入出力データの構造不一致検出解決法に関する実験†*

岡本克己†† 橋本正明†††

非手続き型言語からメモリ効率や実行効率が良いプログラムを生成するためには，Jackson の構造的プログラム設計法の主テーマである入出力データの構造不一致を検出し解決する必要がある．そこで著者らはすでに，非手続き型言語によく現れる集合と写像の性質に着目した，構造不一致自動検出解決法を提案した．本論文では，その検出解決法を実用化していく上の課題とその解決案を明確化するために行った実験について報告する．著者らは，集合と写像の性質を持った ER (Entity-Relationship) モデルに基づく非手続き型言語 PSDL (Program Specification Description Language) のコンパイラへ適用して実験を行った．このコンパイラは，PSDL 仕様を有向グラフへ変換し，集合と写像の性質を用いて解析の上，部分グラフに分割する．さらに，その部分グラフに基づいてプログラム構造を決め，スケルトンを用いてCプログラムを生成する．このコンパイラによる実験の結果，1）グラフ解析方法の改善による構造不一致検出解析精度の向上，2）スケルトンの洗練による被生成プログラムの効率向上，3）ヒューリスティックな解析法を用いた被生成プログラムの大局的最適化によるコンパイラの性能向上，の課題と解決案とが明らかになった．被生成Cプログラムの効率の悪さは手書きのものに対して1.3倍から2.5倍の程度であり，上記の改善案を実現すれば，実用的なプログラムが生成可能と予想される．

1. はじめに

今日，信頼が高いプログラムを効率良く作成する方法が大きな問題となっており，その解決策としてソフトウェア再利用が注目されている．そこで著者らは，従来効果を上げてきたプログラム再利用をさらに改善するため，プログラム仕様の再利用について研究している．具体的には，再利用対象仕様の理解や修正を容易にするため，ER (Entity-Relationship) モデル[1]を適用した非手続き型のプログラム仕様記述言語 PSDL (Program Specification Description Language)[2)~4)]を研究している．

ところで，メモリ効率や実行効率が良いプログラムを作成するためには，プログラムの入出力データの構造不一致[5]を精度良く検出し解決しなければならない．文献5) では，構造不一致はプログラムで同時に処理される入出力データ間で入出力タイミングが同期していない問題として定義されている．そこで，C言語のような手続き型言語でプログラミングする場合，作成するプログラムに構造不一致が存在すれば，プログラムはその構造不一致を検出し，構造不一致を解決するためにデータをプログラム内のテーブルなどに保持して待ち合わせることによって同期がとれるようにプログラムを作成しなければならない．

この構造不一致の検出と解決はプログラムの仕様に関わる問題ではなく，プログラムの実現に関わる問題である．このため，プログラマが構造不一致を意識しなくてもよく，コンパイラが構造不一致を検出し解決してくれる非手続き型言語が存在する．たとえば，配列の性質に着目して構造不一致を検出し解決する MODEL[6] がそれである．

そこで著者らは，仕様の理解や修正を容易にするため，プログラムが構造不一致を意識しなくてもよく，コンパイラが構造不一致を検出し解決する非手続き型言語として，ER モデルを適用した PSDL を研究している．ER モデルでは，実体型や関連型は各々実体や関連の集合である．さらに関連型は，一方の実体型に属するどの実体が他方の実体型に属するどの実体に対応付けられるかを表す写像でもある．このように，PSDL は上記の配列とは全く異なった集合と写像という性質を備えている．集合と写像は高位の非手続き型言語に採用される傾向にあるので，著者らはそれらの性質に着目した構造不一致検出解決法を提案した[7]．

本論文では，文献7) で提案した構造不一致検出解決法を実用化していく上の課題とその解決策を明確化するために PSDL コンパイラを作成して実験を行ったので，その実験について報告する．なお，今回実験した構造不一致検出解決法は，構造不一致のうち，脈絡不一致[5]のみを持ったバッチ処理プログラムを対象

† An Experiment in a Detection and Solution Method for Structure Clash between Program Input-Output Data by KATSUMI OKAMOTO (Sumitomo Metal Industries, Ltd.) and MASAAKI HASHIMOTO (ATR Communication Systems Research Laboratories).

