佳作論文

統一的設計方法論に基づくソフトウェア設計システム†

紫 合 治†† 岩 元 莞 二†† 藤 林 信 也††

ソフトウェア開発の成功の可否は，設計をいかにうまく行うかどうかに大きく依存する．本論文では，モジュールの呼び出し関係を中心としたモジュール分割法とデータフローを中心としたモジュール分割法を融合した統一的設計法を導入し，この設計法に基づいたソフトウェア設計を効率化する設計システムについて述べる．

この新しい設計法では，システムを複数要素の結合として表現し，各要素をさらに下位の複数要素を結合したシステムとして設計する．この詳細化をプログラムモジュールの切り出しとその仕様定義に至るまで段階的に進める．分割法に一定の規則を設定し，ある段階で呼び出し関係で結合された要素が次の段階ではデータフロー結合のシステムとして設計でき，またその逆も可能となるようにしている．これにより，それぞれの場面に最も適した分割法が選択でき，広い分野にわたり適用可能な方法論となっている．

設計システムはこの方法論に沿った設計を表現する設計言語とそのプロセッサからなる，設計言語は主としてモジュール間関係を形式的に表現するものである．プロセッサは，この言語で書かれた設計情報を設計データベースに登録し，一貫性の解析を行い，各種設計文書を自動生成する．さらに，この方法論で設計されたシステムを既存のプログラミング言語で実現するための簡単な仕掛けが用意されている．

1. まえがき

ソフトウェアが複雑化，大型化するに従って，高い信頼性や保守性をいかにして達成するかが大きな問題になってくる．大規模ソフトウェアでは，その設計の良否がシステムの信頼性，保守性を大きく左右する．一般に，ソフトウェアシステムの設計は，

① 全体をより単純な要素の単純な組み合せに段階的に分割する作業（モジュール化設計）

② 最終的に得られた要素に対応するプログラムモジュールを設計する作業（モジュール内設計）

に分けられる．最終的な要素が十分に単純で小さく，それぞれ独立した機能をもっていれば，②の作業は多くの小さなプログラムの設計になる．すなわち，大規模システム特有の複雑さは①のモジュール化設計で扱われる．このため，この複雑さを克服するためモジュール化設計方法論の研究が極めて重要である．

この論文では，モジュール化を決定する重要な要因であるモジュール間結合方式として，制御フロー（呼び出し関係）による結合とデータフロー（データの受け渡し関係）による結合のそれぞれの利点を生かすため，両者を融合した方式を導入し，これに基づいた新しいモジュール化設計法を提案する．さらに，この設計法に従った統一的設計表現を与える設計言語と，それによって表現された設計情報を解析するプロセッサについて述べる．

従来，モジュール結合機構として呼び出し関係による制御フローが広く使われてきたが，そこでのモジュールの実行順序は，それらを呼び出す側のプログラムで完全に定める必要があった．一方，多くの場合，システムの機能は入力データから出力データへの段階的変換として表現できるが，この場合各データ変換処理の実行順序は，「あるデータを生成する処理は，そのデータを使用する処理に先行する」という規定だけで定まる．したがって実行の順序は部分的な順序関係だけを満たせばよく，いくつかのプロセスがデータを受け渡ししながら並行に動作するモデル（データフローモデル）がより適切である．制御フローによるモジュール化では，これを無理に完全な順序になおす必要がある．その場合，実行の順序を表わす制御フローを動的に制御するための変数が随所に挿入された，複雑で変更に対して融通性のないプログラムになることが多い．ここでは，従来の制御フローによるモジュール化では克服できなかったこれらの複雑さを取り除くために，データフローによるモジュール化を導入する．

一方，システムをデータフローだけで実現しようとする試みが，Dennis[1]のデータフロー機械等最近の新しい計算機アーキテクチャの研究に見られるが，この場合，本来のデータ以外にデータフローを動的に制御するためのデータが必要になり，同じ機能を表現した通常の順次処理プログラムよりもかえって複雑になることがある．理解性や融通性の観点から，データフロ

† A Software Design System Based on a Unified Design Methodology by OSAMU SHIGO, KANJI IWAMOTO and SHINYA FUJIBAYASHI (Central Research Laboratories, Nippon Electric Co., Ltd.).

†† 日本電気(株)中央研究所

ーによるモジュール化だけをとり上げるのは片手落ちで現実的ではない.

一般に，システムの機能が，データ（主に順次列データ）の段階的変換として表現できるときはデータフロー機構が適し，実行順序に意味をもたせた処理の組み合せで表現できるときは制御フロー機構が適している．それぞれに適した分野や場面で両者を矛盾なく使う必要があり，無規律に混合するとかえって複雑なシステムになりやすい.

この論文で述べるモジュール化設計法は，機能階層に基づく要素分割法を使って，データフローと制御フローのそれぞれの利点を保ったまま，両方の機構を矛盾なく融合させたものである．従来の設計方法論は，それに適した場合では有効だが適していない場面ではほとんど無力であった（場合によっては有害ですらあった）ためその適用範囲が限られていた．ここで導入した方法論は，問題全体を機能的に独立した部分に分け，それぞれの部分に適切な設計法が選択できるため，システムプログラムから応用プログラムまで広範囲の適用が可能になる.

この設計法に従って設計を進めると，システム全体から段階的に分割して得られた要素の間の関係が定まるが，これらの情報はシステムの開発時だけでなく保守／改造時には特に有益に使われる．しかし，これらの設計情報を手書き文書として管理するのでは，設計変更にともなう文書の修正が困難なだけでなく，設計の妥当性の検証や変更の影響の追跡に際し，膨大な量の文書から特定の関連情報を引き出すために多大の工数を要することになる．これは，従来の多くの設計文書が，その作成に要した労力の割にはあまり活用されなかった理由の一つにあげられよう.

このため，手作業による文書管理に代り，設計データベースにより設計情報の維持管理を行い，設計の一貫性と完全性の解析支援および種々の文書の自動生成機能をもつシステムを開発した．要求定義や設計のための同様の目的をもつ支援システムとして ISDOS[2]，REVS[3]，国内では SSD[4] など多くの例がある．本システムは，上記の方法論に基づいた設計過程を想定して，設計方法論の実践を容易にするように設計言語とシステム機能が定められている．言語は，主としてこの方法論に基づく要素分割設計の結果を形式的に表現するもので，広範囲の適用を狙ってプログラミング言語独立になっている．言語プロセッサは，この言語による設計記述を設計データベースに登録し，データベース上で要素間にまたがる解析を行い，方法論の実践を支援する各種の設計文書を自動生成する．なお，このプロセッサは，ソフトウェア開発保守を統合的に支援するシステム SDMS[5] (Software Development and Maintenance System) の一サブシステムになっている.

