プロセス分解代数に基づくデータフロー図の段階的詳細化

鈴木英明† 高橋直久†

構造化分析手法では，データフロー図 (DFD) を段階的に詳細化して要求分析する．Adler が提案したプロセス分解代数は，DFD の詳細化作業を形式化し，入出力マトリクスから DFD を生成する枠組を与えている．本稿では，データストアと内部フローを考慮したプロセス分解代数を提案し，この代数を用いて「抽象化バランスが均等な DFD」を与える手法について議論する．提案手法は，従来手法では実現困難であった次の特徴をもつ．(1)人手で作成した DFD とほぼ等しい形状に，内部データを含む任意の構造の DFD を生成できる．(2)DFD の詳細化過程で入出力マトリクスの変更を要しない．(3)インクリメンタルに DFD を生成し DFD の詳細化結果を再利用できる．本稿では，さらに，ワークステーション上に実現した提案手法に基づく DFD 詳細化システムを用いた適用実験により，提案手法の有効性を示す．

A Process Decomposition Algebra for Stepwise Refinement of Data Flow Diagrams

HIDEAKI SUZUKI† and NAOHISA TAKAHASHI†

Stepwise refinement of data flow diagrams (DFDs) is one of the most essential tasks in structured analysis of software requirements. A higher-level DFD process, specified by an Input/Output connectivity matrix (I/O matrix), is decomposed into lower-level DFDs in a formal manner by applying process decomposition algebra proposed by Adler. This paper presents other algebra where internal data stores and flows are taken into consideration for the construction of DFDs. It also discusses how to obtain the DFD where all of the processes and data flows are evenly decomposed in terms of abstraction level. Furthermore, the process decomposition experiments using a DFD refinement system developed on a UNIX workstation demonstrate that the decomposition procedure for applying the proposed algebra has the following advantages: (1) An I/O matrix is transformed into the DFD including internal data stores and flows, which has almost the same structure as that of manually decomposed DFDs. (2) The decomposition procedure requires no modification of the I/O matrix which should be transformed into a complex DFD. (3) Producing a DFD in an incremental manner reduces the computation time for the decomposition of a large DFD by re-using small DFDs, which have been decomposed as components.

1. はじめに

ソフトウェア作成における要求分析は，抽象的な仕様記述を段階的に詳細化する作業としてとらえることができる．De Marco の提案した構造化分析 (SA) 手法では，この作業をデータフローダイアグラム(DFD)の詳細化作業としてモデル化し，作業手順を与えている[1]．また，Ward と Mellor は，SA 手法を拡張して，リアルタイムソフトウェアに適した要求分析手法を提案している[2]．

De Marco の手法では，DFD は，処理機能（プロセス）とデータストアをノードとし，処理の依存関係（データフロー）をアークとする有向グラフで表現される[1]．DFD の詳細化作業の手順は次のように与えられている．まず，設計対象ソフトウェアを，1つの処理機能とこの機能に対する入出力データを記述した最も上位の DFD（コンテクストダイアグラムと呼ぶ）として表現する．次に，上位の DFD の各処理機能を複数の要素機能（サブシステム）に分割し，サブシステム間のデータの依存関係を記述して下位の DFD を作成する．この一連の作業を繰り返すことにより，階層的な DFD を得る．上位の DFD から下位の DFD を作成することを，DFD の詳細化または DFD のプロセス分解と呼ぶ．

De Marco や Yourdon は，詳細化のステップが1段階進んだときに詳細化作業を容易に行えるとともに，要求分析過程をトレースするときに全体像がつかみやすくなるように詳細化すべきだとしている[1,3]．De Marco は，経験に基づいて，このように詳細化を行うため，次のような点に留意すべきであると述べて

† NTT ソフトウェア研究所
NTT Software Laboratories

いる[1].

（1） データの依存関係に従って自然に分割すること.

（2） 設計者が理解しやすいダイアグラムであること.

（3） 概念的に意味のあるインタフェースができるように詳細化すること.

このように，これまでの多くの提案では経験則に基づく概念的な議論が与えられているだけであるので，具体的な詳細化作業では設計者の経験に任される部分が多い．Adler は，文献4)で，これらの De Marco の経験則をまとめて，データの依存関係に従い，さらに，各入出力データに対して，それを扱うプロセスを一意に決めた DFD のうち，必要最小数のプロセスとデータフローによって構成される「最も抽象的な DFD」を作成することが重要であると指摘している．さらに，DFD を詳細化する作業を形式化し機械的に行えるようにするため，プロセス分解代数と呼ぶ文（センテンス）の変換系を提案している．

Adler は，プロセスの入出力データについて「AによりBを生成する」という関係に着目し，プロセスとそれに対する入出力を表現する文を定義した．さらに，DFD に対する詳細化手順を，文に対する変換オペレータとして形式化した．彼の手法では，コンテクストダイアグラムの入出力データの依存関係を文で表し，その文に変換オペレータを繰り返し適用することにより下位の DFD を作成する．しかし，この手法は，（1）内部データ（データストアおよび内部データフロー）を取り扱うことができない，（2）扱う DFD の形状に制限があるため詳細化過程で入出力データの依存関係を変更しなければならない場合がある，（3）既存の詳細化の結果を再利用できない，という問題がある．

問題（1）に関して，データストアに対する解決法は，松本と本位田により議論されている[5]．しかし，内部データフローを取り扱うことができないという問題は残されている．また，問題（2），（3）に対する解決法はまだ与えられていない．

問題（2），（3）も，内部データを考慮していないことに起因している．すなわち，入出力データだけで文を定義しているので，文が表現できる DFD の構造が限定されてしまっている．また，文に対するオペレータについて，各基本的な詳細化の手順を組み合わせた詳細化手順をひとつのオペレータとして形式化しているため，変換により生成可能な DFD の構造も制限されてしまっている．このため，本稿では内部データストアと内部データフローを考慮して，任意の DFD を表現することが可能な文と基本的な詳細化手順を定式化したオペレータから成るプロセス分解代数を提案する．さらに，提案手法をワークステーション上に実現し，例題に対して適用した結果について述べる．

以下，2章では，Adler の論文[4]を中心に，従来の定式化と上記の問題について詳しく述べる．3章では，内部データストアと内部データフローを考慮したプロセス分解代数を提案する．文に対するオペレータを十分に基本的にしただけでは DFD の一部が細分化され過ぎてしまう場合があるので，本提案において抽象化バランスを均等にするオペレータ（抽象化オペレータ）を導入する．4章では，提案手法を UNIX ワークステーション上に実現したシステムについて述べ，このシステムをエレベータ問題[3]と本の発注問題[6]に適用し，DFD を人手で詳細化した場合と比較検討する．5章では，提案したプロセス分解代数により，上記の問題が解決し，適用領域の広い定式化手法が得られたことを示す．

