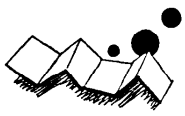
解　説



ソフトウェアを高信頼化するための設計・製造技法†

松　本　吉　弘††

1. まえがき

あるソフトウェアを動作せしめるとき，そのソフトウェア自身のもつなんらかの原因によって，動作の結果，外部へ与える働きが期待されたそれに比して，あらかじめ定めた許容量を越えて合致しないと判定されたとき，ソフトウェア故障 (software failure) が発生したと称し，この際の誤り (error) をもたらした原因をソフトウェア障害 (software fault) という．「誤り」は正しくない状態（またはその一部）のことで，すべての誤りが障害に関係しているとは限らない．データベースの項に人が不用意に入れた値によって故障が起ることもあり得る．

ソフトウェア信頼度 (software reliability) は2つの定義をもつ．第一の定義に従うと，信頼度とは対象とするソフトウェアを特定された環境で作動せしめるとき，ある期間にわたってソフトウェア故障が発生しない確率である．第二の定義に従うと，信頼度とは対象とするソフトウェアを特定された環境で作動せしめるとき，ある期間にわたって，そのソフトウェアに対して要求されている仕様を満たし得る能力の大きさである．第二の定義は，「ソフトウェア障害」の範疇に，要求されている仕様にソフトウェアの動作効果が合致しない事象（たとえばアナログ入力のプリントアウト値の精度が仕様以下）を含めるとするならば，独立させておく意義を失うかも知れない．しかし，設計・製造の観点からすれば，2つの定義を両立させておくほうがより自然と考えられる．たとえば，第一の定義はフォールトトレラント（耐障害力をもった）ソフトウェアの設計問題により強く連関し，第二の定義はより品質の高いソフトウェア (reliable software) のための設計という問題に結びついているからである[27]．

一般にハードウェアの信頼度とのアナロジィでソフトウェアの信頼度を考えたい場合には第一の定義に従う．単に「信頼できるソフトウェア」というときは，あたかも「信頼できる人」というイメージとのアナロジィで，第二の定義のほうがよりふさわしいと感じさせる．第一の定義は，信頼度モデルと称するものによって定量化のための道へ結びついているが，第二の定義はより主観的要素を含むために，実用面での定量化の困難を予想させる．

さて，筆者に与えられた命題では，他の解説との重複を避けるために，第二の定義に重点をおくことが要求されている．したがってフォールトトレラント問題には触れていない．また，ソフトウェアの検査，テストなどの問題も，本稿に与えられた領域外にあるものと考えている．

2. 信頼できるソフトウェア (reliable software)

第二の定義に従って，信頼できるソフトウェアという概念の分析を行う．**図-1** はソフトウェアの生成過程を示している．ユーザから与えられる要求は，一般に体系化される以前の抽象の不規則な集合である．これをいくつかの段階を経て，徐々に計算機ハードウェアという物理的裏付けをもったプログラムおよびデータのコードへと変換してゆく過程がソフトウェアの設計（広義）である．この過程はいくつかのサブステップ，たとえば要求定義，設計（狭義），プログラミング，などに分割して処理される．（本質的には抽象が具象へと次第に形を変えていく連続プロセスと考えるべきである．）いま，この過程をいくつかの小さな変換作業 $P_0, P_1, \cdots P_n$ の連接によって実現するものとする．変換 P_i を始める際に与えられる文書を $Form_i$ とし，P_i の結果，P_{i+1} に与えるべく作成された文書を $Form_{i+1}$ とする．変換 P_n が終った結果生れる $Form_{n+1}$ は計算機ハードウェアとの対応をもったプログラムとデータである．

図の右側には組合せの過程を示している．プログラ

† Design and Manufacturing of the Reliable Software by Yoshihiro MATSUMOTO (Heavy Apparatus Engineering Laboratory, Toshiba Corporation).