以下，2章では設計方法論について述べ，3章，4章で設計言語とそのプロセッサについて述べ，5章で，この方法論によって設計されたシステムを，既存のプログラミング言語で実現するための簡単な道具について報告する.

2. 設計方法論

はじめに，データフローか制御フローの一方だけを用いたモジュール化設計法の問題を議論し，次にそれらを融合した新しい設計方法論を導入する.

2.1 データフロー対制御フロー

計算機が対象とする処理の多くは，入出力データとして順次列データ（順編成ファイル，プリンタ出力，回線データなど）を扱う．この場合，システムは入力から出力への段階的なデータ変換の組み合せによって表わせることが多い(**図1**)．このようなモジュール化は事務処理システムなどによく見られ，中間データは作業ファイル，各変換処理は実行プログラム，変換の組み合せはジョブ制御指示として実現され，変更の影響範囲が小さく理解し易いシステムになる．しかし，このようなシステムは，作業ファイル利用による効率低下のため，このままでは適用できないことがある．このようなとき，各処理単位を呼び出し関係で結合し，パラメータによってデータの受け渡しを行う方式が一般にとられている．その場合，データの順次構造と合わない制御構造になることが多い．例えば，**図2**のように，図1の P_3 を主ルーチン P_3' とし，P_1, P_2, P_4 は P_3' から呼ばれるルーチン P_1', P_2', P_4' にそれぞれ変換したとする．図で D_1', D_2', D_3' は，対応するルーチンの1回の呼び出しで受け渡されるデータを示す．ここで，D_1 を生成する立場で P_1 からみた D_1 の

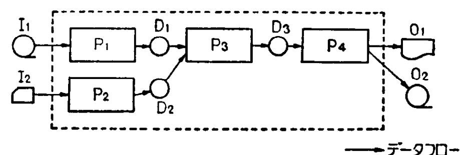


図 1 データフローによるモジュール分割
Fig. 1 Data flow modularization.

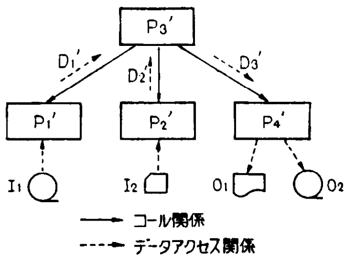
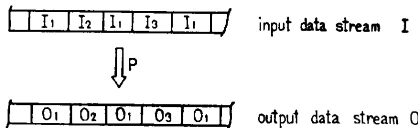
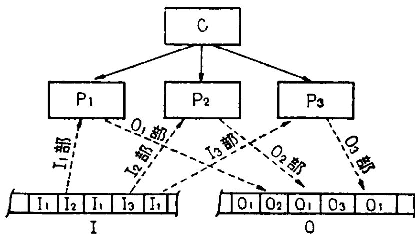


図 2 図1の制御フローへの変換例

Fig. 2 Transformation of figure 1 to control flow modularization.



(a) 入出力データ構造が対応する例
(I_i, O_j はデータの種類を示す)

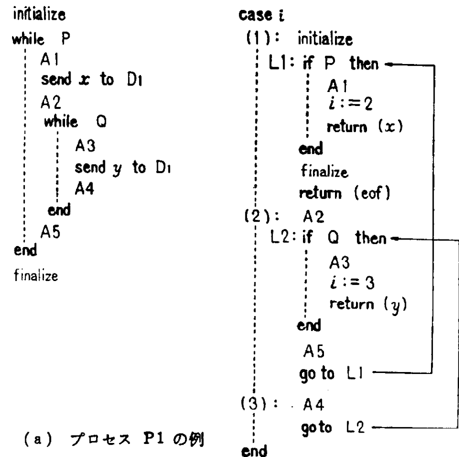


(b) (a)の制御フローによる実現

図 4 制御フローによるモジュール分割

Fig. 4 Control flow modularization.

順次構造は，一つの x レコードに可変個の y レコードが続いている論理的単位の列であるとする．この場合，図1の P_1 は，P_3 での D_1 の受け取り方に関係なく，**図3**(a)のようにこのデータの論理構造に合った制御構造にできる．一方，図2の P_3' が，この構造にかかわりなく P_1' の1回の呼び出しで一つの x または y レコードを受け取って処理したいとする．この場合，P_1' は複数回の呼び出しで一連のまとまった処理ができるように，呼び出されるたびに1レコード分の処理をそのときの状態によって選択する必要があり，図3(b)のように一連の処理が分散され D_1 の順次構造に合わない複雑な制御構造*になる[6]．



(a) プロセス P1 の例

(i は初期値が1の static 変数)
(b) P1 をルーチンに変換した P1′ の例

図 3 プロセスからルーチンへの変換例

Fig. 3 Process/routine transformation.

このような場合，データ変換処理と中間データをそれぞれプロセスとメッセージバッファ**で実現すると，理解し易い制御構造を保ったままファイル入出力での効率低下を除くことができる．図1の P_3 は，D_1 と D_2 のデータが全部出そろうまで実行を待つ必要はないので，D_1 と D_2 をメッセージバッファで実現しても実行結果は変らない．これにより，ファイルインタフェースによる簡潔なモジュール化の方法がシステムプログラムなどにも効率を落すことなく適用できる．このようなモジュール化をデータフローによるモジュール化と呼ぶ．

一方，**図4**(a)に示すように，入力と出力のデータ構造が対応し，かつ対応する部分間の変換がその部分の性質(種類)によって決まる場合，データの部分の種類を判定する主ルーチンを設けて，それぞれの種類に対応する変換処理ルーチンを呼び出す方式(図4(b))が適している***．これを**図5**のように，入力データを種類ごとに分割するプロセス C_1 と，各データの種類ごとの変換処理プロセス P_i，およびデータをまとめて出力するプロセス C_2 にデータフロー結合で実現すると，出力のデータ順序を入力のデータ順序に合わ

* Jackson[7] の設計法によると，入出力データの構造の対応がとれない場合を構造不一致 (structure clash) と呼び，この解決のために上の D_1 のような中間データが導入される．また，P_1 を P_1' に変換することをプログラムインバージョンと呼んでいる．この変換は，プログラム単位が大きく，複数のルーチンで実現されている場合はかなり複雑になる (文献 7) の7章～9章参照)．