2. 背　景

2.1 Adler による DFD の詳細化作業の定式化

以下では，文献4)に従って，Adler によるプロセス分解代数を用いた DFD の詳細化手順について概観する．はじめに，彼の手法で用いている入出力マトリクス，トランスフォームと文について述べる．

入出力マトリクスとは，コンテクストダイアグラムにおける入出力データの対応関係を記述したマトリクスである．マトリクスの i 行ベクトルは第 i 入力データに依存するすべての出力データを示し，j 列ベクトルは第 j 出力データを得るため必要なすべての入力データを示す．

トランスフォーム（transform）T は，入力データあるいはトランスフォームの並び T_1 と，出力データあるいはトランスフォームの並び T_2 を→で結んだ形 $(T_1 \rightarrow T_2)$ として再帰的に定義される．例えば，a, b, c を入力データ，x，z を出力データとしたとき，

$$(a, b \rightarrow x, (c \rightarrow z))$$

は，トランスフォームとなる．トランスフォームの並びをトランスフォームリストと呼ぶ．例えば，

$$(a, b \rightarrow x), (c \rightarrow z)$$

はトランスフォームリストである．文は，トランスフォームまたはトランスフォームリストである．

Adler の手法では，コンテクストダイアグラムの代わりに，入出力マトリクスが設計者により与えられるとしている．まず，入出力マトリクスの各行ベクトルに対応した文を生成する．この文を初期値として6種類のオペレータを繰り返し適用し，文を変換する．これらのオペレータは，De Marco の経験則に従って設計者が DFD を詳細化する作業を形式化したものである．文の変換では，入出力マトリクスを保存するようにオペレータの適用順序が定められている．適用できるオペレータが存在しないと文の変換は停止する．この変換の結果得られた文をグラフ表現することにより，「最も抽象的な DFD」が得られる．

文の意味を解釈する方法として，マトリクス解釈とグラフ解釈がある．マトリクス解釈は，文を解析し，入出力データの依存関係を示すマトリクスを与える．変換後の文から得られるマトリクスと最初に与えられた入出力マトリクスを比較し，入出力データの依存関係が保存されていることを検証する．グラフ解釈は，文の意味をグラフ表現し DFD を与える．

2.2 従来の定式化における問題点

前節に述べた Adler の定式化手法には，次のような問題がある．

問題1—内部データを取り扱えない．

一般に，プログラムでは，モジュール内部で参照する内部データがある．内部データには，モジュールが繰り返し起動されたときに，初めに割り付けられた領域の値が毎回使われるデータ（ファイルなど）とモジュールの起動ごとに生成消滅するデータがある．モジュールを DFD で表現したとき，前者はデータストアとなり，後者は内部データフローとなる．

Adler の定式化手法では，いずれの内部データも考慮していない．このため，他のすべての行ベクトルと共通出力データを持たない孤立行ベクトルが入出力マトリクスに存在すると，関連しない2つのプロセスが生成されてしまう．これは，関連するデータに基づいてコンテクトダイアグラムを作成しなければならないという De Marco の定義[1]に反することになる．

問題2—変換途中に，入出力マトリクスの変更が必要になる場合がある．

Adler の定式化では，文の定義から，グラフ解釈により文が与える DFD の形状は木になる．このため，与えられた入出力マトリクスによっては，設計者がマトリクスを変換の途中で変更しなければならないことがある．

問題3—既存の変換結果を再利用できない場合が生じる．

大規模ソフトウェア開発では，既存ソフトウェアに新たなデータを加えるなどの修正を施したとき，設計を最初からやり直すのではなく，この既存のソフトウェアを部品として使えることが望ましい．しかし，Adler の手法では，オペレータの適用順序がヒューリスティクスにより一意に定められ，さらに，変換の入力として受け付けられる文に制限があるので，既存の変換結果を再度入力として用いることができない場合がある．このため，設計の追加や変更の際に，前の結果を使えないときには，はじめから変換をやり直さなければならないことになる．

松本と本位田は，文献5)で問題1を指摘し，設計者が入出力マトリクスにデータストアを加え，入出力データとの関連を記述させることにより，孤立行ベクトルを取り扱う手法を示している．しかし，内部データフローの問題および問題2，3については，まだ解決法が与えられていない．5章では，上記問題に対して例を用いて詳述し，さらに提案手法による解決法について議論する．

3. データストアと内部データフローを考慮したプロセス分解代数

3.1 DFD 詳細化の定式化手法の概要

本章では，プロセス分解代数の文において，内部データフローを陽に取り扱うために新たな記号（局所記号という）を導入し，前章に示した問題を解決する定式化手法を提案する．

本手法での目標は，Adler の提案と同様に，入出力マトリクスから，外部からの入出力データを扱うプロセスが一意に定まった「最も抽象的な DFD」を求めることである．ただし，本稿では内部データフローを考慮するため，「最も抽象的な DFD」の概念をプロセスとデータフローの各々に分割して整理した「抽象化バランスが均等な DFD」を作成目標として設定する．プロセスとデータフローについて抽象化バランスが均等である DFD を「抽象化バランスが均等な DFD」という．以下に，プロセスとデータフローの各々についての抽象化バランスの均等性について述べる．

次の2つの条件を同時に満たすとき，DFD 内にあるプロセスが必要以上に細分化されていないとみなせるので，プロセスについての抽象化バランスが均等であるという．

（1） DFD 内の任意の2つのプロセスを1つにまとめたとき入出力データの依存関係が満足されなくなる．

（2） 内部データフローのみを入出力とするプロセスが存在しない．

次の2つの条件を同時に満たすとき，不必要なデータフローが DFD 内に存在しないとみなせるので，データフローについての抽象化バランスが均等であるという．

（1） DFD において冗長なフローがない．

（2） すべての潜在的なフロー以外の冗長なフローが存在しない．

ここで，冗長なフローとは，削除しても入出力マトリクスが変わらないデータフローであり，特に，3つのプロセスA，B，Cで，AからB，BからC，AからCへのデータフローが存在し，AからB，または，AからCへのいずれかのデータフローを削除しても入出力マトリクスが変わらないときは，AからCへのデータフローを冗長なフローとする．また，潜在的なフローとは，プロセス間に新たに加えても入出力マトリクスが変わらないデータフローである．

次に，提案手法により DFD を得る過程を示す（**図1**参照）．ここでは，詳細化作業の全体像を理解できるように概要のみを示し，提案手法における文，文の解釈法，文に対するオペレータの詳細については次節以降の各節で述べる．まず，設計者が，入出力マトリクスを与える．さらに，文献5)と同じ方法で，データストアが必要ならば入出力マトリクスに加える．このマトリクスから文を生成し，オペレータを繰り返し適用する．適用可能なオペレータが複数存在する場合には，任意のオペレータを適用することが許される．適用可能なオペレータが存在しなくなると，文の変換は停止し，正規文と呼ぶ文が得られる．正規文のグラフ解釈で得られる DFD は，De Marco らの指摘した経験則に従った詳細化作業により得られる DFD，すなわち，本手法の目標である，外部からの入出力データを取り扱うプロセスが一意に定まった「抽象化バランスが均等な」DFD になる．