†† 東京芝浦電気(株)重電技術研究所

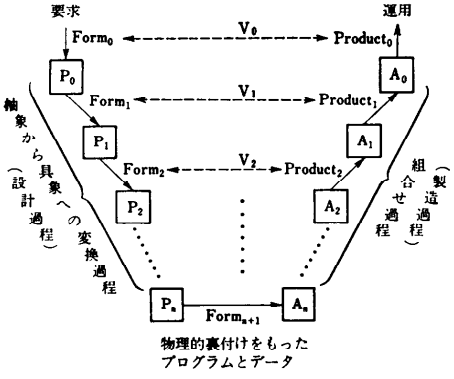


図-1　ソフトウェアの生成過程

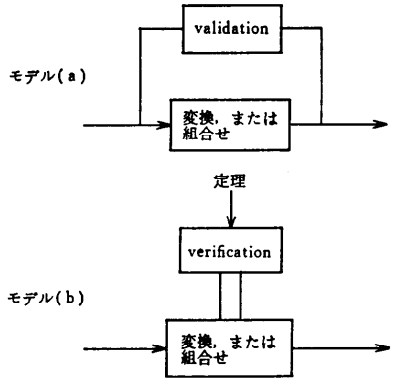


図-2　確認行為

ム，データのモジュールは互いに組み合わされ，さらにこれが計算機ハードウェアと組み合わされ，最後には外部のシステム機器と組み合わされる．連接している各部分組合せ作業を $A_j(j=n, n-1, \cdots, 0)$ とし，A_j への入力を $Product_{j+1}$，出力を $Product_j$ とする．変換Pの数と，組合せPの数は同数である必要はないが，ここでは同数になるように調整されているものとする．

さて，第二の定義に従った信頼できるソフトウェアとはつぎのような要件を満たしたものと考えられる．

（1）　外部のシステム機器と組み合わされて運用に入ったソフトウェア $Product_0$ が，要求 $Form_0$ を十分に満たしていること．

（2）　$Product_0$ は要求 $Form_0$ に明白に示されていないが，暗黙のうちに示されている条件，制約なども満たしていること．

（3）　外界の擾乱などによってソフトウェア内に誤りが発生しないように，外界の異常に対する抵抗力を強くすること（たとえば，入力値の妥当性検査など）．

信頼度に関する第二の定義はソフトウェアの品質(quality)の問題に密接に関係している．フォールトトレラントの問題を除外して考えると，品質の高いソフトウェアとはつぎの評価に耐え得るソフトウェアのことと考える．

（i）　一貫性をもつこと：一貫性 (consistency) とは $Form_i$ のなかで定義されたオブジェクト(もの)，および関係が，$Form_j$ $(i \neq j)$ のなかで定義されたそれに対応するオブジェクト，および関係と矛盾しない性質を意味する．$Product_i$, $Product_j$ $(i \neq j)$ 相互間．また，Form と Product 相互間についても同様である．

一貫性を保つための確認行為は 図-2 に示すような2つのモデルで表わせる．モデル(a)は通常の確認(validation)である．モデル(b)は verification（検証と訳されている）とよばれているもので，入力が定理に従っていることを検証しながら変換，組合せが行われることを示している．各モデルにおける確認手法にはつぎのようなものがある．

モデル(a)　デザインレビュ，デュアル開発チーム方式[26]，パス解析，実行模擬

モデル(b)　構造定理[37]，分割定理[37]，グラフモデル，有限機械モデルなどに必要な意味づけを賦与し，これが成立することを調べる．

（ii）　必要な精度をもつこと：処理の結果，得られる数値が指定された精度を有すること．

（iii）　完結性 (completeness) をもつこと：各 Form 内で定義されたオブジェクト，および関係，さらに各 Product 内に実現されたオブジェクト，および関係に脱落や不足がないことを完結性とよぶ．

（iv）　簡潔性をもつこと：ソフトウェアのもつモジュール間の静的構造，制御流れ，データ流れ，およびモジュール内の制御流れ，データ流れはできるだけ簡潔であるほうがソフトウェアの品質は高いものと評価される．このために複雑度 (complexity) という評価指数が用いられる．複雑度の定義法は種々あるが，つぎの方法[18]はもっとも分りやすい．ひとつの目的をもった作用，または思想を表わした単位をノードで表わし，ノードの活動を支配するノード間信号を有向矢で表わし，これらノードを有向矢で形成されたグラフを制御流れ強結合有向グラフGとよぶ．同様に，各ノード内に蓄積している記憶に対して，他のノードがアクセス（読みとりのためのみ）するパスを有向矢で表わ