** 並列に動作する二つのプロセス間の情報伝達に使われ，メッセージの FIFO (first-in first-out) キューで構成される．プロセスがそれにメッセージを書き込む (読み出す) 場合，キューがいっぱい (空) ならば，そうでなくなるまで待たされる機構をもつ[8]．

*** Jackson[7] の設計法によると，入出力データの構造が合致した場合に相当する (文献 7) の4章参照)．

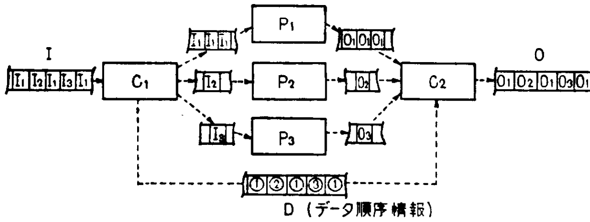
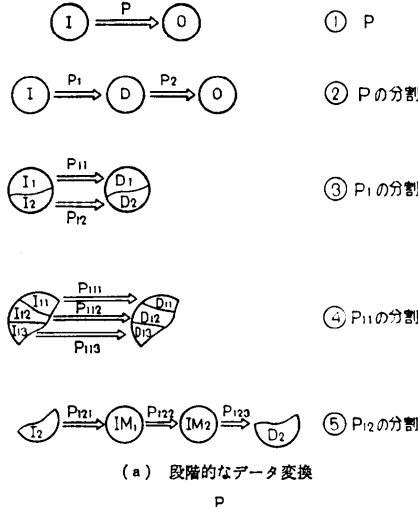
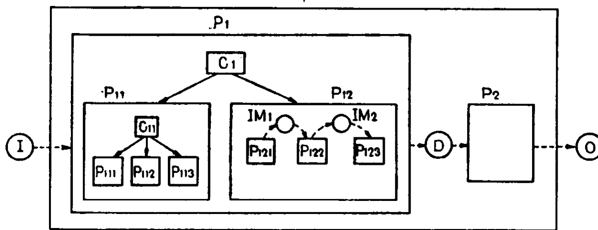


図 5 図4をデータフローに変換した例

Fig. 5 Transformation of figure 4 to data flow modularization.



（a） 段階的なデータ変換



（b） （a）に基づいたシステム構成

図 6 データフローと制御フローの融合例

Fig. 6 Integration of data flow and control flow.

せるために，C_1 から C_2 へデータの順序に関する制御データDを渡さなければならない．図4（b）では C_1 と C_2 をまとめたものが一つの主ルーチンに対応するので，このような要素間にまたがる制御データは必要ない．

一般に，通常の順次処理プログラムでは実行の順序を制御するための制御変数を多用すると，プログラムが複雑で変更が困難になるが，データフローによるモジュール化でも処理対象データ以外の制御のためのデータが入ると同様の問題が生じる．Dennis のデータフロープログラム[1]ではこの制御のためのデータやゲートが多用されて理解しにくくなっていることが多い．データフローが適切な場合に制御フローを使ったり，その逆をすると複雑になるのは，それによって余分な制御データが必要になるためである．

2.2 データフローと制御フローの融合

余分な制御データをできるだけ導入しないで，それぞれの場面でそれぞれに適したモジュール化を使えば，理解し易く変更し易いシステム構造が得られ，信頼性，保守性の向上が図れる．このため，データフローと制御フローを無理なく融合した設計法を導入する．

例えば，**図6**（a）のように，入力Iから出力Oへの変換P(①)を段階的に詳細化していく場合を考える．同図で，②は中間データDを導入したPの詳細化(データフロー分割)，③はIとDの部分構造に従った P_1 の詳細化（制御フロー分割），④はさらに詳細なデータ構造による P_{11} の詳細化（制御フロー分割），⑤は中間データの導入による P_{12} の詳細化（データフロー分割）を示している．以下，同様の詳細化が進められる．この場合，システムPの構成は図6（b）のように階層的に表わせる．図において，C_1, C_{11} はそれぞれ P_1, P_{11} の主ルーチンである．

これを一般化すると次のようなシステム分割法が得られる．

（1） システムの主な入出力データに着目し，それらのデータ構造の枠組を定義する．

（2） （1）で定めた入力データと出力データの構造を調べて，

ⓐ対応があれば，その対応に合わせて制御フローによる要素分割を行い，

ⓑ明確な対応がなければ，入力から出力へ徐々に対応をつけるための中間データを導入して，それらを介したデータフローによる要素分割を行う．

（3） （2）の分割で生じたそれぞれの要素について，（1），（2）の操作を繰り返す．これを，データ構造が詳細に定まり，要素の機能が十分小さく，簡単なプログラムで実現できる見通しが立つまで繰返す．

こうして得られる要素の階層は，機能的なまとまりの包含関係を表わしている．上記の（2）で，ⓐの制御フロー分割で得られる要素を**ルーチン**，ⓑのデータフ

ロー分割で得られる要素を**プロセス**と呼ぶ．ここで，ある要素の実現で制御フロー分割とデータフロー分割のどちらを選択するかは，その要素の入出力データ構造のとらえ方に依存し，それがルーチンかプロセスかというような，その使用環境には依存しない．したがって，図6(b)のP_{12}のように，制御フロー要素のルーチンがデータフロー分割で実現されることがあり，従来提案されていたデータフロー機構であるPORT[9]やDSLM[10]のように，局所的なデータフロー結合が使えない機構では不十分である．

一方，順次列データ以外のテーブルやスタックなどが必要になる場合は，上記のルーチンやプロセスの他に，情報隠蔽[11]やデータ抽象化のためのモジュール要素を使う．このため，抽象機械や抽象データに対応して複数のオペレーションをまとめ，それらの実現で現れるルーチンやデータを外から操作できないようにした**グループ**[12]（制御フロー用）と**モニタ**[8]（データフロー用）要素を導入する．これにより，制御フローやデータフローによるモジュール化方法論と，データ抽象化の概念を使ったモジュール化方法論のそれぞれが，互いの利点を保ったまま共存できる．

2.3 データフローと制御フローの融合に伴う規則

前述した要素の機能的階層に基づくデータフローと制御フローの融合では，要素分割の各段階ではどちらか一方しか使えないように制限しているので，それぞれの方法が混乱することなく，理解し易さを保ったまま適用できる．