図1で，拡張マトリクス解釈とは，文を解析し，内部データフローに対する局所記号と入出力データおよびデータストアの各データの依存関係を表現するマトリクスを与える．また，マトリクス解釈とは，文を解析し，入出力データとデータストアの記号から成るデータの依存関係を表すマトリクスを与える．文の変換において，文のマトリクス解釈により得られるマトリクスと入出力マトリクスを比較して，入出力データの依存関係が保存されていることを検証する．

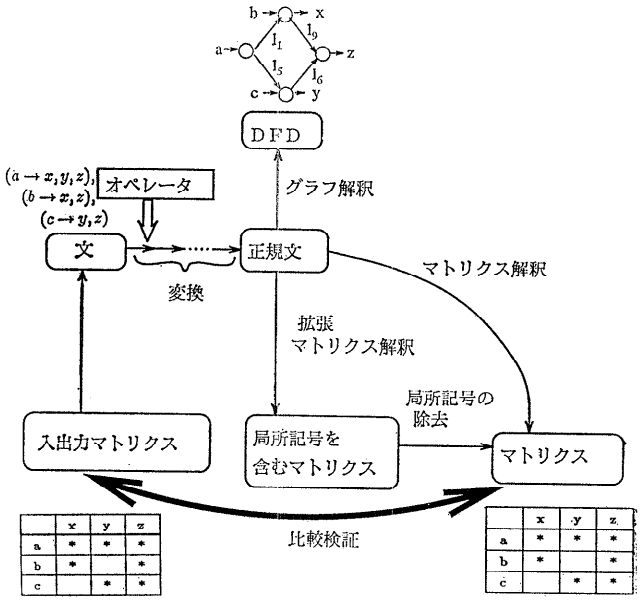


図 1 提案手法の概要

Fig. 1 An overview of DFD refinement based on the proposed process decomposition algebra.

3.2 文

コンテクストダイアグラムの入出力データに対する記号を，s と l 以外のアルファベットの小文字 $a, b, c, \cdots, x, y, z$ で表し，データストアに対する記号を $s_1, \cdots, s_n$ とする．上記の2種類の記号を真入出力記号と呼ぶ．また，内部データフローに対応する記号を局所記号と呼び，l に自然数をつけて，$l_1, l_2, \cdots$ のように表す．以下では，単に，入力記号というときは真入力記号と入力として使われる局所記号の両方を指し，出力記号というときは真出力記号と出力として使われる局所記号の両方を指すことにする．

1つ以上の記号の並びを記号列と呼ぶ．本稿では，記号をコンマ(,)で区切って並びを表現する．入力記号の記号列と出力記号の記号列を矢印(→)で結んだものを項と呼ぶ．例えば，$(a, b, c \rightarrow l_1, x)$ や $(a, l_1 \rightarrow l_1, x)$ は項である．

文は，1つ以上の項の並びである．また，文中に局所記号が出現しない文，あるいは，文中で局所記号が

出現する場合には，すべての局所記号について，同じ局所記号が入力記号と出力記号の両方として文中で使用されている文を閉じた文と呼ぶ．さらに，コンテクストダイアグラムの入出力データやデータストアを扱うプロセスが一意に定まり，かつすべての内部データフローの両端がプロセスに接続されている DFD を与える文は，良形であるという．良形な文は次の条件を満たす文である．

（1） 文を構成するすべての項の→の左辺に，同じ真入力記号が1度だけ出現する．

（2） すべての項の→の右辺に，同じ真出力記号が1度だけ出現する．

（3） 閉じた文である．

例えば，$(a\rightarrow l_1), (l_1, b\rightarrow z)$ は良形な文であるが $(a\rightarrow l_1, z), (l_1, b\rightarrow z)$ や $(a\rightarrow l_1), (l_1, l_2, b\rightarrow z)$ は良形な文ではない．

3.3 文の解釈法

本節では，局所記号を考慮したマトリクス解釈とグラフ解釈について述べる．

まず，拡張マトリクス解釈は，Adler のマトリクス解釈と同様の方法で局所記号を含む文を解析し，局所記号と入出力データから成る要素に対して相互依存関係を表すマトリクスを与える．例えば，拡張マトリクス解釈により，文 $(a\rightarrow l_1, y)$, $(l_1, b\rightarrow z)$ から**図 2-a** を得る．

マトリクス解釈は，拡張マトリクス解釈により得られるマトリクスから，次のようにして局所記号を取り除き，真入出力記号の依存関係を表すマトリクスを与える．すなわち，マトリクス中のすべての局所記号に対して，その局所記号を介して真入力記号から真出力記号へ到達可能か調べ，到達可能ならばその真入出力記号に依存関係があることをマトリクスに明示する．

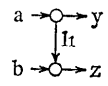
* はデータの依存関係の存在を意味する

	y	z	l_1
a	*		*
b		*	
l_1		*	

(a)

	y	z
a	*	*
b		*

(b)



(c)

図 2 文の解釈例
（a） 拡張マトリクス解釈例
（b） マトリクス解釈例
（c） グラフ解釈例
Fig. 2 Interpretations of sentences.
(a) Extended matrix interpretation.
(b) Matrix interpretation.
(c) Graph interpretation.

さらに，局所記号に対応する行と列を削除する．例えば，文 $(a\rightarrow l_1, y)$, $(l_1, b\rightarrow z)$ では，図 2-a のマトリクスから図 2-b のマトリクスを得る．

グラフ解釈は，文をグラフ表現した DFD を与える．本手法のグラフ解釈では，文中の各項をプロセスとし，項の入出力記号をプロセスの入出力データフローとして表現するだけで良い．このため，Adler のグラフ解釈より簡明である．例えば，文 $(a\rightarrow l_1, y)$, $(l_1, b\rightarrow z)$ のグラフ解釈は図 2-c のように，2つのプロセスと1つの内部フローからなるグラフを与える．

文の拡張マトリクス解釈により得られたマトリクスにおいて，マトリクスの行または列に局所記号が存在する場合，ある局所記号 l_i から同じ局所記号 l_i へ到達可能でない文を非循環文と呼ぶ．非循環文は，プロセス間にループが存在しない DFD に対応する．本稿で提案するプロセス分解代数では，閉じた非循環文を対象とする．

