ソフトウェア工学 82−10
(1991. 12. 3)

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

鈴木英明　高橋直久
NTTソフトウェア研究所

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。さらに、適用例に基づき、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ことを示す。(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。

A Process Decomposition Algebra for Stepwise Refinement of Data Flow Diagrams

Hideaki SUZUKI　Naohisa TAKAHASHI
NTT Software Laboratories

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 experiments on process decompositions 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。つぎに、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おこなうため、次のような点に留意すべきであると述べている[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って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も十分に基本的ではないので、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 を表現可能な文と 十分に基本的なオペレータから成るプロセス分解代数を提案する。

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、提案手法をエレベータ問題[3] 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. DFD の詳細化作業の定式化

以下では、文献 4) に従って、プロセス分解代数における基本的用語を定義した後、プロセス分解代数を用いて DFD を詳細化する手順を概観する。

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。本稿では、並びをコンマ (,) で表す。トランスフォームで、入力データの並びと出力データの並びを矢印 → で結んだ形を項と呼ぶ。例えば、a, b を入力データ、x を出力データとしたとき、

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

は項である。また、a, b, c を入力データ、x, z を出力データとしたとき、$(c \rightarrow z)$ は項であり、

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

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。

つぎに、DFD の詳細化の手順を示す。この形式化では、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、つぎ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、いづ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ス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 が指摘した「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」の概念を拡張した「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が冗長なフローである。

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、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。

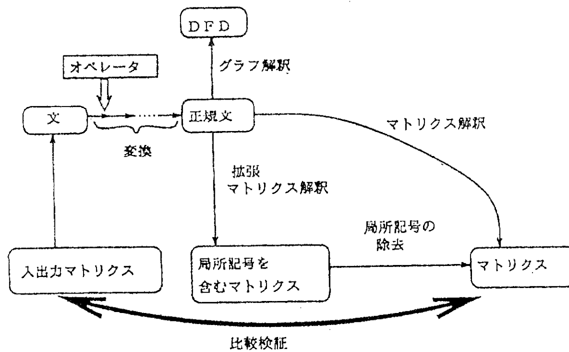


図 1: 提案手法の概要

つぎに、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により、「抽象化バランスが均等な DFD」を得る。

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。

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 $a, b, c, \dots, x, y, z$ で表し、データストアに対する記号を $s_1, \dots, s_n$ 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、$l_1, l_2, \dots$ 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。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。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。

(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) 各局所記号に対して、同じ局所記号がある項の入力と他の項の出力として使用されている。

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 $(a \to l_1, y), (l_1, b \to z)$ から図2を得る。

	y	z	l_1
a	*		*
b		*	
l_1		*	

* はデータの依存関係の存在を意味する

図2: 拡張マトリクス解釈例

	y	z
a	*	*
b		*

図3: マトリクス解釈例

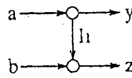


図4: グラフ解釈例

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。さらに、局所記号に対応する行と列を削除する。例えば、文 $(a \to l_1, y), (l_1, b \to z)$ では、図2のマトリクスから図3のマトリクスを得る。

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、$(a \to l_1, y), (l_1, b \to z)$ は正規文であるが、$(a \to l_1, l_2, y), (l_1, l_2, b \to z)$ は正規文ではない。

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 $(a \to l_1, y), (l_1, b \to z)$ のグラフ解釈は図4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。

3.4. オペレータ

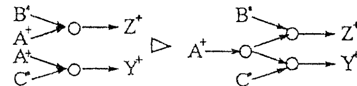
はじめに、各オペレータの定義のために使う記号について説明する。$A^*, B^*, C^*, C_1^*, ..., C_r^*$ などの大文字のアルファベットは空 (NULL) または1個以上の入力記号の並びに対応する記号であり、$X^*, Y^*, Z^*, Z_1^*, ..., Z_n^*$ などの大文字のアルファベットは空または1個以上の出力記号の並びに対応する記号である。記号の右上の $*$ の代わりに $+$ をつけた記号 (A^+, B^+ など) は、1個以上の入出力記号の並びを意味する。さらに、$L_1, ..., L_n, M_1, ..., M_r, L_{11}, ..., L_{nr}$ 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 ▷ (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て、オペレータが起動され、▷ の右辺の条件を満足する場合のみ、記号 ⇒ の右辺の文に変換し、それ以外は変換しないことを意味する。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、(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。

