ICOT Technical Memorandom: TM-1035

TM-1035

段階的ソフトウェアの生成と検証について

松本　一教、本位田　真一（東芝）

March, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

段階的ソフトウェアの生成と検証について
On Stepwise Software Construction and its Verification

松本一教　　本位田真一
Kazunori MATSUMOTO and Shin-ichi HON-IDEN

(株) 東芝
システム・ソフトウェア技術研究所
Systems and Software Engineering Lab., TOSHIBA Corp.

ABSTRACT

The central issue of this paper is a methodology for the use of I/O matrix, data hierarchies and process hierarchies in stepwise data-flow diagram(DFD) design. Defining DFD, the simplest diagram ,which is called the context diagram, is firstly identified. This diagram clears the scope of the process from the view point of the data. And then, it is stepwisely decomposed, in the successively design phase, to logically implement the target software. This decomposition consists of two different kinds methods. One is process-decomposition and the other is data- decomposition. We propose a formal strategy for DFD process decomposition with internal data stores. This formalization is motivated the use of I/O matrix, which is originally proposed by Adler. However his study does not include the idea of data stores and data decomposition. Then, we develop an I/O matrix analysis technique to introduce necessary data store. Moreover, process hierarchies and data hierarchies are defined and used to verify the data decomposition process.

はじめに

ソフトウェアの生成は、抽象的であいまいな仕様記述を段階的に詳細かつ具体的な記述へと変換し、最終的には計算機上において実行可能なコードを得るという段階的なプロセスとしてとらえることができる。さまざまなソフトウェア仕様記述方式において、このような段階的なプロセスを支援する方式が提案されている*[Reisig86,Kramer90]*。しかしながらこれまでの提案においては、*informal*な形式での議論がほとんどであったため、各設計者ごとに独自の方式にもとづくソフトウェア作成が行なわれているという現状である。そこで本稿では、*SA*におけるデータフローダイアグラム*(DFD)*の作成方式に限定して、ソフトウェアの段階的作成をある程度形式的に議論し、その検証についても言及する。

*SA*は*DeMarco[DeMarco78]*により提案されたものであり、*DFD*、データディクショナリ*(DD)*、ミニ仕様書などから構成される。対象ソフトウェアの仕様記述は、まずもっとも上位の*DFD*を記述することにより始まる。このもっとも上位の*DFD*をとくにコンテクストダイアグラムという。次に、コンテクストダイアグラムを、次のレベルに分解した*DFD*を作成する。このときに得られた*DFD*中の*1*つのサブプロセスを新たなコンテクストダイアグラムとみなすことにより、同様の分解が繰り返される。ここで、*DFD*の分解においては、機能に関する分解とデータに関する分解の2つがあることに注意しておく。前者をプロセス分解とよび後者をデータ分解とよぶ。分解において要請されることは、親ダイアグラムと子ダイアグラムにおける、入出力のバランス*[DeMarco79]*である。この2種類の分解をどのようなタイミングで適用するかについてはほとんど議論されていない。

1. DFDにおけるプロセス分解

DFDにおける最上層は、単一のダイアグラムであり(図1)コンテクストダイアグラムとよばれる。コンテクストダイアグラムを記述する目的は、対象範囲を明確にするということである。これをさらに下位のプロセスへと分解する操作をプロセス分解(process decomposition)という。図2は図1をプロセス分解して得られたものである。このようなプロセス分解の方式について、[Gane79, Ross76]などにおいて、informalな形式での議論がなされている。

最近、Adler[Adler88]はプロセス分解を厳密に規定する方法論が存在しないことを問題視し、この過程を機械的に行える方式を提案した。そこでは、コンテクストダイアグラムの他にI/Oマトリックスとよばれる記述を用いる。本章では、まずAdlerの方式について概説し、その拡張について考察する。