†† 住友金属工業(株)
††† (株) ATR 通信システム研究所

* 本研究は，ATR 通信システム研究所で実施された．

としている．第2章では PSDL を概説する．第3章で構造不一致を説明し，構造不一致を検出して解決する PSDL コンパイラ方式を述べる．第4，5，6章で実験とその考察，今後の課題とその解決案について述べる．

2. PSDL

PSDL では，プログラムの入出力データの性質に着目し，プログラム仕様を以下の3階層に分け記述する．

（1） プログラムの入出力データに表される対象世界の情報について，その枠組みを定めた情報層．

（2） 入出力のデータ構造を定めたデータ層．

（3） 入出力ファイルのアクセス方法を定めたアクセス層．

PSDL を，図1を用いて概説する．図1は，ファイル product_d と sale_d を入力して，売上ごとに計算した売上額を，顧客ごとに計算した売上合計も併記してファイル account_d へ出力するプログラム仕様である．

2.1 情 報 層

情報層は，図1では 00 行目の INFORMATION 文と 23 行目の DATA 文の間に記述されている．

（1） 実体型，属性，主キー，関連型

入出力データは，Television，VCR というような実体を表している．図1では，各々の実体型を 01 行目の E (Entity) 文で記述する．その主キー属性は 02 行目の K (Key) 文で記述し，主キーでない属性を 03 行目の A (Attribute) 文で記述する．ここで，02 行目の STR (STRing) は属性の定義域が文字列であることを示し，03 行目の NUM (NUMber) は数値であることを示す．

入出力データは，“sale No. 1 で Television が売られた” というような関連も表している．図1では，各々の関連型を 04 行目の R (Relationship) 文で記述する．それに続けて，関連型で対応付けられた実体型を 05，07 行目の C (Collection) 文で指定する．この文には実体型とその役割を “役割．実体型名” の形で指定する．それらの実体型が異なっていれば 05 行目のように役割を省略してよい．C文の後には，その実体につながる関連の個数を 06 行目の RN (Relationship Number) 文で記述する．06 行目の “M” (Many) は個数が2以上であることを示し，08 行目の “1” は1個であることを示す．

（2） 制約

属性値従属性制約 (AD: Attribute value Dependency constraint) は，属性の値を得るための制約である．図1では，値が得られる属性のA文に続けて，13 行目の＝(equal) 文で記述する．この制約から参照される属性は “属性名” または “役割．関連型名．

```
00: INFORMATION
01:   E product
02:     K name STR
03:     A price NUM
04:   R sold
05:     C .product
06:     RN M
07:     C .sale
08:     RN 1
09:   E sale
10:     K number NUM
11:     A quantity NUM
12:     A amount NUM
13:       = FMULT(.sold..product.price,quantity)
14:   R buy
15:     C .customer
16:     RN M
17:     C .sale
18:     RN 1
19:   E customer
20:     K name STR
21:     A total NUM
22:       = FSUM(.buy..sale.amount)
23: DATA
24:   I product_data
25:     O product_record
26:       %12s product_name
27:         = product.name
28:       %8d product_price
29:         = product.price
30:   I sale_data
31:     O sale_record
32:       %4d sale_number
33:         = sold..sale.number
34:         = buy..sale.number
35:       %16s sale_customer
36:         = buy..customer.name
37:       %12s sale_product
38:         = sold..product.name
39:       %4d sale_quantity
40:         = sale.quantity
41:   I account_data
42:     O account_record
43:       %4d sale_number
44:         = sale.number
45:         = buy..sale.number
46:       %8d sale_amount
47:         = sale.amount
48:       %16s sale_customer
49:         = buy..customer.name
50:       %10d customer_total
51:         = buy..customer.total
52: ACCESS
53:   D product_d INPUT   product_data
54:   D sale_d    INPUT   sale_data
55:   D account_d OUTPUT account_data
```

図 1 PSDL プログラム仕様
Fig. 1 PSDL program specification.

役割2. 実体型名. 属性名" の形で記述する. ここで前者は, 値が得られる実体が持っているほかの属性を参照する際に用い, 一方その実体へ関連で対応付けられたほかの実体の属性を参照する際に後者を用いる.

関連存在従属性制約 (RD: Relationship existence Dependency constraint) は, 既存の実体を対応付ける関連を得るための制約である. この例は図1にはないが, R文とC文に続けて RC (Relationship existence Condition) 文を用いて, 関連が存在する条件を記述する.

実体存在従属性制約 (ED: Entity existence Dependency constraint) は, 既存の実体から新たな実体を得るための制約である. この例も図1にはないが, 得られる実体が属する実体型を定義したE文に続けて EC (Entity existence Condition) 文を用いて, 実体が存在する条件を記述する.