分解オペレータは、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5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。

[オペレータ1]



[オペレータ2]

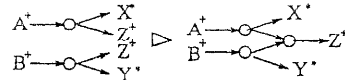


図5: オペレータ 1,2 の操作的意味

抽象化オペレータは、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、DFD の抽象化バランスの均等性を保つものである。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。

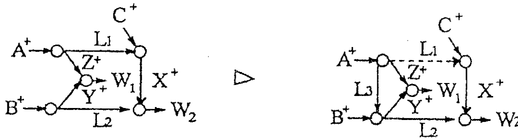
ここで、表1の抽象化オペレータで使用する関数 R_f、R_b を定義する。R_f は、引数の列の要素が真出力記号である場合にはその要素の記号を与え、局所記号である場合には局所記号から到達可能な真出力記号の集合を与える。R_b は、引数の列の要素が真入力記号である場合にはその要素

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。

表1：内部変数を考慮したオペレータ

オペレータ名	記述形式
オペレータ1	$(A^+, B^* \rightarrow Z^+), (A^+, C^* \rightarrow Y^+) \triangleright$ $(A^+ \rightarrow L_N, L_M), (L_N, B^* \rightarrow Z^+), (L_M, C^* \rightarrow Y^+)$
オペレータ2	$(A^+ \rightarrow X^*, Z^+), (B^+ \rightarrow Y^*, Z^+) \triangleright$ $(A^+ \rightarrow X^*, L_N), (B^+ \rightarrow Y^*, L_M), (L_N, L_M \rightarrow Z^+)$
オペレータ3	$(A^+ \rightarrow L_1, Z^+), (B^+ \rightarrow L_2, Y^+), (L_1, C^+ \rightarrow X^+)$ $\triangleright ((Z^+ \cap B^+ = \emptyset) \wedge ((R_f(L_1) \cup R_f(Z^+) \supseteq R_f(L_2) \cup$ $R_f(Y^+)) \vee (R_b(B^+) \supseteq R_b(A^+))) \wedge (A^+ \cap L_2 = \emptyset)$ $\wedge (R_f(L_2) \supseteq R_f(L_1))) \Rightarrow$ $(A^+ \rightarrow L_3, Z^+), (L_3, B^+ \rightarrow L_2, Y^+), (C^+ \rightarrow X^+)$
オペレータ4	$(L_1, A^+ \rightarrow Z^+), (L_2, B^+ \rightarrow Y^+), (C^+ \rightarrow L_1, X^+)$ $\triangleright ((A^+ \cap Y^+ = \emptyset) \wedge ((R_b(L_1) \cup R_b(A^+) \supseteq R_b(L_2) \cup$ $R_b(B^+)) \vee (R_f(Y^+) \supseteq R_f(Z^+))) \wedge (Z^+ \cap L_2 = \emptyset)$ $\wedge (R_b(L_2) \supseteq R_b(L_1))) \Rightarrow$ $(L_3, A^+ \rightarrow Z^+), (L_2, B^+ \rightarrow Y^+, L_3), (C^+ \rightarrow X^+)$
オペレータ5	$(A^+ \rightarrow L_1, Z^+), (L_1, B^+ \rightarrow Y^+) \triangleright$ $((R_f(Z^+) \supseteq R_f(Y^+)) \vee (R_b(B^+) \supseteq R_b(A^+)))$ $\Rightarrow (A^+ \rightarrow Z^+), (B^+ \rightarrow Y^+)$
オペレータ6	$(A^+ \rightarrow L_1), (L_1, B^* \rightarrow Z^+) \triangleright (A^+, B^* \rightarrow Z^+)$
オペレータ7	$(A^+ \rightarrow L_1, Z^*), (L_1 \rightarrow Y^+) \triangleright (A^+ \rightarrow Y^+, Z^*)$
オペレータ8	$(A_1^+ \rightarrow L_1, Z_1^*), ..., (A_n^+ \rightarrow L_n, Z_n^*), (C_1^*, M_1 \rightarrow Y_1^+), ...$ $(C_r^*, M_r \rightarrow Y_r^+), (L_1, ..., L_n \rightarrow M_1, ..., M_r)$ $\triangleright$ $(A_1^+ \rightarrow L_{11}, ..., L_{1r}, Z_1^*), ..., (A_n^+ \rightarrow L_{n1}, ..., L_{nr}, Z_n^*),$ $(C_1^*, L_{11}, ..., L_{n1} \rightarrow Y_1^+), ..., (C_r^*, L_{1r}, ..., L_{nr} \rightarrow Y_r^+)$