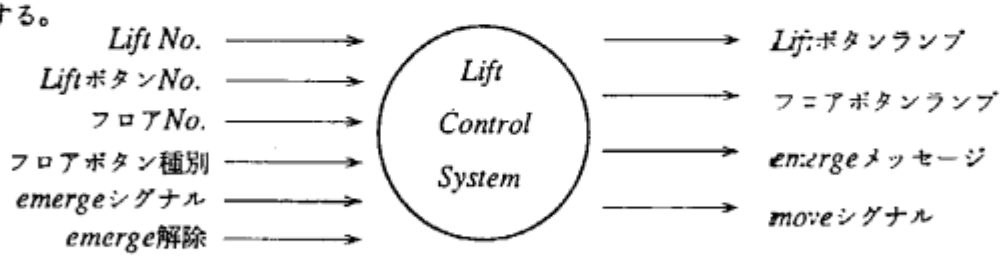


図1. コンテクストダイアグラムの例(リフトの問題)

図2. プロセス分解の例(リフトの問題)

1.1 プロセス分解代数

Adler[Adler88]にしたがって、プロセス分解代数について復習する。まず、基本的な用語を約束する。

I/Oマトリックスとは、図3に示すようなものである。これは、図1のコンテクストダイアグラムに対応し、入力データと出力データの関係を記述したものである。すなわち、第i行ベクトルは入力データiに伴う出力データを示す。第j列ベクトルは、出力データjを得るために必要なすべての入力データを示している。

$$(I^+ \rightarrow O^+)$$

の形式をタームという。ここに$I^+(O^+)$は、入力データ(出力データ)を1個以上ならべたものである。L,Mをタームとするとき、

$$(L \rightarrow M)$$

の形式をsemi-transformという。入力データ記号(出力データ記号)または1個以上のsemi-transformの列をinput-transform list(output-transform list)という。また、これらをまとめてtransform listという。

X,Yをtransform listとするとき、

$(X \rightarrow Y)$

の形式をtransformといい、tarnsformとtransform listをまとめてセンテンスという。

		Lift ボタンランプ(LB)	フロアボタンランプ(FL)	emergeメッセージ(EM)	move (MS)
Lift No.	(LN)	*			*
Lift ボタン No.	(BN)	*			*
フロア No.	(FN)		*		*
フロア ボタン種別	(FB)		*		*
emerge シグナル	(ES)			*	
emerge 解除	(ER)				

図3. リフトの問題におけるI/Oマトリックス

I/Oマトリックスがn行からなるとき、n個のタームからなるtransform listへと変換する。すなわち、第i行の入力データを左辺に、第i行ベクトルの*成分を並べたものを右辺に持つタームを対応させる。図4は、図3のI/Oマトリックスから得られたセンテンスである。

$$(LN \rightarrow LB, MS),\ (BN \rightarrow LB, MS),\ (FN \rightarrow FL, MS),$$
$$(FB \rightarrow FL, MS),\ (ES \rightarrow EM)$$

図4. I/Oマトリックスからセンテンスへの変換

I/Oマトリックスからプロセス分解代数のセンテンスが得られると、それに対して分解オペレータが適用される。分解オペレータには、exact substitution, subset substitution, weak substitutionなど6種類のものがある。これらのオペレータは、プロセス分解において人間の設計者が用いる方法を、プロセス分解代数上のセンテンスを対象として形式化したものである。これらはのオペレータの基本的な方針は、I/Oマトリックスの各成分ごとに部分的なDFDを作成し、それらを1つにまとめあげるというbottom-up的なアイデアに従う(図5)。ここでは、同一の出力を持つプロセスを関連付けるという、exact substitutionについてのみ簡単に紹介する。詳しくは直接文献を参考にされたい。