2.2 データ層とアクセス層

データ層は, 図1では 23 行目の DATA 文と 52 行目の ACCESS 文の間に記述されている. 入出力データの構造は基本データ型や, 連接集団データ型, 繰り返し集団データ型, 選択集団データ型で階層的に定める. これらのデータ型は, 各々 % 文や, O (sequence Order) 文, I (Iteration) 文, S (Selection) 文で記述する. %文は, 基本データ型のデータ形式をC言語と同様に定める. たとえば, 26 行目の %12s は 12 文字の文字列であることを定めている. ところで, 本論文では入出力データは実体やその属性, 関連を表すものと見なしているので, 実体については実体型の主キーを 27 行目のように=文で基本データ型と結合する. 属性については, 属性を 29 行目のように=文で基本データ型と結合する. 関連については, 関連で対応付けられた実体型の主キーを 33 行目と 38 行目の=文で基本データ型と結合する. これらの結合を情報制約 (IC: Information Constraint) と呼ぶ.

アクセス層では, ファイルをデータセットと呼び, 図1の52行目の ACCESS 文に続けて, 53行目の D (Dataset) 文で記述する. D文には, ファイル名, 入出力の区別を記述する. また, データ層とアクセス層を結合するため, product_data のようにデータ型の指定もD文で行う.

3. PSDL コンパイラ方式

入出力データの構造不一致について説明し, その構造不一致を, PSDL コンパイラが検出して解決するための方式を, 1) PSDL 読み込み部, 2) 有向グラフ作成部, 3) 局所的構造不一致検出部, 4) 大局的構造不一致検出部, 5) 構造不一致解決部, 6) コード生成部の各フェーズに分けて述べる.

3.1 構造不一致

第2章で述べたプログラムには, 入力レコードがソートされていないという前提のもとで構造不一致が存在している. すなわち, sale amount の計算では sale と product の入力タイミングが同期していない. さらに, customer total の計算や sale amount と customer total を併記した出力についても入出力タイミングが同期していない. これらの構造不一致を解決するためには, product name と product price, sale number, sale amount, customer name, customer total, および, sale と customer の対応付けをテーブルなどに保持して待ち合わせることによって, 同期がとれるようにプログラムを作成しなければならない.

なお本論文では, 以下の制限のもとで構造不一致を検出して解決する PSDL コンパイラの方式を述べる.

(1) 入出力ファイルは順アクセスされ, 入力レコードはソートされていない.

(2) 入出力レコードに相当したデータ型は1つのファイルに1つしかない.

(3) 1つの入出力レコードは同一型の実体は高々1つしか表さず, 同一の型の関連も高々1つしか表さない.

3.2 PSDL 読み込み部と有向グラフ作成部

PSDL プログラム仕様を読み込み, その構文を解析して有向グラフを作成する. グラフには, 構造節点に分類される実体型節点や属性節点, 関連型節点, データセット節点とがあり, また制約節点に分類される AD 節点や RD 節点, ED 節点, IC 節点とがある. 実体型節点は, 実体型を表すだけでなく, その主キー属性も表している. また, データセット節点は, そのデータ型もまとめて表している.

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入出力の方向に合わせる. **図2**は, 図1の仕様から作成されたグラフである. なお, 本論文では単一方向閉路はないものとする.

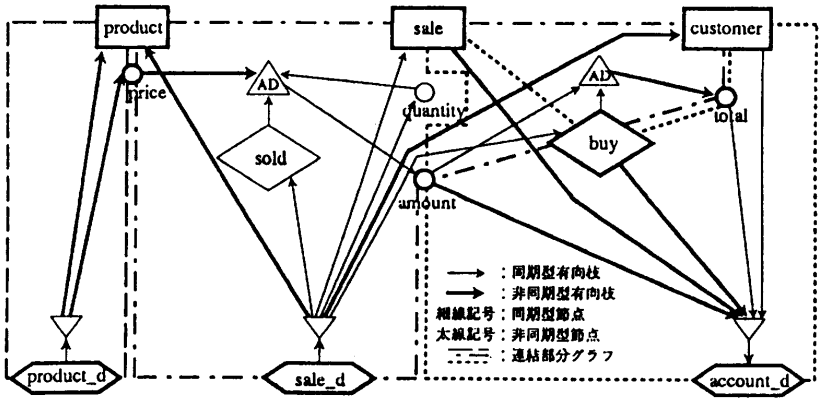


図 2 有向グラフ
Fig. 2 Directed graph.