し，上記のグラフG内の有向矢のうち，データを含む有向矢をこれに加えて形成したグラフをデータ流れ強結合有向グラフG′とよぶ．

G，またはG′に関してつぎの式を用いてもとまる値 $V(G)$（または $V(G')$）を複雑度と考える．$V(G)$ はグラフG内に含まれる1次独立な基本パスの最大数を表わしている．

$$V(G)=e-n+2p \qquad (1)$$

ただし，e：グラフ内の有向枝の数

n：節点の数

p：グラフ集団の数

（v） 使いやすさをもつこと：ソフトウェアの目的，内容，使い方，効果を示す文書が完備し，記述が良質であること，再利用性（re-usability：ソフトウェア，またはその一部を再利用のために必要な便宜を整えてあること），保守性を具えていること，など．

このほかにも品質評価尺度は限りなく考えられる．これらの尺度は応用ごとに重みづけして用いられる．

3. 設計・製造過程と信頼度

図-1 の組合せ A_n をコンパイルなど物理的コード生成作業と考えると，A_n から A_0 に至る過程はソフトウェアの製造過程であり，P_0 から P_n に至る過程はソフトウェアの設計（広義）過程である．

この機構のなかで，2章で述べた要件をソフトウェアが満たすためには，この設計・製造過程につぎに示す基本的な確認行為を組み入れる必要がある．

確認(A)：$Product_i$ を $Form_i$ によって確認する．

確認(B)：$Form_i$ が $Form_{i-1}$ によって示されている要件を十分に満たしていることを確認する．

確認(C)：$Product_{i-1}$ において $Product_i$ のすべてが正しい接合状態で組み合わされていることを確認する．

確認(D)：i に関して $Form_i$，$Product_i$ が，それ自身で矛盾や誤りを含んでいないことを確認する．

確認(E)：$Form_i$ が変換 P_i によって実現可能であることを確認する．*

図-3 はこれら確認行為が，ソフトウェアの設計・製造過程の中にどのように組み込まれるかを示している．確認の際の照合が厳格であればあるほど，ソフトウェアの信頼度は大きくなる．

一般に，確認はテストケースとそれに付属したテストデータに対して，被確認対象が表わす効果を測定し，要求と照合する行為を伴う．テストケースやテストデータは，明白な量として表わせる場合と，経験者の直観や知識というような内在的な存在である場合とがある．デザインレビュ，ウォークスルーによる確認行為は後者の場合に実施される．

1組のテストケースによって行う確認行為を，いまここではラン（run）とよぶことにする．本章の最初にあげた各確認行為には複数のランが必要であり，それによって被確認対象に対する有意入力領域（全入力領域の一部であるが，それに含まれる入力に関してランを行うことによって，被確認対象が十分信頼ありと判断される入力領域）のすべてをカバーするものと考える．

テストケース（テストデータとも）を明白な量として扱える場合について考える．Form がハードウェアの上で実行可能な物理的コードに到達していれば，ハードウェア上でのランが可能であるが，それ以前の

```
Π0 : do   Form1←P0 (Form0)
          until (確認 (B),(E) & 確認 (D)) is successful
          end;
Π1 : do   Form2←P1 (Form1)
          if 確認 (E) is unsuccessful then
              go to Π0;
          elseif (確認 (B) & 確認 (D)) is unsuccessful then
              go to Π1; endif;
          endif;
          end;
Π2 : do   Form3←P2 (Form2)
              (以下 Π1 ブロックと同じ)
                    ⋮
αn : do   Productn←An (Formn+1)
          if 確認 (A) is unsuccessful then
              go to Πn;
          endif;
          end;
αn-1 : do Productn-1←An-1 (Productn)
          until 確認 (C) is successful
          end;
          if 確認 (A) is unsuccessful then
              go to Πn-1;
          endif;
αn-2 : do Productn-2←An-2 (Productn-1)
              (以下 αn-1 ブロックと同じ)
                    ⋮
α0 : do   Product0←A0 (Product1)
          until 確認 (C) is successful
          end;
          if 確認 (A) is unsuccessful then
          go to Π0;
          endif;
```