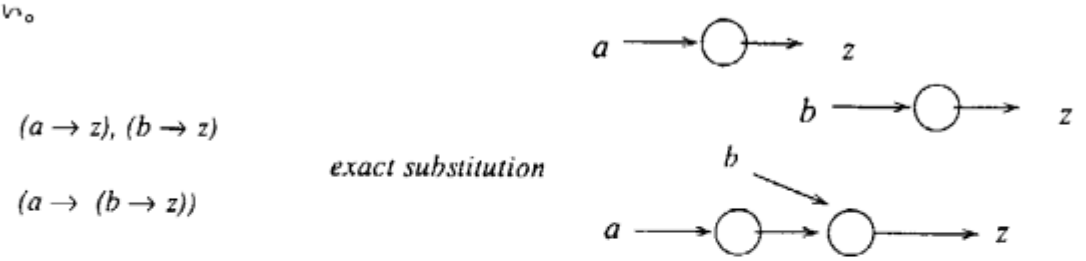


図5. exact substitutionの適用

I/Oマトリックスから得られたセンテンスに対して分解operatorを適用することにより、センテンスの変型を行っていく。変型が終了すると、センテンスからDFDへの変換が行われる。この変換のことをgraph interpretationとよぶ。これに関しては次章で再び議論する。

2. 内部データを考慮したプロセス分解代数

先に紹介したAdlerの方式によれば機械的な方法でプロセス分解を行うことができる。しかしながら、この方法だけでは十分ではない場合もある。すなわち、

(1) 内部データを持つDFDが扱えない

(2) データ分解に関する指針がない

といった問題である。例えば、図1、2、3で示したリフトの問題について考えてみる。図1のコンテクストダイアグラムにおいては、このプロセスに対する(外部からの)入力と(外部への)出力しか明らかにされていないため、プロセスの内部だけで用いられるデータを導入することができない。この内部的なデータはDFDにおけるファイル(データストア)の概念に対応し、現実的なソフトウェア設計においては無視することはできない。本章では、内部的なデータの導入に関して議論し、データストアも含めた分解ができるように、Adlerの方式を拡張する。ここで拡張される方式においては、以下のステップのしたがって分解を行う。

Step1. コンテクストダイアグラムをもとに、I/Oマトリックスを作成する。

Step2. I/Oマトリックスの解析
I/Oマトリックスを解析し、必要ならば内部データを導入する。

Step3. step2で導入された内部データを含めて、DFDの分解を行う。

以下では、各ステップの内容について詳しく議論する。

2.1 I/Oマトリックスの解析と内部データの導入

Step2における、I/Oマトリックスの解析と、必要な内部データを見出すことについて議論する。

Adlerのプロセス分解代数は、各行ごとにプロセスを生成しておき、オペレータの適用によってそれらのうち関連するものを融合するという、bottom-up的な方式であった。このとき、出力データに共通要素がない2つのプロセスは融合されないことに注意されたい。したがって、他のどの行ベクトルとも共通出力データを持たない行ベクトルが存在したとすると、関連するデータのみを洗いだすというコンテクストダイアグラムの性質と矛盾することになる。結局、2つの行ベクトルについて、それらが表す出力データを比較するという操作が重要になってくる。

定義
I/Oマトリックス中の行ベクトルi, jにおいて、iに対応する出力データ集合とjに対応する出力データ集合に共通要素が存在しないとき、iとjは関連を持たないという。他のどの行ベクトルとも関連を持たない行ベクトルを、孤立した行ベクトルという。

(例)
図4のI/Oマトリックスにおいては、ES行およびER行の2つが孤立している。

孤立した行ベクトルが同定されたとき、コンテクストダイアグラム作成のバグである可能性もある。バグではないとすれば、次にそれを孤立ではなくすという操作を行わなければならない。この操作のことをgluingという。Gluingとは、I/Oマトリックスで陽に表現されていない内部データを見出し、それと既存のデータとの関連を明確に定義するという操作にほかならない。Gluingは、孤立行ベクトルを無視して通常のプロセス分解を行い、その結果に応じてI/Oマトリックスを修正するという手順に従う。I/Oマトリックスが修正されると、ふたたび分解が行われ、設計者による検証を受ける。図7は図3にgluingを適用した結果のI/Oマトリックスを示す。

なお、ここで述べたgluingだけでは内部データを十分に洗いだせない場合もある。そのような場合

については、次章で議論する分解過程の検証結果を用いることができる。

```
Procedure Gluing
begin
  if I/Oマトリックスにおける孤立行ベクトルが存在しない then return
  else begin
       孤立行ベクトルを無視したI/Oマトリックスをもとにプロセス分解
       分解結果により、内部データを導入
       内部データを考慮し、I/Oマトリックスを修正
  end
end.
```