3.3 局所的構造不一致検出部

プログラムは多数のデータを順次入力して処理するので，前述のグラフの各有向枝を実体または属性値，関連，データといった対象が多数順次流れており，それらの対象を組み合わせる計算などの処理が節点で起きるものと考える．そこで，有向枝を同期型と非同期型とに分け，実体型節点か，属性節点，関連型節点を介してつながった同期型有向枝の間では対象が一対一にタイミングを同期させて流れており，一方，非同期型有向枝はそれらの節点を介してつながった他の有向枝との間で対象の流れのタイミングが同期していないものとする．なお，第2章で述べた制約は，制約節点で対象の流れのタイミングが同期するように定式化した．なお，対象の流れのタイミングが同期していない非同期型有向枝の箇所で，構造不一致が起きていることになる．

そこで，まず局所的な解析によって非同期型有向枝を検出する．

（1） 実体や関連の和集合

実体型節点か関連型節点に2本以上の枝が流入していれば，それらの枝を非同期型とする．また，それらの実体型の属性節点へ流入している枝も非同期型とする．以下，実体型節点について理由を説明する．

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た実体が2個以上流れてきても，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．和集合をとるためには，実体の集合の間で主キー値を照合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前提なので，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流れてくるとは限らない．したがって，すべての実体を待ち合わせて照合しなければならないので，各流入枝は，同じ実体型節点の他の流出入枝との間で実体の流れの順序が一致しない．

（2） 写像の数量関係[8]

実体型節点または属性節点と，AD 節点との間にある枝は，実体相互の写像，すなわち対応付けの数量関係を表す 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，その枝を非同期型とする．その理由を以下に説明する．

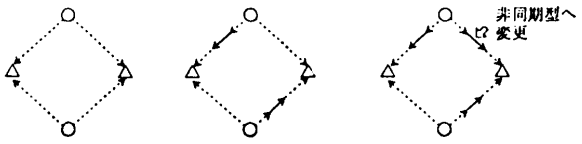
AD へ属性節点から流出する枝の関連数が2以上かMの場合，同じ属性値が同じ AD から何回も参照される．ところが，入力データはソートされていないので，同じ属性値が連続して参照されるとは限らない．したがって，属性値がいつ参照されても良いように待ち合わせなければならないので，上記の流出枝は同じ属性節点の他の流出入枝の間で属性値の流れの順序が一致しない．また逆に，AD 節点から属性節点へ流入する枝の関連数が2以上かMの場合，その AD には図1の 22 行目の合計をとる FSUM のような集合関数が指定され，中間データを必要とする．この中間データは何回も更新されるが，いつ更新されるかわからず，属性節点への流入枝は同じ属性節点の流出枝との間で属性値の流れの順序が一致しない．

3.4 大局的構造不一致検出部

グラフは，前述の局所的解析の結果に基づいて大局的に解析して，さらに構造不一致を検出する．

（1） 閉路による矛盾の検出と解決

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



(1) 方向混在閉路 (1) 2-way circuit. (2) 矛盾の検出 (2) Inconsistency detection. (3) 矛盾の解決 (3) Inconsistency solution.

○：構造節点 ←：同一方向を持った同期型有向枝のつながり
△：制約節点 ←：非同期型有向枝

図 3 閉路による矛盾の検出と解決
Fig. 3 Inconsistency detection and solution with 2-way circuit.

理由を以下に説明する．図3(2)に示す非同期型有向枝で分けられた閉路部分のいずれか1つに着目すると，その中では同期型有向枝しかないので対象の流れの順序が一致しており，すべての制約を対象ごとに同期して実行できる．一方，非同期型有向枝については，前述のようにすべての対象を待ち合わせたので，すべての対象が流入した後でないと制約の実行は先に進めない．ところが，図3(2)に示す非同期型有向枝はすべて同じ方向を持っているので，閉路部分相互の実行順序が三すくみのようになっており，どの閉路部分から先に実行してよいか決められない．この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本を非同期型に変えればよい．

(2) プログラムの大局的な最適化

前述の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．その解は，局所的解析で得られた非同期型枝をすべて包含し，しかも上記の(1)の解析で矛盾を生じない非同期型枝の集合のうち，プログラムの効率が最も良いものとして得られる．しかし，この解決法自体が大きな課題であるので，後述の実験では以下の方法を用いた．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．

3.5 構造不一致解決部とコード生成部

非同期型有向枝として検出された構造不一致を解決するプログラムのデータ構造と手続き構造を自動的に設計して，その構造に沿ってスケルトンを割り当て展開してコード生成を行う．スケルトンは，C言語のマクロとして記述されたもので，数行から 10 数行のCコードからなる部品の集まりである．

データ構造は，以下の順序で設計する．

(1) 構造節点の同期性

グラフ中で，非同期型枝を持っている実体型節点と，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．

(2) 配列変数とスカラ変数の割り当て

非同期型の実体型節点と，属性節点，関連型節点には配列変数を割り当て，同期型の構造節点にはスカラ変数を割り当てる．たとえば，図2のグラフの解析結果を用いてデータ構造を設計して，コード生成すると，図4のデータ定義部のようになる．