図-3 設計・製造過程における確認行為の存在

* 設計過程（P変換の過程）において「組合せによって実現可能であること」を前提としているので，「$Product_{i+1}$ が組合せ A_i によって実現可能である」という《確認(E)の対偶》は省略してある．

過程における Form では，シミュレーションによるランが必要となる．シミュレーションは Form の中で定義されたオブジェクトと，オブジェクト間の静的関係の上で，テストケースに対する制御流れを追跡し，オブジェクトのもつ作用の連接を要求と照合し，確認する動作である．この作用の連接のことを動的モデルとよぶことにする．明白なテストケースによる確認行為に対しては，この行為の基準となる Form の作成過程において動的モデルを想定し，テストケースを同時に作成することが必要である．

4. 設計・製造技法

ソフトウェア生成過程は 図-1 に示したように，要求 Form$_0$ から Product$_0$ への変換・組合せ過程である．この過程に一貫した連続性があると考えたいが，要求がまったくの抽象であるのに対して，Product$_0$ は物理的存在であるため，途中で質的変化を与えなければならない．したがって，図-1 に示したようないくつかのサブステップ（PまたはAのこと）を設けて，段階的な質的変化をはからざるを得ない．サブステップの切り方に原則はない．与えられた問題に合わせて最適のサブステップを選べばよい．サブステップの説明に入る前にまず一般的思想について述べておく．

ソフトウェアを体系的に具現するために，つぎに示す一連の思想に従う．

(B1) ソフトウェアを構成すべき基礎的サブシステムを識別し，その役割を決定する．これらサブシステムはこの段階では抽象である．

(B2) B1 で識別したサブシステムから成るソフトウェアの静的構造を決定し，各サブシステム間の関係を定義する．

(B3) 各サブシステムに物理的な具体性をもったモジュール空間（内部にはコードが入る予定）を割り当てる．

(B4) 各モジュール間のインタフェース（接合部の形式と接合関係）を決定する．

(B5) ソフトウェアの動的モデルを想定し，要求されている各機能に対する動的モデルの中の部分構成要素の対応を検査し，要求に対する確認を行う．

(B6) モジュール空間内にプログラム，またはデータのコードを割り付ける．

B2 で決定した静的構造に対して，B3 以下の思想の実施を，構造の最上位の部分にあるモジュール空間から順に行う方法がトップダウン，最下位から始める方法がボトムアップである．ボトムアップは，B3 で創造するモジュール空間に対して既存のコード，または前例経験の適用を予定する場合に採用することが多い．この2つの方法はソフトウェアの性質や，設計者に与えられた環境に合わせて併用されるのが普通である．

以下，図-1 のサブステップに従って，設計・製造技法の一例を示す．

変換 P_0：ユーザから発せられる原始的要求は，一般に離散的な自然文による意志表現の形式をとる．ひとつの自然文がひとつの有意（他と区別できる）要求を表わすようにし，各文に固有の識別記号を与える．すなわち，ひとつの記号がそれによって他と識別しておかしくない単独の要求項目を代表しているようにし，それぞれを要求 $r_i(i=1,2,\cdots N)$ とする．すべての要求 r_i のユニオン集合が全要求Rを十分に満たす必要がある．すべての i に関して，r_i はつぎの4つ組で表わされる．

$$r_i=(x_i, y_i, f_i, p_i) \qquad (2)$$

ただし，x_i, y_i はそれぞれ，目的とするソフトウェアに対する入力 X および出力 Y の部分集合である．f_i は機能 F の要素とする（ただし，$Y=F(X), y_i=f_i(x_i)$）．p_i は f_i に対する制約，または f_i が具えていなければならない特性に関する主張を表わすものである．f_i と f_j（ただし，$i\neq j$）の間の関連度を s_{ij} とする．s_{ij} は f_i と f_j との間の共通性を量的に示したもので s_{ij} の定め方については種々の研究がなされている[8),12)]．すべての r_i と s_{ij} が定義されると，r_i をノード，s_{ij} をノード間を連結するリンクとする**図-4** のようなグラフが描ける．いくつかのノードのグループをクラスタ (cluster) と称している．クラスタ内の全リンクからクラスタ強度をもとめることがで

