構造不一致でないにもかかわらず，構造不一致として検出されるものがある．たとえば，図1の仕様において，に `sale.amount` を得るための AD が `product.name` も参照する場合，方向混在閉路ができるため，実体型 `product` に対する和集合演算で生じた非同期性が大局的解析によって波及し，本来は同期型のままで良い関連型 sold も非同期型となる．この問題を解決するため，同期性を有向枝に持たせるのではなく，同一構造節点につながる2つの有向枝間の関係として持たせた．

(1) 枝の定義の変更

局所的解析の後，構造節点を経由して同じ方向を向いている枝を一つの枝とみなす処理を加えた．この処理ではある構造節点に m 本の流入枝と n 本の流出枝がある場合，その節点がなくなり m×n 本の枝に変わる．このとき，どちらの枝も同期型の時のみ新たにできた枝も同期型とする．さらに，以下の条件を満たす場合，その枝を同期型へ変える．

(2) 非同期型から同期型への変更

以前の PSDL コンパイラでは，局所的解析で非同期型に決められた枝は同期型に変わることはなかった．プログラムの実行効率の点からは，非同期型枝が少ない方がよいので，局所的解析で非同期型に指定された枝の中で，同期型に変更可能な枝を選ぶ処理を追加する．

実体型節点へ複数の枝が流入するときは，和集合をとるために，流入枝のすべてを非同期型に決めていたが，以下の条件を満たす枝は同期型へ変更する．

1. 実体型節点へ流入する枝が，情報制約節点から流出している．

2. かつ，その実体型節点自身またはその実体型の属性節点から流出する枝が，制約節点へ流入している．

3. かつ，その制約節点へ流入する関連の発生元が，1の情報制約節点である．

上記の3つの条件が満たされる場合，情報制約節点から実体型節点へ流入する枝は実体をつくり出すのではなく，実体型のなかの特定の実体を選び出す検索と考えられる．このときにはすでに作られた実体を参照することになるので，実体の待ち合わせは必要ない．そのため実体型へ流入する枝を同期型へ変えても問題はない．ただし，この枝も大局的解析で矛盾の解決のために非同期型へ変更される枝の候補になっている．

今までの解析では，実体型節点へ複数の枝が流入する場合，すべての枝を非同期にしていたので，上記の1,2,3を満たす場合のように同期にできる場合でも，非同期になっていた．このため，実行効率が劣るプログラムが生成されていた．

6 実験と考察

筆者らは，$n = 1$ としたヒューリスティックなカットセット解析法を，以前の実験用 PSDL コンパイラの大局的解析と置き換えて実装した．この新しいコンパイラは，およそ 500 行の PSDL プログラム仕様をコンパイルすることができる．プログラム仕様のコンパイルはおよそ3分である (SUN 3/80 のユーザ時間)．以前のコンパイラがコンパイルできる短いプログラムを使うことによって，大局的解析にかかる時間を以前のコンパイラと新しいコンパイラで比較した結果が表2である．生成

されるCプログラムは以前のコンパイラと新しいコンパイラでは若干違う.

以下に考察と今後の研究課題を述べる.

(1) 処理ネックの解決

このヒューリスティックなカットセット解析の実験で,大局的解析の実行時間の大幅な減少と長いプログラム仕様のコンパイルが許容時間内でできることが確認できた.また,以前のコンパイラの処理ネックが解決した.さらに,構造不一致解決精度が劣らないことも確認できた.

グラフ分割のような他のグラフ処理は,閉路解析のように致命的問題ではない.新しいコンパイラは500行までをコンパイルできた.この制限は自己記述から来ている.それゆえ,この制限は,たとえばCでプログラムすることにより,簡単に解決できる.

(2) ヒューリスティックな知識

カットセット解析法に適用したこのヒューリスティックな知識は, PSDLプログラム仕様から得られる有向グラフの均質性に基づいているように思われる.それゆえ,我々は容易に解決精度を落す例を作れる.しかし,調査は既存のPSDLプログラム仕様が均質であることを示した.近い将来,このヒューリスティクスの知識は十分に検証されなければならない.