手続き構造は，以下の順序で設計する．

(1) 有向グラフの分割

グラフを以下に述べる方法で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．このようにグラフを分割すれば，部分グラフの中では対象の流れのタイミングがすべて同期する．

(2) 手続きブロックの割り当てと実行順序

各々の部分グラフへ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2つの部分グラフの境界上のすべての非同期型節点は，その流入枝を一方の部分グラフの中に持ち，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，ブロック間に半順序ができるので，この半順序からブロック間の全順序を得て，実行順序とする．

(3) 手続きブロック内の手続き割り当てと実行順序

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できている．この半順序から手続き実行のための全順序を得る．

(4) 手続きブロック内の繰り返しループの割り当て

各々の手続きブロックへ1つ以上の手続き繰り返しループを割り当てる．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，図2の左側の破線で囲まれた手続きブロックでは，図4の手続き部の while 文に示すようにファイ

```
#include <stdio.h>

                :

char            productname[50][65];
float           productprice[50];

float           saleamount[100];
float           salenumber[100];
float           salequantity;

struct buyStruct {
int             customer;
int             sale;
}               buy[100];

char            customername[50][65];
float           customertotal[50];

main()
{
if ((account_data = fopen("account", "w")) == NULL) {
fprintf(stderr, "File can't open <%s>\n", "account");
exit(1);
}

                :

while (1) {
if (fgetc(product_data) == EOF)
break;
else
fseek(product_data, -1L, SEEK_CUR);
product_id++;
fscanf(product_data, "%12c", productname[productCount]);
strcmprs("%12c", productname[productCount]);
productIndex = EntityUnionproduct(productname[productCount]);
fscanf(product_data, "%08d", &wdata);
productprice[productIndex] = wdata;
fseek(product_data, 1L, SEEK_CUR);
}

                :

for (saleIndex = 0; saleIndex < saleCount; saleIndex++) {
buyIndex = GetEntRelshipIndex(2, 2, buyCount, saleIndex, buy);
buycustomerIndex = buy[buyIndex].customer;
buysaleIndex = buy[buyIndex].sale;
wdata = salenumber[saleIndex];
fprintf(account_data, "%04d", wdata);
wdata = saleamount[saleIndex];
fprintf(account_data, "%08d", wdata);
fprintf(account_data, "%-16s", customername[buycustomerIndex]);
wdata = customertotal[buycustomerIndex];
fprintf(account_data, "%010d", wdata);
fprintf(account_data, "%s", "\n");
}
fclose(account_data);

                :

}
```

図 4 被生成Cプログラム
Fig. 4 Generated C program.

ル product_data が空になるまでループを繰り返すように割り当てられている．

ここで，すべての制約が各々1つの関数としてコードが生成される．図4の手続き部は図2から得られたプログラムの手続きの一部である．

4. 実　　験

前章で述べた PSDL コンパイラ方式を実用化していく上の課題とその解決策を明確化するために実験を行った．本章では，その実験について述べる．

4.1 実験用 PSDL コンパイラの作成

言語処理系の作成でよく用いられる自己記述は，1）使いやすい新言語でコンパイラの作成や拡張ができる，2）新言語のデバッグが兼ねられる，3）新言語の評価データが得られる，などの実験に向いた利点を備えているので，著者らも自己記述によって PSDL コンパイラを作成した．このコンパイラは，第3章で述べた6つのフェーズに分けられている．なお，自己記述のコンパイラ自体をCプログラムへ変換するため，PSDL 核コンパイラも作成した．このコンパイラは性能を無視しても簡便に作成するのが望ましいので，すべての実体型や属性，関連型をテーブルで生成することによって，構造不一致の検出解決は避けた．このため，PSDL 読み込み部とコード生成部のみを持たせ，C言語で作成した．この PSDL 読み込み部は PSDL コンパイラでも共用し，PSDL コンパイラのその他のフェーズを PSDL で自己記述した．

これらのコンパイラは SUN 3/60 C 上に作成した．PSDL コンパイラは 30 本以上のプログラムからできているが，それらはファイルを介して接続した．なお，生成時に属性へ割り当てた変数のデータ形式は，一律に数値型の場合は単精度浮動小数点数とし，文字列型の場合は 64 文字とした．

表1 評価データ
Table 1 Evaluation data.