図6. *Gluing*の手順

2.2. Graph Interpretation

プロセス分解代数により自動的な分解が終了すると、次にその分解結果(代数のセンテンス)から*DFD*への変換が行われる。これは、本質的には*Adler*の方式に従うが、内部データを扱うための拡張が必要になる。おおまかな手順を示しておく。詳細は*[Adler88]*を参考にされたい。

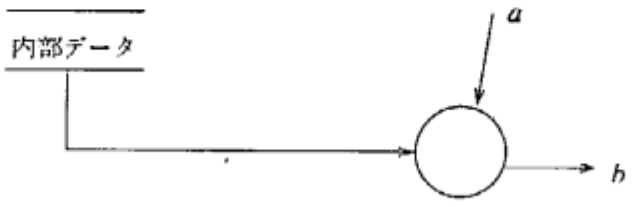
Step1. センテンスからすべての*trasnsforms*をリストアップする
*(*これらの*transforms*から*DFD*プロセスが作られる*)*
*Step2. step1.*で得られた*transform*のうち、タームはそのままプロセスとする。データ記号*a*に対しては、*(a → null)*なるプロセスを対応付ける.
Step3. プロセス間のデータフローを同定する

		Lift ボタンランプ*(LB)*	フロアボタンランプ*(FL)*	*emerge* メッセージ*(EM)*	*move (MS)*	*Lift*状態 *(LS)*
Lift No.	*(LN)*	*			*	
Lift ボタン *No.*	*(BN)*	*			*	
フロア *No.*	*(FN)*		*		*	
フロア ボタン種別	*(FB)*		*		*	
emerge シグナル	*(ES)*			*		*
emerge 解除	*(ER)*					*
*Lift*状態	*(LS)*	*	*			

図7. 中間データの導入とあらたな*I/O*マトリックス

内部データの導入によって、*step2*および*3*を拡張する必要が生じる。詳細は別の機会にゆずって、簡単な例を示しておく。

*(*例*) (*内部データ → *(a → b))*なるセンテンスには、次の*DFD*が対応する。



3. 分解過程の検証

*DFD*の各プロセスにおいては、そこに現れるデータおよびプロセスとデータが同一のレベルの抽象度を持つことが望ましい。例えば、図8におけるコンテクストダイアグラムを考えよう。この場合、出力データ*move*シグナル*1*、*move*シグナル*2*、*move*シグナル*3*というデータは、もうひとつ上の抽象度を持つ*move*と記述すべきである。なぜなら、この時点においては、*move*シグナルがどの入力データと関連して発せられるかということが重要であり、*move*シグナルの具体的な内容に立入る必要はないからである。このような抽象度のアンバランスが直ちにバグに結び付くわけではないが、可読性の低下にむすびつく可能性がある。これまでには、このデータ抽象度のバランスということに関する議論はほとんどなされてこなかったといってよい。ここでは、データの抽象度を定量的に議論するための尺度を導入し、それを用いた検証について論じる。