DeMarco, Yourdon らが示した構造化分析手法[1),3)]の指針によると，DFD 中にプロセスのみで構成されるループを記述することは望ましくない．このため，本論文においても，この指針に従い，閉じた非循環文を要素とする集合をプロセス分解代数の対象としている．なお，循環文に対して本プロセス分解代数をそのまま適用すると，プロセスの抽象化バランスが均等でない DFD が作られることになる．

マトリクス解釈が同じになる良形な文のうち，項の数がもっとも少なく，かつ局所記号の数がもっとも少ない文を正規文と呼ぶ．例えば，$(a\rightarrow l_1, y)$, $(l_1, b\rightarrow z)$ は正規文であるが，$(a\rightarrow l_1, l_2, y)$, $(l_1, l_2, b\rightarrow z)$ は正規文ではない．

3.4 オペレータ

はじめに，各オペレータの定義のために使う記号について説明する．$A^*, B^*, C^*, C_1^*, \cdots, C_r^*$ などの大文字のアルファベットは空（NULL）または1個以上の入力記号の並びに対応する記号であり，$X^*, Y^*, Z^*, Z_1^*, \cdots, Z_n^*$ などの大文字のアルファベットは空または1個以上の出力記号の並びに対応する記号である．記号の右上の＊の代わりに＋をつけた記号（A^+, B^+ など）は，1個以上の入出力記号の並びを意味する．さらに，$L_1, \cdots, L_n, M_1, \cdots, M_r, L_{11}, \cdots L_{nr}$ は各々局所記号に対応する記号である．

オペレータの機能は，（オペレータ適用パタン▷（適用条件⇒適用結果））という構造により記述される．これは，▷の左辺にパタンマッチで一致する文に対し

表 1 内部変数を考慮したオペレータ
Table 1 Operators.

No.	記述形式
1	$(A^+, B^*\rightarrow Z^+), (A^+, C^*\rightarrow Y^+)\triangleright$ $(A^+\rightarrow L_N, L_M), (L_N, B^*\rightarrow Z^+), (L_M, C^*\rightarrow Y^+)$
2	$(A^+\rightarrow X^*, Z^+), (B^+\rightarrow Y^*, Z^+)\triangleright$ $(A^+\rightarrow X^*, L_N), (B^+\rightarrow Y^*, L_M), (L_N, L_M\rightarrow Z^+)$
3	$(A^+\rightarrow L_1, L_2, Z^*), (B^+\rightarrow Y^+), (L_1, C^+\rightarrow X^+)\triangleright$ $(((R_f(L_1)\cup R_f(L_2)\cup R_f(Z^*))\supseteq R_f(Y^+))\vee(R_b(B^+)\supseteq R_b(A^+)))$ $\wedge\neg(R_b(A^+)\supseteq R_b(B^+))$ $\wedge(((R_f(L_2)\cap R_f(Z^*))\cup R_f(Y^+))\supset R_f(L_1)\cup(R_f(L_2)\cap R_f(Z^*)))$ $\Rightarrow(A^+\rightarrow L_3, L_2, Z^*), (L_3, B^+\rightarrow Y^+), (C^+\rightarrow X^+)$
4	$(L_1, L_2, A^*\rightarrow Z^+), (B^+\rightarrow Y^+), (C^+\rightarrow L_1, X^+)\triangleright$ $(((R_b(L_1)\cup R_b(L_2)\cup R_b(A^*))\supseteq R_b(B^+))\vee(R_f(Y^+)\supseteq R_f(Z_+)))$ $\wedge\neg(R_f(Z^+)\supseteq R_f(Y^+))$ $\wedge(((R_b(L_2)\cap R_b(A^*))\cup R_b(B^+))\supset R_b(L_1)\cup(R_b(L_2)\cap R_b(A^*)))$ $\Rightarrow(L_3, L_2, A^*\rightarrow Z^+), (B^+\rightarrow Y^+, L_3), (C^+\rightarrow X^+)$
5	$(A^+\rightarrow L_1, Z^+), (L_1, B^+\rightarrow Y^+)\triangleright$ $((R_f(Z^+)\supseteq R_f(Y^+))\vee(R_b(B^+)\supseteq R_b(A^+)))$ $\Rightarrow(A^+\rightarrow Z^+), (B^+\rightarrow Y^+)$
6	$(A^+\rightarrow L_1), (L_1, B^*\rightarrow Z^+)\triangleright(A^+, B^*\rightarrow Z^+)$
7	$(A^+\rightarrow L_1, Z^*), (L_1\rightarrow Y^+)\triangleright(A^+\rightarrow Y^+, Z^*)$
8	$(A_1^+\rightarrow L_1, Z_1^*), \cdots, (A_n^+\rightarrow L_n, Z_n^*), (C_1^*, M_1, \rightarrow Y_1^+), \cdots,$ $(C_r^*, M_r\rightarrow Y_r^+), (L_1, \cdots, L_n\rightarrow M_1, \cdots, M_r)$ $\triangleright$ $(A_1^+\rightarrow L_{11}, \cdots, L_{1r}, Z_1^*), \cdots, (A_n^+\rightarrow L_{n1}, \cdots, L_{nr}, Z_n^*),$ $(C_1^*, L_{11}, \cdots, L_{n1}\rightarrow Y_1^+), \cdots, (C_r^*, L_{1r}, \cdots, L_{nr}\rightarrow Y_r^+)$

て，▷の右辺の適用条件を満足する場合に，このオペレータが適用可能であるという．適用条件部がない場合，すなわち（オペレータ適用パタン▷（⇒適用結果））という構造の場合は，▷の左辺に（オペレータ適用パタン▷適用結果）と略記する．

提案手法では，**表 1** に示す 8 種類のオペレータを用いる．オペレータは，分解オペレータ（オペレータ 1，2）と抽象化オペレータ（オペレータ 3〜8）の 2 種類に分けられる．さらに，各オペレータは，各入出力を処理するプロセスが唯一に定まる DFD で，かつ，「抽象化バランスが均等な DFD」を求めるための操作を，文を変換する規則として表現している．

分解オペレータは，各入出力を処理するプロセスが唯一に定まる DFD を求める操作を形式化している．すなわち，この種類のオペレータは，2 つのグラフに同じ入出力データが存在するとき，それらを 1 つにまとめるため，新たなプロセスを加える操作に対応する．オペレータ 1 は入力に同じデータがある場合の操作で，オペレータ 2 は出力に同じデータがある場合の操作である．オペレータ 1，2 の操作的意味をグラフ表現すると**図 3** になる．オペレータの適用パタン部の項に共通に使われている記号は，この記号に対応する文の入(出)力記号の並びのうちその個数が最大となる並びに対応させる．

抽象化オペレータは，「抽象化バランスが均等な DFD」を求める操作を形式化している．すなわち，この種類のオペレータは，細分化し過ぎたプロセスをまとめる操作と，冗長なフローを削除する操作に対応している．特に，3〜5 の抽象化オペレータは，フローの抽象化バランスを均等化する操作であり，6〜8 の抽象化オペレータは，プロセスの抽象化バランスを均等化する操作である．