さらに，データフローの適用を理解し易くするため，メッセージバッファの使用に次の制限を設ける．

① 各メッセージバッファに対し，データを送るプロセス (producer) と受け取るプロセス (consumer) はそれぞれ一つとする．

② 各プロセスは，メッセージバッファが空またはいっぱいかどうかを直接知る手段を持たない．すなわち，メッセージバッファに対しては送り出し (send) か受け取り (receive) のオペレーションしかなく，空で受け取り（いっぱいで送り出し）を実行すると常に待たされる．

この二つの制限によって，システムの機能が各プロセスのスケジュールのされ方によらず確定的に定まることが保証される[13]．また，要素分割に関し，次の規則を設ける．

① データフロー結合の要素Cがデータフローで分割された場合，Cが使用していた各メッセージバッファはそれぞれCの一つの内部要素（内部プロセス）でのみ使用できる．

② データフロー結合の要素Cが制御フローで分割された場合，Cが使用していたメッセージバッファはCの任意の内部要素で使用できる．

ただし，①,②とも，その使い方はCでの規定に従う（Cが送り出しをするメッセージバッファは，内部でも送り出ししかできない）．

③ 制御フロー結合の要素Cがデータフローで分割された場合，Cが呼び出していた各ルーチンやオペレーションはそれぞれCの一つの内部要素でのみ呼び出せる．

④ 制御フロー結合の要素Cが制御フローで分割された場合，Cが呼び出していたルーチンやオペレーションはCの任意の内部要素から呼び出せる．

上記の①と③は，システムの機能が確定的に定まることを保証するための制限である．ただし，実際に同時動作がなく，かつ呼び出されるルーチンが内部状態を保持しない関数的な場合は，③の制限を除くことができる．②のように，通常の順次処理ルーチンからメッセージバッファを使用する場合は，従来の順編成ファイルと同様に扱うことができ，通常の順次処理プログラムに慣れている設計者やプログラマにもメッセージバッファが簡単に使える．

3. 設計言語

設計言語は，設計の結果を統一的に記述するためだけでなく，設計の考え方そのものを形成する指針になり，設計標準化のための重要な役割をもつ．ここで述べる設計言語 SDL (System Design Language) は，前章の設計方法論に基づく要素分割を主に，さまざまの設計情報を設計データベースへ登録するための言語である．読み易い設計文書や各種の関連をもった設計情報は，プロセッサによって自動生成することができるので，SDL ではむしろ書き易さを重視し，重複した記述をさけるようにしている．以下に SDL の特徴的な機能について述べる．

3.1 コンポーネント

2章の方法論に従うと，設計は要素の段階的な分割を中心に進められる．SDL では，この要素に対応するものを**コンポーネント**と呼び，SDL 記述の単位になる．コンポーネントは機能的なまとまりを持つシステムの部分を表わすもので，システム全体から徐々に最終的なコンポーネントへ分割される．コンポーネン

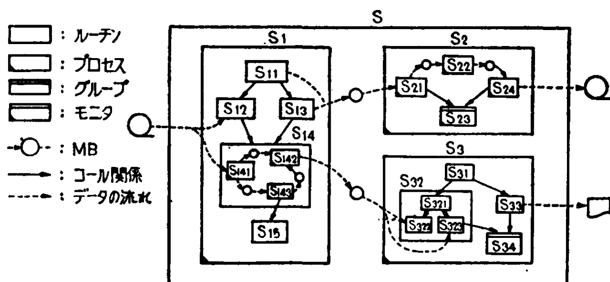


図 7 SDL によるコンポーネント階層の例
Fig. 7 Illustration of components hierarchy in SDL

トの階層において，あるコンポーネントがその実現のためにさらにいくつかのコンポーネントに分割されるとき，そのコンポーネントを**ノンプリミティブ**と呼ぶ．分割によってコンポーネントの親子関係が定まり，それ以上分割されないコンポーネント，すなわち子を持たないコンポーネントを**プリミティブ**と呼ぶ．これは一般にプログラムモジュールに対応する．プログラミング段階ではノンプリミティブコンポーネントは記述の対象にはならないが（強いていえばリンケージ指示に対応），設計ではむしろノンプリミティブコンポーネントの記述を重視する．ノンプリミティブコンポーネントは，最終的なプログラムモジュールを機能に従って階層的に分類した結果を表わしているともいえる．

コンポーネントには，2章で述べた**ルーチン**（通常のサブルーチンの機能をもつ），**プロセス**（並列処理の単位になる），**グループ**（抽象データ，抽象機械の表現），**モニタ**（プロセス間共有抽象データの表現）の他に，どのコンポーネントにも含まれない最上位のコンポーネントである**ルート**と，メモリ上の具体的なデータ領域，ファイル，データベースなどを表現する**データ**がある．これらの種類はコンポーネント自身の持つ性質を表わし，その実現の仕方を示す，プリミティブ，ノンプリミティブの区別とは独立である．

図 7 に SDL によるコンポーネント階層の例を示す．ここで，S, S_1, S_2, S_3, S_{11}, S_{12} などのコンポーネントがそれぞれ SDL の記述単位になる．

3.2 コンポーネントの記述

SDL 記述は，$ で始まる文頭語を持った文の集まりよりなる．各コンポーネントは，その種類を示す文頭語（$PROCESS, $ROUTINE など）を持ったヘッダ文で始まり，$END 文で終る．ヘッダ文には，そのコンポーネントの管理上の名前である**ユニット名**（システム全体で一意）と，その機能を表わす**コンポーネント名**（親を同じくする子の間で一意）が指定される．このような機能名と管理上の名前の分離は，多くの設計者によって開発される大型システムでは必須になる．

コンポーネントの記述は，そのコンポーネントを使用する側にとって必要な機能概要，パラメータ仕様，入出力条件記述などからなる**外部仕様部**と，そのコンポーネントがどのような子コンポーネントのどのような関係によって実現されるかを示した**内部仕様部**に分けられる．**図 8** にプロセスコンポーネント記述の枠組を示す．また**表 1** に SDL 記述文一覧を外部仕様部と内部仕様部に分類して示す．表1で星印（*）を付けた文は，本体

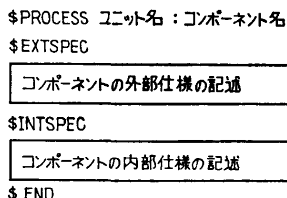


図 8 プロセスコンポーネント記述の枠組
Fig. 8 Skeleton of process component description.