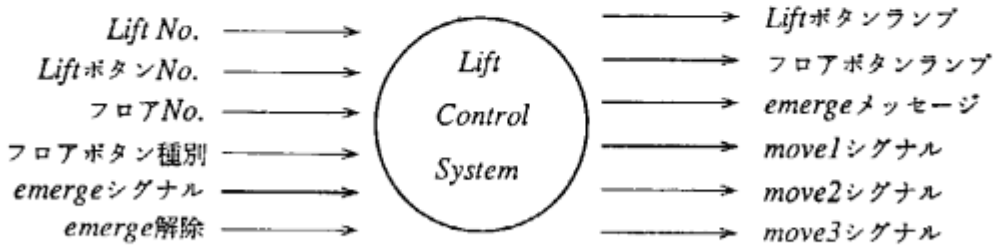


図8. データ抽象度にアンバランスのあるコンテクストダイアグラム

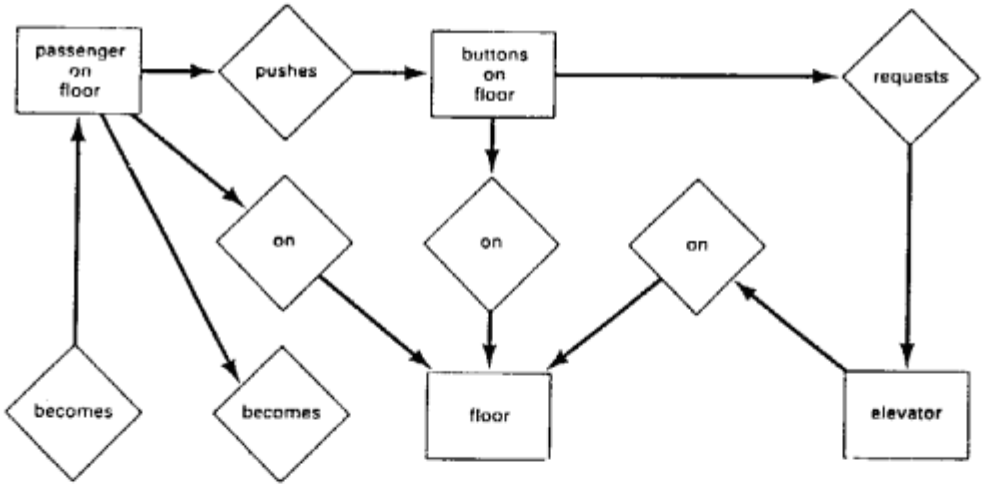


図9. *ER*ダイアグラムの例(*[Davis90, p.66]*の一部)

3.1 プロセス、データと*entity*の階層

ソフトウェアが対象とする実体*(entity)*間における関係を示すための方法として、*ER*ダイアグラムが良く用いられる。これは、*entities*がどのような*(*機能的*)*関係により相互に関連するかを図的に示したものである。例えば、リフトの問題の場合においては図9のようになる。この図において、ボックスは*entity*を表し、ダイアモンドは*entity*を関係付ける*relation*を表す。

*ER*ダイアグラムにおいては、図中の*relation*と*DFD*におけるプロセスとの関連が示されないという問題がある。そこで、*ER*ダイアグラムの概念を*DFD*のプロセスに反映させた、*process-entity*ダイアグラム*(PE*ダイアグラム*)*を約束する。