ここで，表 1 で抽象化オペレータの適用条件を記述するために用いた関数 R_f, R_b を定義する．R_f は，引数の列の要素が真出力記号である場合にはその要素の記号を与え，局所記号である場合には局所記号から到達可能な真出力記号の集合を与える．R_b は，引数の列の要素が真入力記号である場合にはその要素の記号を与え，局所

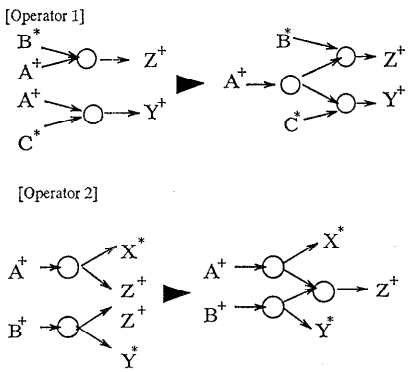


図 3 オペレータ 1，2 の操作的意味
Fig. 3 Operator 1 and 2.

記号である場合には局所記号へ到達可能な真入力記号の集合を与える．例えば，文 $(a\rightarrow l_1, y), (l_1, b\rightarrow z)$ において，$R_f(l_1)$ は $\{z\}$ であり，$R_b(l_1)$ は $\{a\}$ である．また，$\supseteq$ は集合の包含関係を表す．

図4～図7は，表1の抽象化オペレータの意味を例を使ってグラフ表現したものである．オペレータ3，4は，図4に示すように，2つのプロセス間に潜在的なフローLが存在し，フローLを加えることにより，既にあるフロー L_1 が冗長になる場合，フローLを加え（太線で示す），フロー L_1 を削除する（点線で示す）．オペレータ5は，図5に示すように，2つのプロセス間に存在する冗長なフローL_1を削除する．オペレータ6，7は，図6に示すように，2つのプロセスを1つにまとめる．オペレータ8は，図7に示すように，局所記号のみを入出力とするプロセスを削除し，データフローを張り変える．

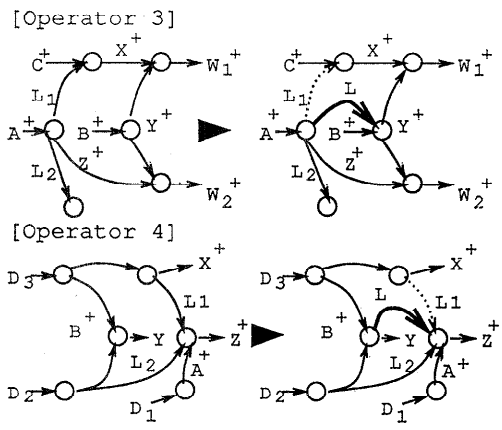


図 4　オペレータ3，4の意味を表す例
Fig. 4　Operator 3 and 4.

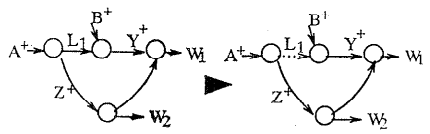


図 5　オペレータ5の意味を表す例
Fig. 5　Operator 5.

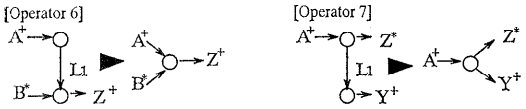


図 6　オペレータ6，7の意味を表す例
Fig. 6　Operator 6 and 7.

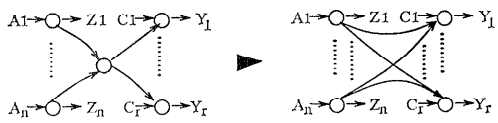


図 7　オペレータ8の意味を表す例
Fig. 7　Operator 8.

4. DFD 詳細化システムの実現と適用実験

4.1 システム構成

3章で述べた手法に基づき DFD を詳細化するシステムを UNIX ワークステーション上に Common Lisp を用いて実現した[7]．このシステムは，初期値文生成部，文変換部，グラフ解釈部，DFD 編集部からなる．以下に，各部の機能概要を述べる．

（1） 初期値文生成部―ユーザ指定の入出力マトリクスを読み込み，各行ベクトルを各々項に変換する．これらの項の並びからなる文（入出力マトリクスに対する初期値文と呼ぶ）を文変換部に与える．

（2） 文変換部―与えられた文を変換し正規文を作成する．ここでは，簡単のため，文に対してオペレータ1から8の順に適用できるか調べ，最初に適用可能となったオペレータを適用する．オペレータを適用した結果得られる文に対して，同様に適用可能なオペレータを見つけて適用する．また，適用可能なオペレータがひとつもなくなると変換を終了する．

（3） グラフ解釈部―文をグラフ解釈し，「抽象化バランスが均等な DFD」を作成する．このとき，DFD 編集部の仕様に従い，DFD の構造を表現する．

（4） DFD 編集部―プロセスとデータフローの接続関係を保存させたまま，DFD 図面の各要素の表示位置を調整することにより，グラフ解釈の結果を見やすい DFD 図面にする機能をユーザに与える．本システムでは，CASE ツール STP[8]の DFD 編集機能を利用している．

4.2 適 用 実 験

前節で述べた DFD 詳細化システムを2つの例題に対して適用した結果について述べる．

（1） エレベータ問題

エレベータ問題は，各ボタンの操作によってエレベータを目的階まで移動させる制御問題である[3]．文献 3) ではこの問題に対する DFD の例として**図8**をあげている．ここでは，この図から入出力マトリクスを**図9**のように設定し，上記システムの入力とする．初期値生成部は，図9のマトリクスから，次のような初期値文を与える．

$(a\rightarrow x, y, s_1), (b\rightarrow x, y, s_1), (s_1\rightarrow s_3), (s_2\rightarrow z, s_2, s_3),$

$(s_3 \rightarrow z,\ s_2,\ s_3)$

上記の文に対して，変換部は**図 10**- a に示すようにオペレータを繰り返し適用し，次の正規文を与える．

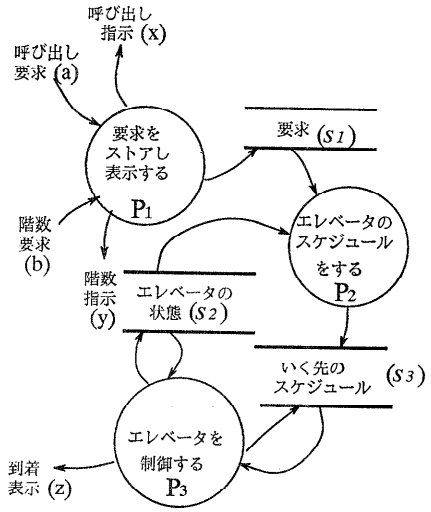


図 8 エレベータ問題の DFD
Fig. 8 Sample DFD : An elevator problem.

	x	y	z	s_1	s_2	s_3
a	*	*		*		
b	*	*		*		
s_1						*
s_2			*		*	*
s_3			*		*	*

図 9 エレベータ問題の入出力マトリクス
Fig. 9 I/O matrix for an elevator problem.