テスト・プログラム仕様	1	2	3	4	5	6	7
PSDL 記述 (行数)	56	36	48	72	38	41	35
被生成Cプログラム (行数)	180	125	171	220	154	156	146
手書きCプログラム (行数)	101	48	137	145	95	97	127
データ定義部の倍率 (被生成/手書き)†	2.96	2.14	1.36	2.23	2.10	1.97	1.73
手続き部の倍率 (被生成/手書き)†	1.84	2.41	1.70	1.47	1.77	2.14	1.22
全行の倍率 (被生成/手書き)†	1.96	2.36	1.64	1.55	1.81	2.11	1.28
全コンパイル時間 (sec)	64.7	23.1	73.5	78.3	24.0	25.0	24.0
大局的解析時間 (sec)	35.2	3.6	42.7	35.3	2.0	1.5	5.8
割合 (大局的解析/全コンパイル)	0.54	0.16	0.58	0.45	0.08	0.06	0.24

† CプログラムからCコンパイラ Gcc Ver. 1.36 によって生成したアセンブラ・プログラムのステップ数の倍率.

4.2 評価データ

実験で得られた評価データを以下に示す.

(1) テスト・プログラムのデータ

大局的構造不一致検出のための両方向閉路解析には，膨大な閉路数による解析能力の限界があるので，今回の実験では表1に示すように PSDL 記述が 100 行以下の7本の PSDL プログラム仕様を用いた.

また，表1にはそれらの PSDL 仕様から PSDL コンパイラで生成されたCプログラムの行数を，手書きCプログラムと対比して示している．ただし公平を期すため，両者ともCコンパイラで生成されたアセンブラの静的なステップ数で示す．さらに，PSDL コンパイラの実行時間を，コンパイル時間中で大局的解析のフェーズが占める時間とその割合とを合わせて示す.

(2) コンパイラのデータ

PSDL コンパイラの読み込み部の規模は 2,200 行であり，その他の部分の規模は 14,400 行であった．そのうちC言語で作成されたスケルトンは 500 行であった．また，実験段階の PSDL では関数や述語の定義をC言語で記述しており，その部分の 4,000 行が含まれている．フェーズ別に見ると特にコード生成部が大きく，約5割を占めた．その他のフェーズは同程度の大きさであった.

5. 考　　察

前章の実験結果をもとに考察を述べる.

(1) 構造不一致の検出と解決の精度

本論文の検出解決法は文献4)と比較すると，以下の工夫によって検出と解決の精度が向上した.

文献4)ではグラフの節点だけが同期性を持ったが，本稿では節点の他に有向枝にも同期性を持たせた．そこで同一の非同期型節点に接続する2本以上の同期型有向枝に着目すると，それらの枝は文献4)では別々の部分グラフに含まれた．一方，本論文では同じ部分グラフに含まれるので，性能の良いプログラムを生成でき，構造不一致の解決精度が向上した.

また，文献4)の大局的構造不一致検出では，同一節点から流出して他の同一節点に流入する有向枝群を解析した．しかし，その解析法では検出できない非同期型節点が存在するので，本論文の両方向閉路解析法を考案し，検出精度を向上した.

なお，今後の課題で後述するように検出精度には向上の余地がまだ残されている.

(2) 生成されたプログラムの実行効率

プログラムの実行効率はアセンブラの動的命令数で比較するのが望ましいが，アセンブラの静的命令数でもおおよその比較ができる．そこで，PSDL コンパイラから生成されたCプログラムと手書きCプログラムを，アセンブラの静的ステップ数で比較した表1の手続き部を見ると，前者の効率の悪さは後者の1.3倍から2.5倍の程度であった.

しかし，今回の実験は構造不一致検出解決精度の向上が主目的であったので，スケルトンは500行と小さく，洗練されていなかった．したがって，スケルトンのマクロの種類を増やすことにより，まだ実行効率の向上が可能である.

(3) PSDL コンパイラの性能

表1から，PSDL コンパイラのコンパイル時間には，大局的解析が大きく影響しており，大局的解析に時間がかかる仕様ほどコンパイル全体の時間も大きくなることがわかる．これは，3.4 節で述べた閉路による矛盾の検出を行う際の閉路探索に多大な時間を要するからである．このためにテストプログラム仕様2と3のように，表1において PSDL 記述量は，36 行，

48 行と大きな差がないにもかかわらず，全コンパイル時間は，23.1 秒，73.5 秒と約3倍，さらに大局的解析のフェーズに着目すると，3.6 秒，42.7 秒と約 12 倍というように大きな差がついた．現 PSDL コンパイラでは，閉路探索が性能ネックになっている．