表 1 SDL による文の一覧
Table 1 SDL statements list.

区分	文頭語	内容
外部仕様部	$ FUNCTION*	機能概要の説明
	$ KEY	キーワード，キーフレーズの記述
	$ PARM	パラメータ仕様（入出力の別，属性，意味等）
	$ OPER	オペレーション仕様（O/V/OV の別，パラメータ，機能，効果等）
	$ SYSPARM	システム生成時に定められるパラメータ（テーブルサイズ等）の宣言
	$ SYSENV	システム環境（OS，ハードウェアとのインタフェース等）の宣言
	$ COND*	入出力条件の記述
	$ EXAMPLE*	使用例，動作例等の記述
	$ PERFORMANCE*	効率仕様の記述
内部仕様部	$ CONV*	慣例規則（名前付け，メッセージの形等）の定義
	$ SYNONYM	同義語の定義
	$ TYPE	データの属性（type）の定義．Pascal とほぼ同じ
	$ VAR	コンポーネント間で共通に使う変数の宣言
	$ MB	メッセージバッファの宣言
	$ FILE	ファイルの宣言
	$ ICOMP	内部（子）コンポーネントの宣言
	$ CONNECT	内部コンポーネント間関係の定義
	$ ASSERT*	内部コンポーネント間で一般に成立すべき条件の記述
	$ ALGORITHM*	全般的な処理方式の記述

（* は，内容の記述が非形式的な叙述からなる文を示す）

```
54-10-17   16:27:53        S D M S - 6  (V : DE1.0)           PID: S1970M              ページ  1
    1      $ROUTINE
    2         DEUS2      : UNIT-SUMMARY-DATA-GEN
    3      $EXTSPEC
    4      $FUN
    5         /* FIND DESIGN DATA BASE AND GENERATE UNIT SUMMARY DATA
    6         INTO TOKEN FILE. THIS COMPONENT IS EXECUTED IN BATCH-ENV. */
    7      $KEY
    8         "DEMO", "SDL", "UNIT SUMMARY", "USUM", "BATCH", "DATA BASE"
    9      $COND
   10         IN /* PARM-FILE AND UNIT-NAME-FILE MUST BE ATTACHED */;
   11         OUT /* MESSAGE-FILE AND TOKEN-FILE IS GENERATED */;
   12         BRK /* IF UNIT IS NOT FOUND, NO TOKENS ARE GENERATED */
   13      $EXAMPLE
   14        /*     IF UNIT-NAME-FILE=UN1,UN2,UN3 AND PARM=NOFUN,
   15               TOKEN-FILE=
   16         <UN1-BEGIN><ROUT><UN1><CN1>...<FUNB><COND><PERF><END>...<END>
   17         <UN2-BEGIN><GROU><UN2><CN2>...<FUNB><COND><PERF><END>...<END>
   18         <UN3-BEGIN><ROUT><UN3><CN3>...<FUNB><COND><PERF><END>...<END>
   19                  <EOF>
   20               WHERE UN1 AND UN3 ARE ROUTINE, AND UN2 IS GROUP      */
   21      $PERFORMANCE
   22         MEMORY /* LESS THAN 64K WORDS */;
   23         TIME /* LESS THAN 30 SEC/UNIT */
   24      $INTSPEC
   25      $ICOMP
   26         ROUTINE DEUS21   UNIT-SUMMARY-BATCH-MAIN, DEUS22   UNIT-HEAD-AND-ID-INF-GEN,
   27           DEUS23   UNIT-FUNCTION-GEN, DEUS24   UNIT-INT-STRUCTURE-GEN, DEUS25
   28           CALL-STRUCTURE-GEN
   29      $MAIN
   30         DEUS21
   31      $CONNECT
   32         DEUS21    CALL DEUS22;
   33         DEUS21    CALL DEUS23 IF /* FUN=YES */;
   34         DEUS21    CALL DEUS24 IF /* INT=YES */;
   35         DEUS21    CALL DEUS25 IF /* CALL=YES */;
   36         DEUS22    CALL DEDB1 / V;
   37         DEUS22    CALL DEDB3 / V;
   38         DEUS22    CALL DEUSTKNS / O;
   39         DEUS23    CALL DEDB1 / V;
   40         DEUS23    CALL DEUSTKNS / O;
   41         DEUS24    CALL DEDB1 / V;
   42         DEUS24    CALL DEDB2 / V;
   43         DEUS24    CALL DEUSTKNS / O;
   44         DEUS25    CALL DEDB1 / V;
   45         DEUS25    CALL DEDB2 / V;
   46         DEUS25    CALL DEUSTKNS / O;
   47         DEUS21    CALL DEUSTKNS / O;
   48         DEUS21    RD   PARM-FILE;
   49         DEUS21    RD   UNIT-NAME-FILE;
   50         DEUS21    WT   MESSAGE-FILE
   51      $ALGORITH
   52        /*     OPEN FILES; INPUT PARM-FILE;
   53               WHILE: UNIT-NAME FILE NOT EOF
   54                 INPUT UNIT NAME FILE( UN );
   55                 GENERATE UNIT SUMMARY DATA FOR UN
   56               END:
   57               CLOSE FILES           */
   58      $END
```

図 9 SDL によるコンポーネント記述例（プロセッサの出力コマンドによる）

Fig. 9 Illustration of SDL component (listed by SDL processor output command).

が叙述もしくは文の種類を示す予約語, 例えば, 入出力条件の IN (入力), OUT (出力), ABT (アボート時) など, を伴った叙述からなる説明文で構成される. このように, 叙述による非形式的な記述もその内容に従って分類して書かれるので, SDL が設計で記述すべき事項を示すチェックリストになるとともに, 例えば効率仕様一覧など, 内容に従った設計文書の自動生成が可能になる. **図 9** に, SDL によるコンポーネント記述の例を示す.

3.3 コンポーネント間関係の記述

SDL の大きな特徴の一つは, $ CONNECT 文によるコンポーネント間関係の記述にある. 前述の方法論によると, あるコンポーネントの外部との関係は, そのコンポーネントが要素として認識された時点, すなわちその親コンポーネントの内部設計時(要素分割時)に定まる. このため, SDL ではあるコンポーネントCの外部との関係 (パラメータなどは除く) は, Cの親コンポーネントの内部仕様部に書かれる. 従来の設計書では, このようなCの外部との関連は, Cのインタフェース情報としてC自身に書かれていたが, SDL ではこのような設計書は, プロセッサによって親コンポーネントから情報を抽出して自動生成される.