$(a, b \rightarrow x, y, s_1), (s_1, l_{12} \rightarrow s_3), (s_2, s_3 \rightarrow z, s_2, s_3, l_{12}).$

得られた正規文をグラフ解釈し，さらに，DFD 編集部を用いてDFDの表示を見やすく調整すると図 10-b に示す DFD を得る．

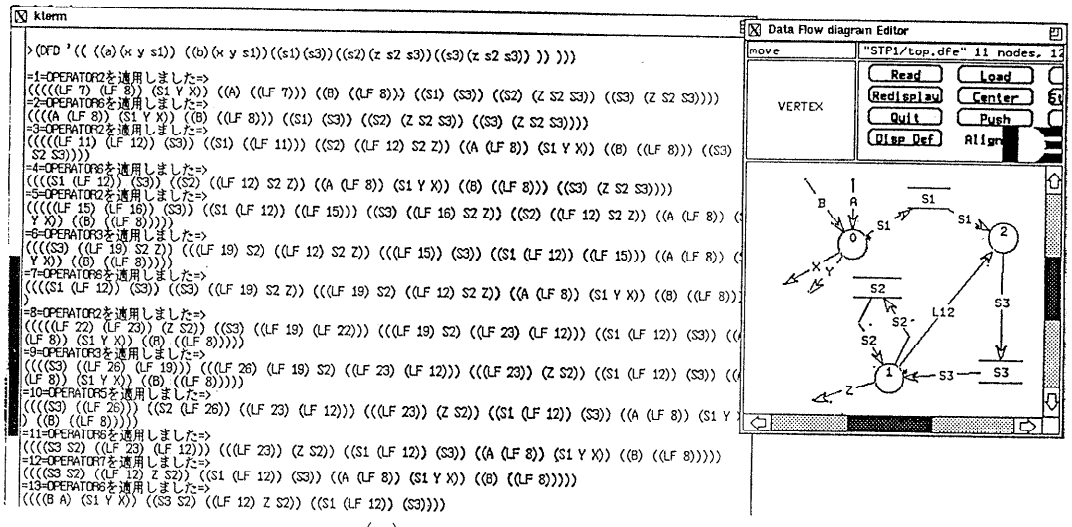
（2） 本の発注問題

本の発注問題は，客が本を注文し，出版社が本を発送するまでをシステム化する問題である[6]．文献6)ではこの問題に対する DFD の例として**図 11** をあげている．図 11 から入出力マトリクスを**図 12** のように設定すると，本システムにより，**図 13** に示す DFD を得る．

図 8 と図 10- b または，図 11 と図 13 を比較したとき，本システムにより，ほぼ等しい DFD が得られることがわかる．ただしこの場合，データストアに対するデータの流れが異なっている．これは，本手法では各データストアを扱うプロセスを入出力各々に対して唯一に定めるとしたためである．

De Marco の経験則では，データストアを扱うプロセス数について何も規定されていない．このため，本論文では以下の理由で，データストアを外部からの入出力データと同様に扱い，データストアを扱うプロセスを唯一に定める．

1. データストアを外部からの入出力データと同様に扱うことで変換系としての議論が簡単になる．
2. この変換系で得られた DFD が持つ性質が明確になる．
3. DFD を得た後で，必要ならば，設計者がデータ



(a) (b)

図 10 エレベータ問題の結果
Fig. 10 Terminal windows drawn by an experimental DFD refinement system for an elevator problem.

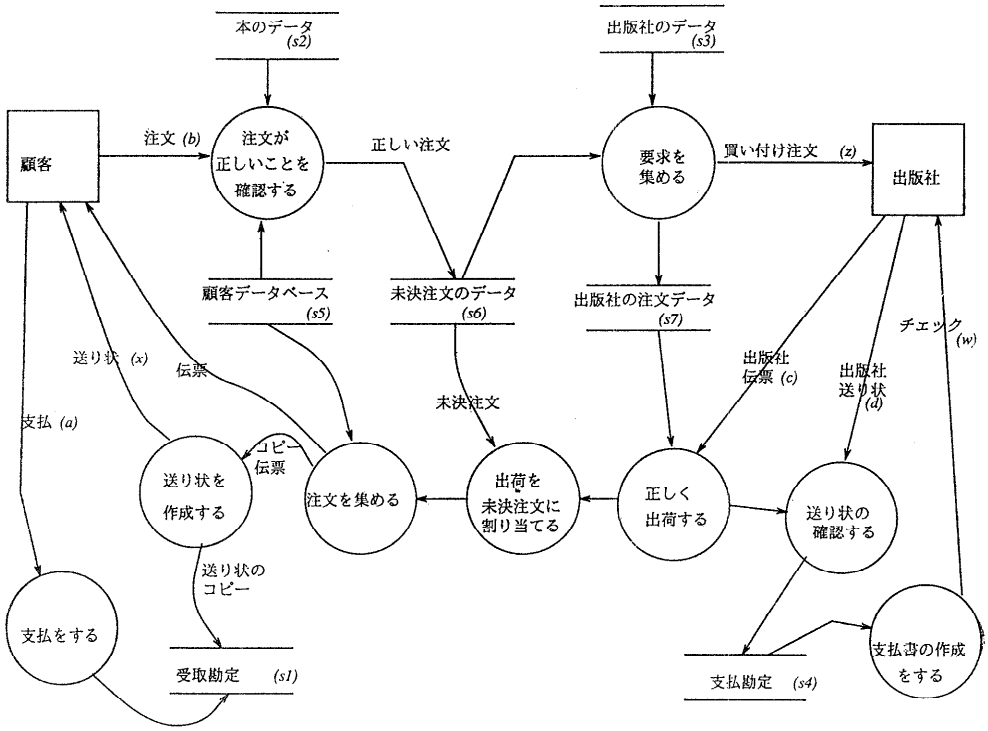


図 11 本の発注問題の DFD
Fig. 11 Sample DFD : A book company.

	x	y	z	w	s_1	s_6	s_7	s_4
a						*		
b						*		
c	*	*			*			*
d								*
s_4				*				
s_5	*	*			*	*		
s_6	*	*	*		*		*	
s_7	*	*			*			*
s_2						*		

図 12 本の発注問題の入出力マトリクス
Fig. 12 I/O matrix for a book company problem.