[オペレータ3]



[オペレータ4]

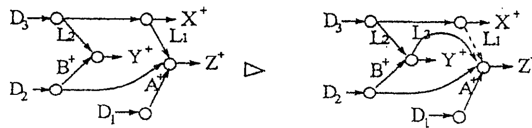


図6: オペレータ3,4の意味を表す例

図6〜図9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6に示すように、2つのプロセス間に潜在的なフロー L_3 が存在し、フロー L_3 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_3 を加え、フロー L_1 を削除する。オペレータ5は、図7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8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9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。

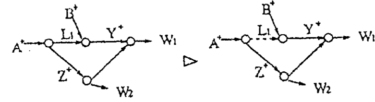
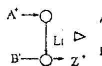


図7: オペレータ5の意味を表す例

[オペレータ6]



[オペレータ7]

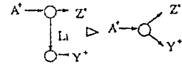


図8: オペレータ6,7の意味を表す例

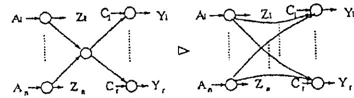


図9: オペレータ8の意味を表す例

4. 適用実験

本章では、3章で述べたプロセス分解代数を、エレベータ問題[3]に対して、適用した例について述べる。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。文献 3) ではこの問題に対するDFDの例として図10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11-aのように設定し、それに対して提案手法を適用してDFDを求める。図11-aのマトリクスから、次のような文の初期値を得る。

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

上記の文に対して、次のようにオペレータを繰り返し適用できる。ここで、$\triangleright$ の上に記した $RuleN$ はオペレータ N を適用することを意味する。

$$(a, b \rightarrow x, y, s_1), (s_1 \rightarrow s_3), (s_2, s_3 \rightarrow z, s_2, s_3) \overset{\text{Rule 2}}{\triangleright}$$
$$(a, b \rightarrow x, y, s_1), (s_1 \rightarrow l_1), (l_1, l_2 \rightarrow s_3), (s_2, s_3 \rightarrow z, s_2, l_2$$
$$\overset{\text{Rule 6}}{\triangleright} (a, b \rightarrow x, y, s_1), (s_1, l_2 \rightarrow s_3), (s_2, s_3 \rightarrow z, s_2, l_2)$$

得られた文からグラフ解釈により図12に示すDFDを得る。図10と図12を比較したとき、提案手法により、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。