コンポーネント間の関係として次の2種類がある.

① ルーチンやオペレーションを呼び出す関係.

② データやメッセージバッファを操作する関係.

これらの関係は, 「～が(主語)～を(目的語)～の場合 (条件)～する (動詞)」という内容に従って, 次の形式の文の並びで統一的に表現する.

主語 動詞 目的語 1 〔TO 目的語 2〕〔IF 条件〕

ここで〔 〕は省略可を示す. また文と文の間は ; で区切る. 主語は目的語のコンポーネントやデータに動詞で示された動作を行うコンポーネントを表わす. 動詞は動作の種類を示し, CALL, SND/RCV (メッセージバッファの送り出し／受け取り), SET/REF (データ値の代入／参照), RD/WT (ファイルの読込み／書出し), CPY (データのコピー) などが用意されている. CPY などの動詞の場合 "TO 目的語 2" が必要になる. CALL の目的語としてグループかモニタのオペレーションを表わす場合は, 呼び出されるオペレーションの集合を次のように指定することができる.

U 1 CALL G1/(OP 1, OP 2, OP 3)

これは, U1 がグループ (またはモニタ) G1 のオペレーション OP 1, OP 2, OP 3 を呼び出すことを表わす. 目的語がデータの場合, データのある部分要素だけを操作することを表わすために, ／をつけて修飾した記法を用いることができる.

U 1 REF D 1/PART 1

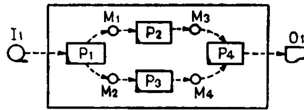
これは, U1 が D 1 の PART 1 部分を参照することを表わす. さらに, ファイルやメッセージバッファに対する動的な操作の仕方を表現するために, 操作するデータの内容の制限を付記できる. 例えば

U 1 WT F 1 {ページヘッダブ};

U 2 WT F 1 {ページホンタイブ}

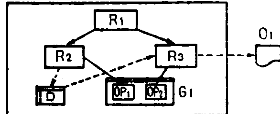
は, U1 がファイル F1 のページヘッダ部を, U2 がページ本体部を出力することを表わす. IF部分は, 動作の条件を記述し, 例えば次のように用いる.

U 1 CALL U 99 IF {DATABASE OVERFLOW OCCURS}




```
$ CONNECT
  P1 RD I1 ; P4 WT O1 ;
  P1 SND M1 , M2 ;
  P2 RCV M1 ; P2 SND M3 ;
  P3 RCV M2 ; P3 SND M4 ;
  P4 RCV M3 , M4
```

（a） データフロー分割の $CONNECT 記述例




```
$ CONNECT
  R1 CALL R2 IF {-----} ;
  R1 CALL R3 IF {-----} ;
  R2 CALL G1/OP1 ;
  R3 CALL G1/OP2 ;
  R2 SET D ; R3 REF D ;
  R3 WT O1
```

（b） 制御フロー分割の $CONNECT 記述例

図 10　コンポーネント間関係の記述例

Fig. 10　Illustration of inter-component relationship.

図 10 にコンポーネント間関係の記述例を示す.

一般に，コンポーネント間関係はシステムの静的な構成を表現しているが，システムを完全に理解するには，動的な構成に関する情報も必要になる．例えば，あるコンポーネントがいくつかのルーチンを呼び出し，データを操作する場合，どのような順序で呼び出しやデータ操作を行うかを示すことが動的な情報の記述になる．これを完全に詳細に書いたものがプログラムである．しかし，これらの動的な情報は，ⓐコンポーネントの機能やコンポーネント間関係の記述からかなり推定でき，ⓑ詳細な動的情報はプログラミング作業で定めるべきであって設計時には考慮すべきでない，という考えによりコンポーネント間関係の記述から除いた．ただし，静的な記述を一部補うために，使用するデータの内容の制限や，IF 条件による動作の制限などを明記できるようにした．また，どうしても詳細な動的情報が必要な場合は，$ ALGORITHM 文に書くことができる.

3.4　データ値の履歴の記述

コンポーネントの機能や処理方式を理解する上で，それが操作するデータの内容の理解が極めて重要である．ファイルやメッセージバッファの場合，データ要素（1 レコード分のデータ）の内部よりもデータ列全体としての内容の理解が重要である．このため，SDL ではコンポーネントが入出力するデータ列の内容を表現する**データ値の履歴** (value history) という概念を導入した.

データ値の履歴はコンポーネントが送り出す可能性のあるデータ列や受け取ることができるデータ列を形式言語の文法を記述する記法である正規表現や BNF 表現などで記述したものである．一つのファイルやメッセージバッファのデータ値の履歴は，一般に，データの送り出し側(producer)から見た表現と受け取り側(consumer) から見た表現に分けられる．例えば，次のようなメッセージバッファ M1 の宣言で,

$MB M1 : CHAR VHP {M1=line* eof, line=char⁺ eol, char=nonb|blk} VHC {M1=(word|blks)* eof, word=nonb⁺, blks=(blk|eol)⁺}

“CHAR” は M1 の要素データの属性，VHP と VHC に続く｛　｝でかこまれた記述は，それぞれ送り出し側と受け取り側から見たデータ値の履歴を示している．VHP と VHC が等しい場合は，単にVHで示す.

4.　設計言語プロセッサ

SDL による設計記述を設計データベースに登録し，全体としての一貫性を解析したり，各種の設計情報を得るために設計言語プロセッサがある.

プロセッサは，次の四つに大別される.

① 設計データベース：設計記述を関連をつけた形で保存する.

② SDL エディタ：設計記述の設計データベースへの登録，修正，削除を行う.

③ SDL アナライザ：設計データベースを調べて，設計の全体的な無矛盾性，一貫性を解析する.

④ SDL レポータ：設計データベースから各種の設計文書を出力したり，設計情報の検索を行う.

プロセッサの全体構成図を**図 11** に示す.

SDL は非形式的な叙述文を多く含んでいるが，これは分野によって，また特定のプロジェクトによって，統一した形式的記述にできることがある．この場合,

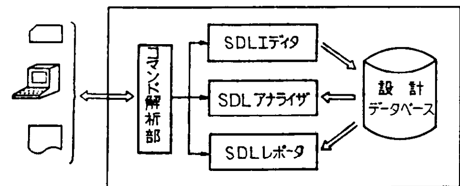


図 11　設計言語プロセッサ構成図

Fig. 11　The design language processor organization.