（4） 実時間処理プログラムの生成と仕様の再利用

実時間処理プログラムについては実時間データ入出力のタイミング不一致がプログラム構造へ大きく影響するので，そのプログラム生成に今回の PSDL コンパイラは適用できない．このため，実時間データ入出力のタイミング不一致の検出と解決についても研究中である[9]．タイミング不一致は構造不一致とプログラム中に混在して現れるので，両検出解決法の統合が必要である．

また，自己記述の PSDL コンパイラを実験のために何回も修正した経験から，PSDL が仕様の再利用に適していることが確認できた．そこで著者らは，PSDL で記述された仕様の再利用技術を研究中である[10]．

6. 今後の課題

PSDL コンパイラ方式を実用化していく上での課題とその解決策を述べる．

（1） 本論文の構造不一致検出解決法の制限条件

本論文の構造不一致検出解決法は，有向グラフ上に単一方向閉路が現れず，しかも構造不一致のうち，脈絡不一致[5]のみを持ち，順序不一致[5]と境界不一致[5]とを持たないバッチ処理プログラムの仕様から，正しく動作するプログラムを生成することができる．一方，これらの制限条件に触れる場合は，正しく動作するプログラムは生成できない．これらの制限条件の解除は今後の課題である．

（2） 構造不一致の検出と解決の精度

上記の制限条件の範囲で，本論文の構造不一致検出解決法は，正しく動作するプログラムを生成できるが，構造不一致の検出と解決の精度を向上させることによって，さらに実行効率とメモリ効率の良いプログラムを生成できる余地が残っている．

たとえば，著者らは従来，PSDL プログラム仕様から得た有向グラフ上の構造へどのように同期性を持たせるか，あるいはその有向グラフをどのように大局的に解析するかを研究することによって，プログラムの入出力データの構造不一致の検出精度の向上，つまりは被生成プログラムの性能向上を研究してきた．文献4)ではグラフの節点に同期性を持たせたり，本論文では節点のほかに有向枝にも同期性を持たせ，さらには両方向閉路解析法を有向グラフの大局的解析に採用することによって構造不一致の検出精度の向上が得られることが判明した．

さらに今回の実験の結果，本論文では，グラフの有向枝へ同期型か非同期型かの性質を持たせて構造不一致の中の脈絡不一致[5]を検出したが，この検出法では，脈絡が一致しているにもかかわらず，不一致として検出されるものがある．この問題を解決するためには，同期型か非同期型かの性質を有向枝へ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る解決案があることが判明した．この解決策を施せば，たとえば，図1 の仕様において sale. amount を得るための AD が product. name も参照するとした場合，そこに両方向閉路ができるため，実体型 product に対する和集合演算で生じた非同期性が閉路解析によって波及し，本来は同期型のままでよい関連型 sold も非同期型となる．しかし，これが同期型のままでプログラム構造を設計できるようになり，さらに性能のよいプログラムが生成可能となる．

また，本論文では非同期型の実体型が検出された場合，その型に属する実体はすべてテーブル上で待ち合わせるようにプログラム構造を決めた．しかし，入出力データ中に部分的にしか構造不一致が存在しない場合には，すべての実体をテーブル上で待ち合わせる必要はなく，構造不一致の存在する部分だけを待ち合わせればよい．このように部分的な待ち合わせを行った方が被生成プログラムの実行効率やメモリー効率がよくなる．この部分構造不一致は，従来，1階層の有向グラフ記述しかしていなかったために解決できていないことが今回の実験で判明した．したがって，多階層の有向グラフ記述を用いた部分構造不一致の検出解決法も検討していく必要がある．

さらに，本論文では脈絡不一致[5]について述べたが，順序不一致[5]や境界不一致[5]の研究も必要である．

（3） 生成されたプログラムの実行効率

生成されたプログラムの実行効率向上を図るためには，前述の構造不一致検出解決能力を向上させることも重要であるが，PSDL コンパイラのコード生成部と，スケルトン中の処理ルーチンや関数を改善することも必要である．現 PSDL コンパイラでは，処理ルーチンや関数は種々の状況に適用でるように冗長な

機能を持たせて実現している．この処理ルーチンや関数を個々の状況にのみ適用するように専用化してスケルトンを洗練し，そのスケルトンをコード生成部で使い分けてプログラムを生成すれば，さらに実行効率が向上する．

さらに，情報層で記述された属性については，4.1節で述べたように一律な形式の変数を生成したが，データ層の基本データ型と従属性制約中の演算などから変数の形式を的確に決めることも必要である．