定義*(PE*ダイアグラム*)*

PEダイアグラムとは、有向グラフ(V,E)である。ここに、

$V: V_p \cup V_e$

V_p: DFD中のプロセスの集合

V_e: eneitiesの集合

$(p,q) \in E \Leftrightarrow p \in V_p, q \in V_p$であり、$q$は$p$を分解して得られたプロセス、または

$p \in V_p, q \in V_e$であり、qはpが実現するentity

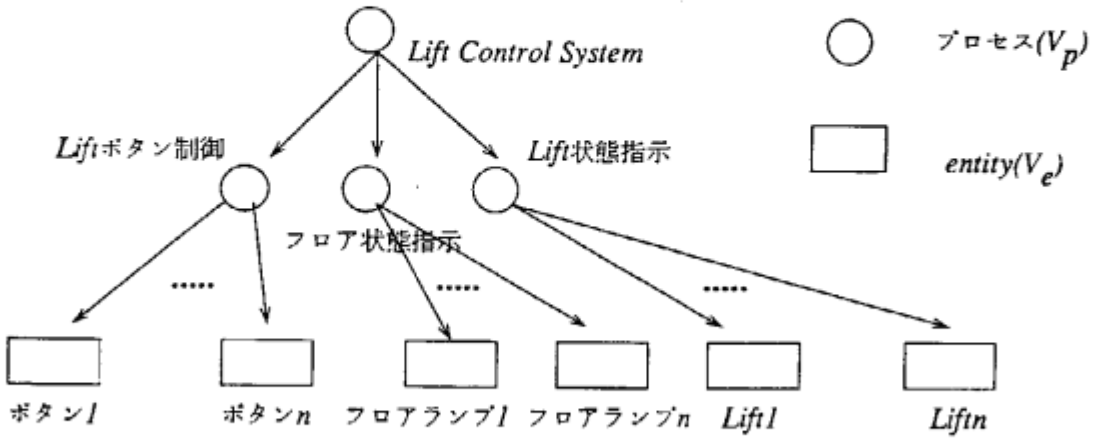


図9. PEダイアグラムの例

PEダイアグラムのノードnに対して、nから到達できるリーフ(entities)の集合を、プロセスnが関与するentitiesの集合という。直観的には、プロセスnにおいて、対象とされるentitiesの集合を示している。さて、DFDにおけるデータに関しても同様なダイアグラムを構成する。

定義(DEダイアグラム)

DEダイアグラムとは、有向グラフ(V,E)である。ここに、

$V: V_d \cup V_e$

V_d: DFD中のデータの集合

V_e: eneitiesの集合

$(p,q) \in E \Leftrightarrow p \in V_d, q \in V_d$であり、$q$は$p$を分解して得られたデータ、または

$p \in V_d, q \in V_e$であり、qはpに関連するentity

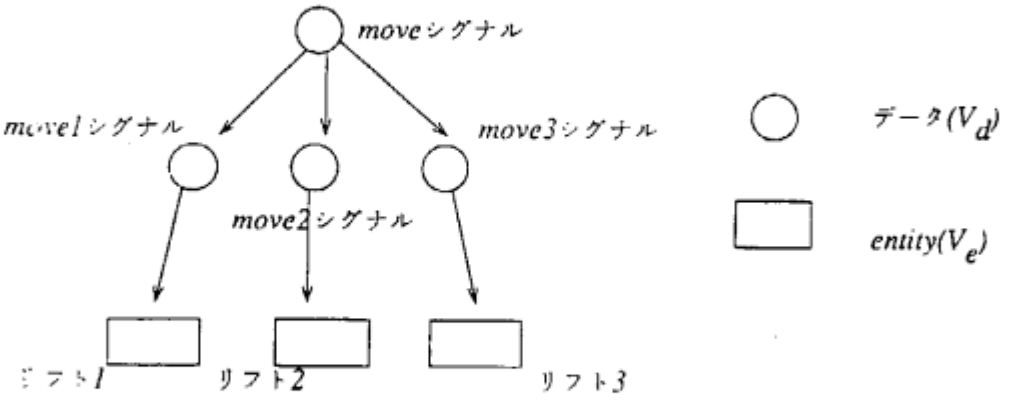


図10. DEダイアグラムの例

DEダイアグラムにより、データが関与するentitiesが明確に指示されることになる。先と同様にして、データdが関与するentitiesの集合を約束する。図9、図10にPEダイアグラムとDEダイアグラムの例をそれぞれ示した。

定義
ERダイアグラムにおいて、entity Aからentity Bにいたるパスが存在するとき、AとBとはcompatibleであるという。

定義
データiとデータjに関連するentiryの集合をそれぞれEi, Ejとする。また、Ei,Ejの各要素とcompatibleなentitiesの集合をcom(Ei), com(Ej)とする。このとき、

case1: $com(Ei) \cap com(Ej) = \phi$ であればデータiとjには関連がない
case2: $com(Ei) = com(Ej)$ であれば、EiとEjは同一のレベルである
case3: $com(Ei) \subset com(Ej)$であれば、EjはEiより上位レベルである

のように約束する。

(例)
データmoveとmove1について、$com(E_{move}) = \{Lift_1,...,Lift_n\}$, $com(E_{move1}) = \{Lift_1\}$とすると、データmoveはmove1より上位である。

定義
データiとデータjに対して、iとjが同一のレベルかまたはiがjの上位レベルであり、DEダイアグラムにおけるiのどの子ノード(データ)もjより下位のレベルであるとき、iはjに対して極大抽象度のデータであるという。

定義
プロセスpの入力データの集合をI, 出力データの集合をOとする。任意のデータ$i \in I$について、iに対する極大抽象度を持つデータ$j \in O$が存在するとき、このプロセスは入出力データ抽象度のバランスが保たれているという。