(3) 同期性解析対象の改善

同期性の解析対象を有向枝から,同一構造節点につながる2つの有向枝間の関係に変えたことにより,実体の和集合演算と検索を区別できるようになった.このため,データの検索処理を行なう被生成プログラムの実行効率が改善した.さらに処理の形態に依存した同期性の解析方法を見つけることにより,生成されるプログラムの実行効率を改善することは今後の課題である.

(4) プログラムの最適化

プログラム効率の最適化はPSDLコンパイラの大局的解析に依存している.上で述べカットセット解析と閉路解析は,両方とも非同期型枝の数を最小にするだけなので,もちろん最適な解法を与えるわけではない.それゆえ,実行時間と記憶領域を最小化する,さらに洗練された最適化法を将来研究する必要がある.

7 まとめ

PSDLプログラム仕様から得られた有向グラフにおいて,構造不一致の非同期型枝を含むカットセットは,その枝を含むすべての閉路の矛盾を解決する.すべてのカットセットを検査する必要はない.それゆえ,ヒューリスティックなカットセット解析法は,許容される時間内に適切なカットセットを見つけることができる.この方法に適用されるヒューリスティックな知識は, PSDLプログラム仕様から得られるグラフの均質性に依存している.実験は,以前のPSDLコンパイラの処理ネックを解決したことを示した.さらに,実験は構造不一致の解決精度が劣ってないことも示した.また,同期性の解析対象を改善することにより,データの検索処理も同期性解析で自然に解決することができた.

将来,上で述べたヒューリスティックな知識は十分に検証されなければならない.加えて,筆者らは構造不一致の解決精度を向上させるために,別の解析法を研究中である.この解析法は,本論文で述べた方法と融合させなければならない.もちろん,この論文で述べたプログラムの最適化は最良ではない.それゆえ,将来もっと洗練されたヒューリスティックな解析法を研究する必要がある.

謝辞 — 日頃ご指導いただく葉原会長,寺島社長,太田室長に深く感謝いたします.また,ご討論いただいた研究室の諸氏,ならびにPSDLコンパイラの作成にご協力いただいた日本電子計算株式会社の諸氏に感謝いたします.

参考文献

[1] F.P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, *Computer*, Vol. 20, No. 4 pp. 10-19, 1987.

[2] T.J. Biggerstaff, A.J. Perlis (Eds), *Software Reusability*, Vol. I, ACM Press, 1989.

[3] K. Okamoto, M. Hashimoto, Visual Programming with Reusable Specification Described by a Conceptual Data Model- and Constraint-Based Language, *Proc. of HCI*, Vol. 1, pp. 587-591, 1991.

[4] M. Hashimoto, K. Okamoto, A Set and Mapping-based Detection and Solution Method for Structure Clash Between Program Input and Output Data, *Proc. of IEEE COMPSAC*, pp. 629-638, 1990.

[5] K. Okamoto, M. Hashimoto, On Real-Time Software Specification Description with a Conceptual Data Model-Based Language, *Proc. of ICCI*, pp. 186-190. 1990.

[6] K. Okamoto, M. Hashimoto, Program Generation from Real-Time Software Specification Described with a Conceptual Data Model-based Language, *Proc. of SEKE*, pp. 261-270, 1991.

[7] M.A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.

[8] N.S. Prywes, A. Pnueli, Compilation of Nonprocedural Specifications into Computer Programs, *IEEE TOSE*, Vol. SE-9, No. 3, pp. 267-279, 1983.

[9] P. Mateti, N. Deo: On Algorithms for Enumerating all Circuits of a Graph, *SIAM J. Comput.*, Vol. 5, No. 1, pp. 90-99, 1976.

[10] S. Tsukiyama, H. Ariyoshi, I. Shirakawa, Algorithm to Enumerate All the Cutset in $O(|V| + |E|)$ Time per Cutset, *Proc. of ISCAS*, PP. 645-648, 1979.