（4） PSDL コンパイラの性能と被生成プログラムの大局的最適化

PSDL コンパイラの性能ネックは大局的解析における閉路探索によって生じている．このネックを解消するためには，閉路をすべて認識せずに大局的解析を行わなければならない．そこで，閉路解析をカットセット解析に替え，その中にヒューリスティックな解析法を導入する研究を行っている．

7. おわりに

構造不一致の検出と解決を特徴とする PSDL コンパイラの実験について述べた．実験の結果，本論文の両方向閉路解析法を考案したことや有向枝に同期性を持たせたことによって従来の PSDL コンパイラよりも構造不一致の検出精度が向上した．また，PSDL コンパイラによって生成されたプログラムの実行効率の悪さは，手書きプログラムに比べて 1.3 倍から 2.5 倍の程度であった．一方，PSDL コンパイラの性能については，構造不一致の大局的解析によって性能ネックが生じていることが判明した．

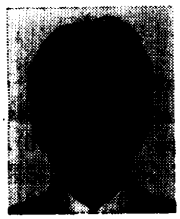
また，実験の結果，1）グラフ解析方法の改善による構造不一致検出精度の向上，2）スケルトンの洗練による被生成プログラムの効率向上，3）ヒューリスティックな解析法を用いた被生成プログラムの大局的最適化によるコンパイラの性能向上，の課題とその解決策が明らかになったので，今後は，PSDL コンパイラ方式が実用に耐えうるよう，それらの課題と解決策を具体化していく予定である．

謝辞 日頃ご指導いただく ATR 通信システム研究所の葉原会長，寺島社長，竹中前室長，山下前社長，ご意見をいただいた通信ソフトウェア研究室の諸氏に深く感謝します．また，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) Chen, P. P.: The Entity-Relationship Model—Toward a Unified View of Data, *ACM Trans. Database Syst.*, Vol. 1, No. 1, pp. 9-36 (1976).
2) 橋本正明：EAR モデルに基づく情報構造記述を用いたプログラム仕様記述法 PSDM，情報処理学会論文誌，Vol. 27, No. 7, pp. 697-706 (1986).
3) Okamoto, K. and Hashimoto, M.: On Real-Time Software Specification Description with a Conceptual Data Model-Based Language, *Proc. 2nd ICCI 1990*, pp. 186-190 (1990).
4) 橋本正明：非手続き型言語と入出力データの構造不一致，情報処理学会論文誌，Vol. 29, No. 12, pp. 1141-1150 (1988).
5) Jackson, M. A.（鳥居宏次(訳)）：構造的プログラム設計法の原理，p. 318，日本コンピュータ協会，東京 (1980).
6) Prywes, N. S. and Pnueli, A.: Compilation of Nonprocedural Specifications into Computer Programs, *IEEE Trans. Softw. Eng.*, Vol. SE-9, No. 3, pp. 267-279 (1983).
7) Hashimoto, M. and Okamoto, K.: A Set and Mapping-based Detection and Solution Method for Structure Clash between Program Input and Output Data, *Proc. 14th COMPSAC 1990*, pp. 629-638 (1990).
8) Tsichritzis, D. C. and Lochovsky, F. H.: *Data Models*, p. 381, Prentice-Hall, New Jersey (1982).
9) Okamoto, K. and Hashimoto, M.: Program Generation from Real-Time Software Specification Described with a Conceptual Data Model-based Language, *Proc. 3rd SEKE 1991*, pp. 261-270 (1991).
10) Okamoto, K. and Hashimoto, M.: Visual Programming with Reusable Specifications Described by a Conceptual Data Model- and Constraint-Based Language, *Proc. 4th HCI 1991*, pp. 587-591 (1991).

（平成3年9月9日受付）
（平成4年2月14日採録）



岡本 克己（正会員）

1963 年生．1987 年大阪大学基礎工学部情報工学科卒業．同年住友金属工業(株)入社．1988～1991 年 ATR 通信システム研究所出向．現在，住友金属工業(株)システム研究開発部に勤務．これまで主にソフトウェア自動作成の研究，ソフトウェア開発保守支援システムの研究開発などに従事．

橋本 正明（正会員）

昭和 21 年生．昭和 43 年九州大学工学部電子工学科卒業．昭和 45 年同大学院修士課程修了．同年日本電信電話公社電気通信研究所勤務．昭和 63 年 ATR 通信システム研究所出向．これまで主にオペレーティング・システム，中間言語マシン，データベース設計支援ツール，ソフトウェア自動作成の研究実用化に従事．著書「データ中心のプログラム仕様記述法」（井上書院）．工学博士．電子情報通信学会，人工知能学会，IEEE 各会員．