本章では、主としてデータの抽象度について議論してきたが、同様の方法をプロセスに対しても適用できる。このような方法を用いるとき、ユーザは最も下位のレベル(データ、プロセス分解が終了したとき)においてのみ、データあるいはプロセスとentityとの対応付けを行えば良く、さほどの負担は生じないはずである。なお、ここで用いたように中間的な概念(分解途中のデータ、プロセス)をentityと対応付けるという考えは、AIにおける故障診断[Hamscher90]などで用いられている。

3.2 I/Oマトリックスの検証

Adler[Adler88]では、プロセス分解のqualityに関する検証方式が提案されている。すなわち、I/Oマトリックスからプロセス分解代数による分解を行いgraph interpretationにより対応するDFDを得た後、逆にDFDからmatrix interpretationとよばれる方式により、I/Oマトリックスを逆生成する。それともとのI/Oマトリックスとを比較することが、分解の検証である。しかしながら、この方法では、階層的に分解を繰り返した場合に対応することができないという問題点がある。そこで、本節においては、下位のプロセスに対するI/Oマトリックスを上位のプロセスのI/Oマトリックスと比較することによる検証方法について議論する。

いま、プロセスpがn個のサブプロセスp1,...,pnに分解されたとする。このとき、各Piがさらにデータ分解されたものとする。このような場合に、分解された各プロセスに対するI/Oマトリックスを作

成し、それらをデータディクショナリにしたがって合成する。これと分解される前のプロセスに対するI/Oマトリックスを比較することで、分解に関する検証あるいは内部データの見い出しを行うことができる。図11に簡単な場合の例を示す。

5. 関連研究

本章では、他の関連する研究との比較を行う。 *Reisig[Reisig86]*は、ペトリネットを記述言語とするときの、段階的なソフトウェア生成について論じている。そこでは、オブジェクト指向と類似の観点により、対象システムをいくつかの*agents*の集まりとしてとらえる。そうして、各*agent*に対する設計を済ませた後それらを*1*つにまとめあげるという操作を行なう。*Reisig*においては、このような設計過程が満たすべき条件について議論されている。本稿で提案した方法が*top-down*的な立場に立つのに対して、*Reisig*の方法は*bottom-up*的な要素が強い。

*Kramer[Kramer90]*では、段階的なソフトウェア生成を目的として、ペトリネットを拡張した仕様記述言語が提案されている。しかしながら、その記述言語を利用する際の方法論についてはほとんど考察されていない。