ストアを扱うプロセスが複数になるように容易に変更できるので，データストアを扱うプロセスを唯一にするとしても，実用上の問題はない．

なお，松本と本位田の文献5)でも，データストアを扱うプロセスに関して同様の手法をとっている．

5. 考　　察

5.1 内部データ（データストア）の考慮

Adler の定式化手法を用いた場合には，データストアに対応した記号が与えられていないので，4章で示したエレベータ問題を解く際，入出力マトリクスは**図14**となり，出力データ z が孤立し，文を変換して DFD を得ることができない．これに対して，提案手法ではデータストアを表す記号を含む文に対してのオペレータを定義しているため，4.2節で示した適用例から分かるように，入出力マトリクスにおいて孤立行を解消し DFD を得ることが可能になっている．

5.2 適用 DFD の範囲の拡大

プロセス分解で得られる DFD の形状について，具体例を用いて従来の問題点と提案手法による解決法を議論する．**図15**-a のような入出力マトリクスを考える．このマトリクスから $(a \rightarrow x, y, z), (b \rightarrow x, z), (c \rightarrow y, z)$ を初期表現として与える．この文に対するプロセス分解を従来手法で行った場合，2つの部分グラフの出力データを1つにまとめるとき，入出力マトリクスを図15-b のように変更することが要求される．この要求に応じると，得られる DFD は**図16**のようになり，最初に与えた入出力マトリクス（図15-a）に対応する DFD（**図17**）ではなくなっている．これに対し，提案手法では，入出力マトリクスの変更要求なしに，2つの部分グラフの出力データを1つにまとめるオペレータ（オペレータ2）を適用することが可能に

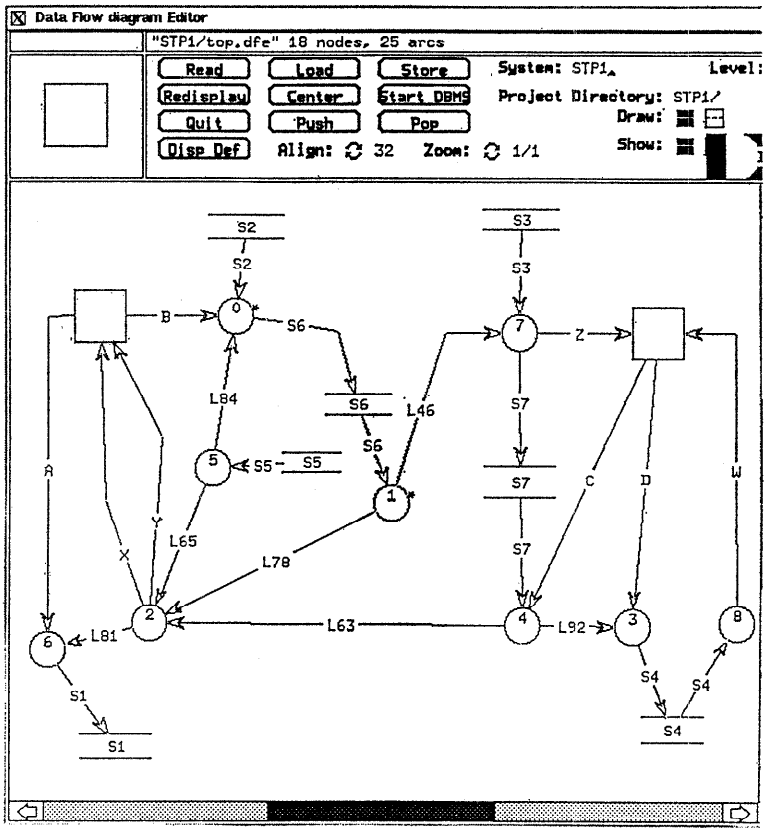


図 13 本の発注問題の結果
Fig. 13 A result of a generated DFD for a book company problem.

	x	y	z
a	*	*	
b	*	*	

図 14 データストアを考慮しない入出力マトリクス
Fig. 14 I/O matrix without data stores.

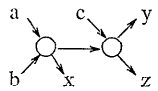


図 16 従来手法で作成される DFD
Fig. 16 A DFD derived conventional methods.

	x	y	z
a	*	*	*
b	*		*
c		*	*

(a) 初期値
(a) Initial matrix.

	x	y	z
a	*	*	*
b	*	*	*
c		*	*

(b) 変更後
(b) Modified matrix.

図 15 初期値と変更後の入出力マトリクス
Fig. 15 An initial I/O matrix and a modified I/O matrix.

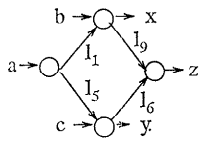


図 17 初期マトリクスに対応した DFD
Fig. 17 A DFD for the initial I/O matrix.

なっているので，変換結果として，図17の DFD が得られる[10]．

5.3 再利用によるインクリメンタルな変換

本手法では，内部フローを局所記号として陽に扱うことにより，任意の閉じた非循環文を初期値とすることを可能にしている．また，各オペレータは，基本的な詳細化手順を形式化しているため，閉じた非循環文が与えられると，オペレータの適用順序によらず変換結果として正規文が得られる[9]．このことは，プロセス分解代数による文の変換が必ず停止することと，変換が停止した場合には文は正規文になっていることを証

明すればよい．まず，停止性は，分解オペレータと抽象化オペレータの作成の仕方により，次のように証明される．分解オペレータは，2つの項に含まれる入出力記号に対して，同じ記号の並びが存在しないと適用されない．分解オペレータを適用すると，2つの項に共通する記号の並びを扱う項を追加し，局所記号の数を2つ増加させる．このとき，上記の同じ記号の並びが存在する項の組みを1つ減少させる．このため，分解オペレータを有限回適用すると，文中の任意の2つの項に共通する記号の並びがなくなるので，分解オペレータしか適用できない場合には，必ず停止する．また，抽象化オペレータは全体の項数または局所記号数を減らすかまたは変化させず，抽象化オペレータしか適用できない場合には，必ず停止する．さらに，分解オペレータと抽象化オペレータを組み合わせた場合，抽象化オペレータは共通に出現する記号の並びを持つ項の組みの数は変えないので，分解オペレータは有限回で適用できなくなる．分解オペレータを適用しないと項数や局所記号数は増加しないので，抽象化オペレータも有限回で適用できなくなる．よって，分解オペレータと抽象化オペレータの組合せによる文の変換は必ず停止する．また，変換が停止しても変換結果が正規文でない場合があると仮定して矛盾を導くことにより，変換が停止した場合に変換結果が正規文であることも証明される．

このため，既存の変換結果に新たな項を追加してできる文が閉じた非循環文であるならば，その文から変換を行っても正規文が得られる．すなわち，この文に対する入出力マトリクスを用いて最初から変換をやり直さずに，既存の結果を利用してインクリメンタルに文を変換しても正規文が得られる．