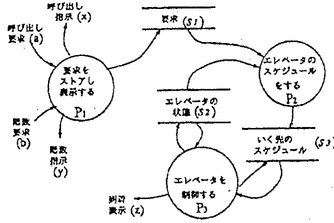


図 10: エレベータ問題の DFD

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

(a) データストアを追加した場合

	x	y	z
a	*	*	
b	*	*	

(b) 外部データのみの場合

図 11: エレベータ問題の入出力マトリクス

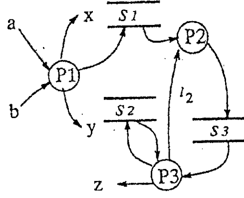


図 12: エレベータ問題に提案手法を適用した DFD

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１１-b 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、5章で示した文の変換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１３-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 13-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 14-b のようになり、最初に与えた入出力マトリクス (図 13-a) に対応する DFD (図 14-a) 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なっている。以下に変換例を示す。この結果、図 14-a の DFD を得る。

$$(a \rightarrow x, y, z), (b \rightarrow x, z), (c \rightarrow y, z)$$
$$\overset{\text{Rule 2}}{\triangleright} (a \rightarrow l_1, y), (b \rightarrow l_2), (c \rightarrow y, z), (l_1, l_2 \rightarrow x, z)$$
$$\overset{\text{Rule 6}}{\triangleright} (a \rightarrow l_1, y), (c \rightarrow y, z), (l_1, b \rightarrow x, z)$$
$$\overset{\text{Rule 2}}{\triangleright} (a \rightarrow l_1, l_3), (c \rightarrow l_4, z), (l_1, b \rightarrow x, z), (l_3, l_4 \rightarrow y)$$
$$\overset{\text{Rule 3}}{\triangleright} (a \rightarrow l_1, l_5), (l_5, c \rightarrow l_4, z), (l_1, b \rightarrow x, z), (l_4 \rightarrow y)$$
$$\overset{\text{Rule 6}}{\triangleright} (a \rightarrow l_1, l_5), (l_5, c \rightarrow y, z), (l_1, b \rightarrow x, z)$$
$$\overset{\text{Rule 2}}{\triangleright} (a \rightarrow l_1, l_5), (l_5, c \rightarrow y, l_6), (l_1, b \rightarrow x, l_7), (l_6, l_7 \rightarrow z)$$

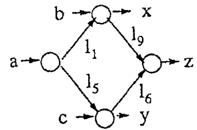
	x	y	z
a	*	*	*
b	*		*
c		*	*

(a) 初期値

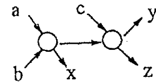
	x	y	z
a	*	*	*
b	*	*	*
c		*	*

(b) 変更後

図 13: 初期値と変更後の入出力マトリクス



(a) 初期マトリクスに対応した DFD



(b) 従来手法

図 14: 作成される DFD

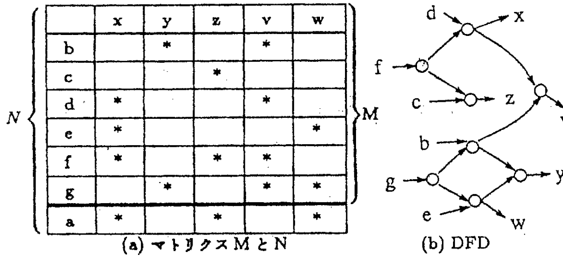
5.3. 再利用によるインクリメンタルな変換

本手法では、文に対して適用可能ならば任意のオペレータを適用することを許しているため、適用オペレータの選択法により文の変換過程が複数存在することになる。しかし、どのような文を与えた場合でも、すべての変換過程により得られる文は同一の正規文になる。今、入出力マトリクスMが与えられたとき、Mを分割したマトリクスに対応する文 A と B に対して、各々を変換した結果を $\hat{A}$、$\hat{B}$ とする。変換結果の一意性により、$\hat{A}$ と $\hat{B}$ を並べた文にオペレータを適用して変換した結果と、マトリクスMに対応する A と B を並べた文をそのまま変換した結果が同じになる。変換結果の一意性は、文の項数と局所記号の数による帰納法と真入出力記号の個数による帰納法により証明される。ここでは、例を用いて変換結果の一意性により既存のプロセス分解結果を再利用できることを示し、証明の詳細については別途報告する。