新たに導入した形式的記述を解析するシステムを容易に付加できるように，プロセッサは拡張可能なシステムになっている．ここでは，基本的なプロセッサの機能について述べる．

4.1 SDL エディタ

SDL エディタは，SDL で書かれた設計記述の構文／意味解析を行った後，その内容を解析して設計データベースへの登録，置換え，削除などを行う．これらの処理は下記の単位で行える．

① コンポーネント単位：コンポーネントのユニット名（またはユニット名の集合）を指定して，コンポーネント記述の登録，置換え，削除，出力を行う．

② 文単位：コンポーネントのユニット名（またはユニット名の集合）とその中の文頭語（または文頭語の集合）を指定して，文の登録，置換え，削除，出力を行う．

③ データ単位：データ名（またはデータ名の集合）と，データ名が一意でないときはその有効範囲を指定して，データ定義の内容を修正する．この場合，その定義が書かれているコンポーネントを指定しないで，データ名だけで修正を指示できる点が②と異なる．

このように，SDL エディタは，通常のテキストエディタと異なり言語の構文に従って編集の指定を行う．

設計データベース中では設計情報がコンポーネント間の関係をつけた形で保存されている．したがって1ケ所を修正すると関連項目も自動的に修正される．例えば，あるコンポーネントを削除すると，その親コンポーネントの内部コンポーネント宣言からそのコンポーネントは自動的に除かれる効果をもつ．これによって，コンポーネント間にまたがった全体的な統一性が常に保たれる．

4.2 SDL アナライザ

SDL アナライザは主にコンポーネント間にわたる関係の無矛盾性や一貫性を解析する．一般に，要素間にまたがる関連情報の誤りは，開発工程の後になる程発見と訂正が困難になり多大な工数を要する．アナライザはこれらの誤りを早期に発見するために，設計データベース内の情報の関連をたどりながら無矛盾性の検査を行う．

例として，コンポーネントの外部環境に対する操作の権限の妥当性検査について述べる．図12に示すように，コンポーネント S_1 が S_2, S_2 が S_3 を含み，その外部環境としてグループGのオペレーションを呼び出すとする．S_1, S_2, S_3 が呼び出せるGのオペレーションの集合をそれぞれ O_1, O_2, O_3 とすると，S_3 は S_2, S_2 は S_1 に許された操作の権限を越えることはできないので，

$$O_1 \supseteq O_2 \supseteq O_3$$

の関係が成立しなければならない．データの操作に関しても同様の権限の検査がなされる．

4.3 SDL レポータ

SDL エディタによる設計データベースの登録は主にコンポーネント単位，文単位で行われるが，SDL レポータによる設計結果の検索や各種の設計文書の出力は，主に複数のコンポーネントにまたがってなされる．

レポータの対象となるコンポーネントの範囲を限定するために，コンポーネントの包含関係や各種の条件を組み合せてコンポーネント集合が指定できる．例えば，

UN=A>X, Y…ユニット名による包含関係の指定
CT=ROUTINE…コンポーネントの種類の指定
MID=SHIGO…設計者 ID の指定
KEY="SDL", "DATA FLOW"…鍵語の指定

と指定すれば，コンポーネントAの子孫のうちXとYの子孫を除いた部分（図13）のルーチンコンポーネントで，設計者が SHIGO, かつ鍵語として "SDL" と "DATA FLOW" が書かれているコンポーネントの集合が指定される．

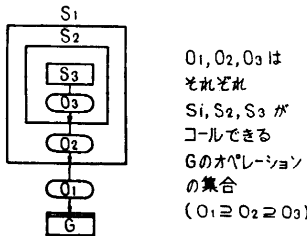


図 12 コンポーネント間関係の制限
Fig. 12 Restriction for inter-component relationship.

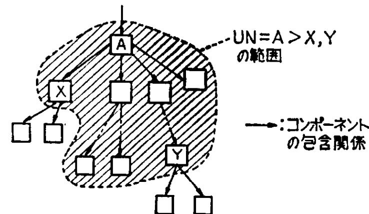


図 13 コンポーネント集合の包含関係による指定
Fig. 13 Illustration of components set representation.

このように指定されたコンポーネントの集合につき，

- コンポーネントの概要設計書
- コンポーネントの環境情報を含む詳細な設計書
- コンポーネントのインタフェース一覧
- コンポーネントにまたがる効率仕様一覧

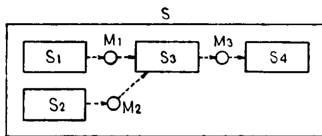
などが出力できる．また，コンポーネント間関連情報に関しては次のような出力が得られる．

- コンポーネントの包含関係によるシステム構成図
- 結合関係による呼び出し階層図，データフロー図
- データ操作のクロスリファレンス表
- ルーチンやオペレーション呼び出しのクロスリファレンス表

このような一括出力の他に，検索のために一部だけを見る各種機能も用意されている．さらに，これらの出力結果を SDL エディタや別の出力コマンドの指示パラメータとして使えるので，報告書作成機能や編集機能を組み合せたきめの細かい設計支援ができる．

5. FORTRAN, COBOL でのデータフロー機構

前述した方法論に基づいて設計したシステムを通常の順次処理プログラムで実現する場合，メッセージバッファによるプロセス結合の実現方法が問題になる．このため，通常の FORTRAN や COBOL のプログラムから簡単にメッセージバッファを使うためのメッセージバッファ模擬機構を作成した[14]．ここでは FORTRAN を例にとり，その機能を簡単に述べる．



(a) データフロー分割

```
DFBEGIN  S              -------- DFで実現されるコンポーネント名の指定
PROC.C   4              -------- プロセス数の指定
PROC     1, S1          ┐
PROC     2, S2          │------ プロセスコードと
PROC     3, S3          │       プロセス名の指定
PROC     4, S4          ┘
MB.C     3              ----------MB数の指定
MB       M1, 1, 3, d1, q1 ┐ MB名，producerとconsumerの
MB       M2, 2, 3, d2, q2 │-- プロセスコード，データサイズ(di)
MB       M3, 3, 4, d3, q3 ┘  バッファ容量(qi)の指定
DFEND
```

(b) (a)の構成の宣言文

図 14 データフロー結合の宣言

Fig. 14 Declaration of data flow combination.

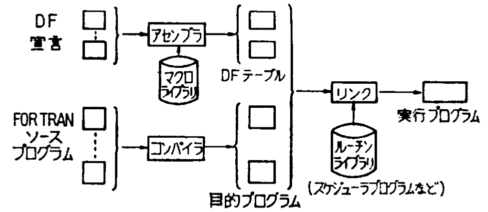


図 15 システム作成の手順

Fig. 15 System implementation processes.