以下にインクリメンタルな変換の例を示す．**図 18** の入出力マトリクスMを考える．Mに対する初期値文 S_1 に DFD 詳細化システムの文変換部を適用すると，21 ステップで**図 19** の DFD をグラフ解釈とする正規文 $\hat{S}_1$ が得られる．

次に，Mに対して，図 18 の7行目の行ベクトルを加えたマトリクスNを考える．（上記の変換結果である）正規文 $\hat{S}_1$ に7行目の行ベクトルに対応する項 t $(t=(a\rightarrow x, z, w))$ を加えた文を文変換部の入力とする．このとき，4ステップの変換で**図 20** の DFD を与える正規文 $\hat{S}$ が得られる．

マトリクスNに対する初期値文 S_0（S_1 に t を加えた文）に文変換部を適用すると，23ステップで正規文 $\hat{S}$ が得られる．すなわち，この例題では，初期値文から変換をやり直すのに比べて，変換結果の再利用により計算量が約 1/6 に減少している．

	x	y	z	v	w
b		*		*	
c			*		
d	*			*	
e		*			*
f	*		*	*	
g		*		*	*
a	*		*		*

N　　M

図 18 入出力マトリクス
Fig. 18 I/O matrix example for incremental transformation.

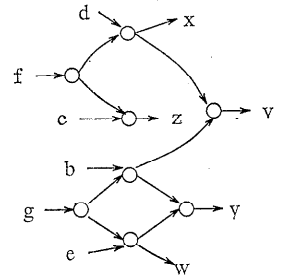


図 19 Mに対する変換結果
Fig. 19 A DFD derived from matrix M.

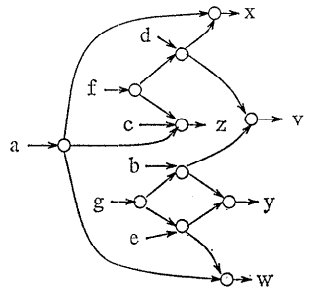


図 20 インクリメンタルな変換結果
Fig. 20 A DFD incrementally derived from a DFD shown Fig. 19 and a new term.

6. おわりに

本稿では，DFD の段階的詳細化に対して，データストアと内部データフローを考慮したプロセス分解代数により形式化する手法を提案した．また，ワークステーション上に実現した DFD 詳細化システムを用いた適用実験により，本手法の有効性を示した．

内部データを考慮したプロセス分解代数の特徴は次

のとおりである．

（1） 入出力データと共にデータストアと内部データフローを表す記号を用いて文を表現し，さらに，文に対するプロセス分解オペレータを十分に基本的な形で与えている．このため，入出力マトリクスに対する文の変換により，任意の構造の DFD を得ることができる．これにより，変換途中で入出力マトリクスを変更しなければならないという問題が解決されている．

（2） 抽象化オペレータを備えることにより，「抽象化バランスが均等な DFD」，すなわち，プロセスとデータフローについて一部だけが必要以上に細分化されることのない DFD を与えている．

（3） 文に対するオペレータの適用順序によらず「抽象化バランスが均等な DFD」が得られる．これにより，既存の設計情報を部品として活用したり，オペレータを並列に適用することにより計算時間を短縮できる．

本稿で述べた文の変換系に対して，(a)必ず正規文が得られること，および，(b)得られる可能性のある正規文の集合において一意に定まる代表元が存在し，この正規分の集合の要素から代表元に変換する手段が与えられることが，保証されている[9]．これらのことから，提案手法により，与えられた入出力マトリクスに対して，「抽象化バランスが均等な DFD」が一意に定められることが示される．

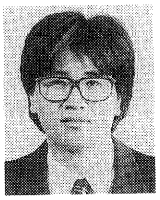
今後の課題として，複数の文の変換過程の中で計算量の最も少ない変換を可能にするため，オペレータの適用順序に対する戦略を明確化することがあげられる．

謝辞 本研究の機会を与えてくださりご支援頂いたソフトウェア研究所ソ技部磯田定宏部長，ならびに国立勉リーダに感謝します．また，山本修一郎主幹員には，プロセス分解についてご討論頂いた．基礎研究所の小川瑞史主任，ソフトウェア研究所の伊藤智子さんに文献5)を教えて頂いたのが，本研究を始めるきっかけとなった．これらの方々ならびに日頃ご討論頂くソ構グループ・ソ知グループの皆様に感謝いたします．

参 考 文 献

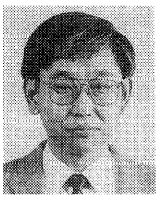
1) De Marco, T.: *Stuctured Analysis and System Specification*, Yourdon, New York (1978).
2) Ward, P. T. and Mellor, S. J.: *Structured Development for Real-Time Systems*, Vol. 2, Yourdon, New York (1985).
3) Yourdon, E.: *Modern Structured Analysis*, Yourdon Press (1989).
4) Adler, M.: An Algebra for Data Flow Diagram Process Decomposition, *IEEE Trans. Softw. Eng.*, Vol. SE-14, No. 2, pp. 169-183 (1988).
5) 松木一教，本位田真一：段階的ソフトウェアの生成と検証について，人工知能学会研究会，SIG-FAI-9002 (1990).
6) Davis, A. M.: *Software Requirements Analysis and Specification*, p. 516, Prentice-Hall (1990).
7) 鈴木英明，高橋直久：プロセス分解代数に基づくデータフロー図の段階的詳細化システムの設計と実現，第 44 回情報処理学会全国大会論文集，No. 5, pp. 337-338 (1992).
8) IDE: Software through Pictures Release 4.2 Reference Manual (1989).
9) 鈴木英明，高橋直久：内部データフローを考慮したプロセス分解代数における変換結果の一意性，日本ソフトウェア科学会プログラム合成変換研究会，pp. 1-12 (1992).
10) 鈴木英明，高橋直久：プロセス分解代数に基づくデータフロー図の段階的詳細化について，情報処理学会ソフトウェア工学研究会，91-SE-82 (1991).

（平成4年 3 月16日受付）
（平成5年 1 月18日採録）



鈴木 英明（正会員）

昭和36年生．昭和60年早稲田大学理工学部数学卒業．同年，NTT 武蔵野電気通信研究所入所．以来，推論機構，定理証明系，プログラム合成，CASE システム，プログラムの抽象化の研究に従事．現在，NTT ソフトウェア研究所研究主任．



高橋 直久（正会員）

昭和26年生．昭和49年電気通信大学応用電子卒業．昭和 51 年同大大学院修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，機能分散型並列計算機，データフロー型計算システム，関数型プログラミング，並列分散 OS，ソフトウェア・リエンジニアリングなどの研究に従事．現在，NTT ソフトウェア研究所主幹研究員．工学博士(東京工業大学)．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．