図１５-a の入出力マトリクス M を与える。Mに対する文にプロセス分解を施すと、9ステップで、以下の文になり、図１５-b の DFD となる。

$$(b \rightarrow y, v), (c \rightarrow z), (d \rightarrow x, v), (e \rightarrow y, w),$$
$$(f \rightarrow x, z, w), (g \rightarrow y, v, w) \overset{\text{Rule 2}}{\triangleright} \ldots \overset{\text{Rule 6}}{\triangleright}$$

$(l_1, b \rightarrow l_3, l_7), (l_5, c \rightarrow z), (l_2, d \rightarrow x, l_8), (l_6, e \rightarrow l_4, w),$
$(f \rightarrow l_5, l_2), (g \rightarrow l_1, l_6), (l_3, l_4 \rightarrow y), (l_7, l_8 \rightarrow v)$



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

(a) マトリクスMとN　(b) DFD

図15: 入出力マトリクス(a)とそれに対するDFD(b)

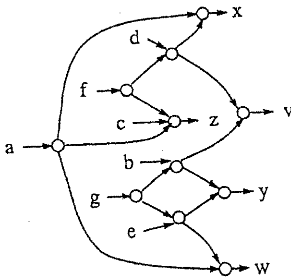


図16: インクリメンタルな変換結果のDFD

次に、Mに対して、7行目の行ベクトルを加えたマトリクスNを考える。すでに変換済みの文に、新たに付け加えた行ベクトルに対応する項$(a \rightarrow x, z, w)$を加え、オペレータ2を4回繰り返し適用する。その変換結果はつぎのようになり、図16のDFDを得る。

$(l_1, b \rightarrow l_3, l_7), (l_5, c \rightarrow z), (l_2, d \rightarrow x, l_8),$
$(l_6, e \rightarrow l_4, w), (f \rightarrow l_5, l_2), (g \rightarrow l_1, l_6),$
$(l_3, l_4 \rightarrow y), (l_7, l_8 \rightarrow v), (a \rightarrow x, z, w)$

Rule 2 ▷ Rule 2 ▷ Rule 2 ▷ Rule 2 ▷

$(l_1, b \rightarrow l_3, l_7), (l_2, d \rightarrow l_9, l_8), (l_6, e \rightarrow l_4, l_{14}),$
$(f \rightarrow l_5, l_2), (g \rightarrow l_1, l_6), (l_3, l_4 \rightarrow y), (l_7, l_8 \rightarrow v),$
$(a \rightarrow l_{10}, l_{11}, l_{13}), (l_9, l_{10} \rightarrow x), (l_{11}, l_5, c \rightarrow z),$
$(l_{13}, l_{14} \rightarrow w)$

マトリクスNに対する文の変換を最初から行なった場合、16ステップで上記と同じ文を得る。すなわち、マトリクスNを用いて変換をやり直すと16ステップかかるが、マトリクスMを用いた変換結果を再利用した場合、4ステップとなり、再利用により計算量が1/4に少なくなっていることがわかる。

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。この手法の特徴は次の通りである。

(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変換結果が一意に定まる。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することが可能である。

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。また、提案手法を計算機上で実現したシステムのプロトタイプは現在構築中である。

謝辞　本研究の機会を与えてくださり御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御討論頂いた。小川瑞史主任、保田智子社員に文献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御討論頂くソ構グループ・ソ知グループの皆様に感謝致します。

[参考文献]

1)T. De Marco: *Structured Analysis and System Specification,* New York Yourdon, 1978

2)P.T. Ward and S.J. Mellor: *Structured Development for Real-Time Systems,*vol. 2. New York Yourdon, 1985

3)E. Yourdon : *Modern Structured Analysis,* YOURDON PRESS, 1989

4)M. Adler: An Algebra for Data Flow Diagram Process Decomposition, IEEE Trans. on Software Engineering, SE-14, No.2, 1988

5) 松本一教、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、February,1990