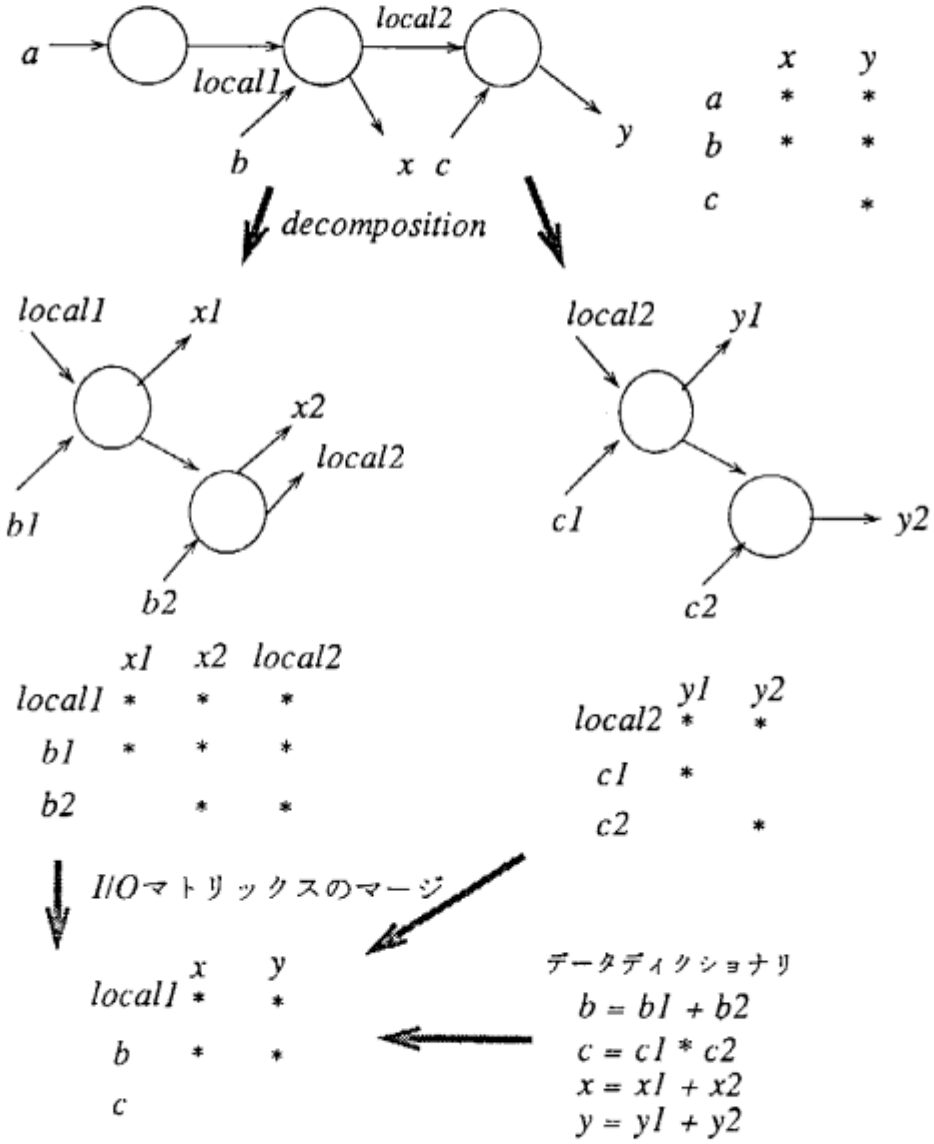


図11. I/Oマトリックスの検証

おわりに

ソフトウェアを段階的に設計する方法について議論した。本稿では、*DFD*の作成に関する部分のみを取り扱い、実際のコード化(ミニ仕様書の作成)については論じなかった。このことについては、機を改めて報告する予定である。

対象とするシステムにリアルタイム性やリアクティブ性が含まれるような場合においては、*DFD*からコーディングまでの過程に相当の困難がある。本稿で述べた方式を、リアルタイム*SA*のような方式に対して拡張することも考えられる。あるいは、*Kramer[Kramer90]*らのように、*high-level*ペトリネットを用いて、データフローと同時にデータの同期を設計していくことも考えられる。そうした拡張について、本方式を適用することは今後の課題である。

また、並列プログラムを対象とする場合、たんなる*I/O*マトリックスでは不十分なことがある。データ間の同期を表現しうるような拡張が望まれる。

謝辞

本研究の一部は、第5世代コンピュータプロジェクトの一環として行われた。研究の機会を与えて下さった方々に謝意を表します。

参考文献

[Adler88] M.Adler: An Algebra for Data Flow Diagram Process Decomposition, IEEE trans. on SE, Vol. 14, No. 2, 1988.

[DeMarco78] T. DeMarco: Structured Analysis and System Specification, New York Yourdon, 1978.

[Reisig86] W. Reisig: Petri Nets in Software Engineering, Petri Nets: Applications and Relationships to Other Models of Concurrency, 1986.

[Kramer90]B. Kramer et al: Petri Net Based Models of Software Enginerring Processes, HICS-23, 1990.

[Davis90] A. Davis: Software Requirements :Analysis and Specification, Prentice Hall 1990.

[Hamscher90] W.Hamscher: XDE: Diagnosing Devices with Heuristic Structure and Known Component Failure Modes. CAIA 1990.

[Gane79] C. Gane et al: Structured System Analysis: Tools and Techniques, Prectice-Hall, 1979.

[Ross76] D.Ross et al : An Approach to Structured Analysis, Comput. Decisions, Vol.8, No.9, 1976.