今，図 14 (a) のように，コンポーネント S が，内部プロセス S_1, S_2, S_3, S_4 をメッセージバッファ M_1, M_2, M_3 によって結合したデータフロー分割で実現されたとする．ここで，S はプロセスでもルーチンでもよく，S_1, S_2, S_3, S_4 はプリミティブでもノンプリミティブでもよい．このとき，S の内部コンポーネント結合の様子を図 14 (b) のように宣言する．この宣言文を処理することによって S の DF テーブル（データフローテーブル）ができる．プロセス S_1, S_2, S_3, S_4 の目的プログラムに，この DF テーブルと簡単なスケジューラプログラム（アセンブラで約 200 ステップ）を結合すれば S の実行プログラムが実現できる（図 15)．

メッセージバッファの操作は FORTRAN の CALL 文で行う．例えば M_1 にデータ d を送る場合は

CALL MPUT (M_1, d)

M_1 からデータを x に受け取る場合は

CALL MGET (M_1, x, isend)

と書く．ここで isend はデータの終りを知るための変数である．これらの命令を実行すると，スケジューラプログラムが S の DF テーブルを使いながらプロセス間のデータの受け渡しのタイミングを制御する．例えば，S_1 で "CALL MPUT (M_1, d)" を実行したとき，もし M_1 のバッファがいっぱいならば，その CALL 文の次の文の番地（S_1 の実行再開番地）と，S_1 が待ち状態 (wait) であることを S の DF テーブルに書き込んで，実行可能状態 (ready) のプロセス（例えば S_3）の実行に移る．さらに，S_3 中で "CALL MGET (M_1, x, isend)" が実行されたとき，S_1 は実行可能状態にされる．

このように，データフローで実現される各コンポーネントごとに DF テーブルを持つことによって，局所的な実行管理ができ，2.2 節の設計方法論で示したような制御フローとデータフローが階層的に融合したシステムを FORTRAN や COBOL によって簡単に実現できる．

6. むすび

並列プロセスやメッセージバッファの概念を用いたデータフロー機能と従来の呼び出し関係を用いた制御フロー機能を融合した新しいモジュール化設計法，それに基づく設計を支援する設計言語とそのプロセッサ，およびその設計結果を自然な形でプログラム化するための簡単な機構について述べた．

データフロー分割での中間データの設定の仕方や，データベースなどの順次列データでない大容量データを使った場合の設計法などの点で，今後さらに研究を進めるべき問題が残されているが，この新しい設計法は従来提案されていた多くの設計法 (Jackson 法[7]，ワーニェ法[15]，Parnas のモジュール化[11]，Morrison[10] 等によるデータフロー分割など）を矛盾なく包含し，従来の設計法を系統的に統合させたもので，広い適用範囲を持っている．また，この新しい設計法を使うことによって，階層的な要素分割の過程でこれらの従来の設計法のそれぞれが最も適している場面を切り出すことができ，それぞれの設計法をより有効に適用することが可能になる．

この設計方法論に基づいた設計言語とそのプロセッサによって，設計者は必要最小限の情報を設計データベースに登録し，それをもとにさまざまの視点から開発対象システムを解析できる．この設計システムは，ACOS-6 の TSS 処理形態で対話的に利用できるように実現されている．これらの有効性は，今後の実際のソフトウェア開発への適用によってさらに評価していくつもりである．

この論文で示したデータフローと制御フローの融合方法は，単にソフトウェアだけでなく，従来の順次処理機械と最近話題になっているデータフロー機械の双対性に着目し，両者の利点を活かした新しい計算機アーキテクチャのあり方を探る指針ともなろう．

謝辞 この研究を進めるに当り協力していただいた日本電気中央研究所西村高志研究員をはじめ関係諸氏に感謝する．特にモジュール間結合関係の記述法とプログラミング支援ツールの設計に関しては西村氏の意見が反映されている．また，同祢津孔二コンピュータシステム研究部長と，藤野喜一基本ソフトウェア開発本部長には終始励ましと有益な助言をいただいた．ここに深く感謝する次第である

参考文献

1) Dennis, J. B.: First version of a data flow procedure language, Lecture Notes in Computer Science, Vol. 19, pp. 362-376 (1974).
2) Teichroew, D. and Herskey III, E. A.: PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems, IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 16-33 (1977).
3) Bell, T. E. et al.: An extendable approach to computer-aided software requirements engineering, ibid., Vol. SE-3, No. 1, pp. 49-60 (1977).
4) 内田裕士：ソフトウェア開発支援システムSSD, 情報処理学会ソフトウェア工学研究会資料 5-2 (1978).
5) 岩元莞二，藤林信也，紫合　治，西村高志：SDMS：ソフトウェア開発／保守支援システム，情報処理学会第20回全国大会，p. 327 (1979).
6) 紫合　治，藤林信也，岩元莞二：並列処理の概念を用いたモジュール分割，情報処理学会第16回全国大会，p. 573 (1975).
7) Jackson, M. A.: Principles of program design, Academic Press (1975).
8) Hansen, P. B.: Operating system principles, Prentice Hall (1973).
9) Balzer, R. M.: PORTS-a method for dynamic interprogram communication and job control, AFIPS Conference Proceedings 38 (SJCC) pp. 485-489 (1971).
10) Morrison, J. P.: Data stream linkage mechanism, IBM System J., Vol. 17, No. 4, pp. 383-408 (1978).
11) Parnas, D. L.: On the criteria to be used in decomposing systems into modules, Commun, ACM, Vol. 15, No. 12, pp. 1053-1058 (1972).
12) Shigo, O., Shimomura, T., Iwamoto, K. and Maejima, T.: Implementing the abstraction technique in software development, 2nd USA-Japan Comp. Conf., pp. 517-522 (1975).
13) 紫合　治，岩元莞二：ある制限をもつ並列処理システムのデッドロックについて，昭52年度電子通信学会情報部門全国大会，p. 68 (1977).
14) 紫合　治，西村高志：FORTRANにおけるメッセージバッファの模擬，情報処理学会第19回全国大会，p. 263 (1978).
15) ワーニェ，フラナガン（鈴木君子訳）：ワーニェ方式によるプログラミング学習 ㊤/㊦，日本能率協会 (1973).

（昭和54年8月30日受付）