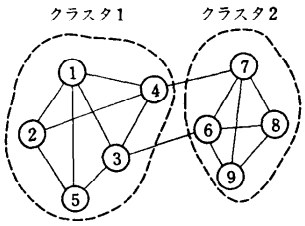


図-4 クラスタリング説明図

き，クラスタ間全リンクからクラスタ間結合度をもとめることができる．適切なクラスタリングを行うことによって，目的とするソフトウェアの静的構造を創成することができる[2),9)]．クラスタリングアルゴリズムに関しては多くの研究がある[1),7)]．

ソフトウェアは計算機システムの中に格納され，計算機システムはさらに大きなシステムの一部として包含されるのが普通である．もっとも外にあるシステムから順にシステム構造を決定してくると，ソフトウェアに到達したときには，式(2)の x_i, y_i（または X, Y）の構造はすでに確定していると考えられる[13),16)]．これがために，Fの構造もおのずから定まってくる．このような見方が適用できる問題では，Fの構造を見出すためのクラスタリングは必要としない．PSL/PSA[31)]はこのような場合に適用して，要求Rの構造を明示するのに便利である．**図-5** は，PSL のモデルと式(2)との対照を示している．変換 P_0 では，このようにして目的のソフトウェアに期待されている機能Fの構造を決定する．この構造がソフトウェアのもっとも基本となる静的構造を与える．

変換 P_1：上で決定した構造内の各要素（機能単位）相互の関係を定義する．関係 (relation) はつぎの3つに分類される．

- 静的構造における相対関係
- データ情報の授受，または共有関係
- 制御の流れの上における相対関係

関係の定義はできるだけ理解しやすい様式に従って記述し，多くの人々による目視検査が可能であるようにせねばならない．SADT, HOS, SREM, ペトリネット法，EPOS[3)]，状態遷移図法[15)]などがある[24)]．はじめの2つはデータ情報の記述により適しており，あとの4つは制御流れの記述に適している．System Encyclopedia Manager[32)]では関係の定義を自由に追加変更できる．これら技法（一部のものは計算機で支援されている）の目的とするところは，設計者の思想を体系的に展開できること，確認(B)，(C)，(E)を容易に行い得ることにある．

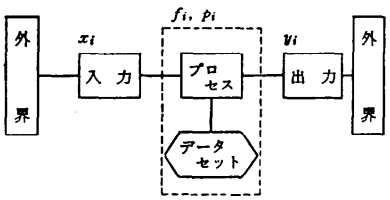
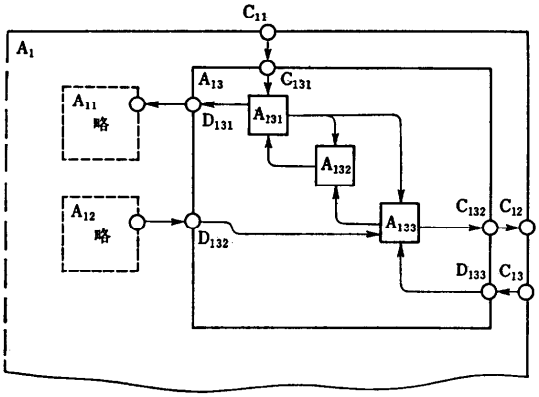


図-5 PSL モデルと要求4つ組との対照

変換 P_2：変換 P_0, P_1 でもとめた構造要素（機能単位）のそれぞれは，ソフトウェア構成要素のあるひとつのグループに対応するものとして，以後の設計が進められる．このサブステップでは，変換 P_0, P_1 でもとめた機能単位を，物理的モジュールの割当てを受けるためにさらに分割し，分割した最小機能単位にモジュール空間（中にプログラム，またはデータコードを充当する予定のもの）を割り当てる．この機能単位分割と，モジュール割付け作業の部分が，ソフトウェア設計中でもっとも設計者の知力を必要とするところである．一般にソフトウェアの機能単位に影響を与える重要な因子に，制御とデータがある．事務処理のような問題では制御がほとんどデータの移動に依存して発生するが，外界プロセスの制御問題では制御が外界のイベントによって独立に流れ，データの流れがこれに従属して発生するのが普通である．モジュールの大きさをいかに選ぶかによって，モジュール間の静的構造，制御流れ，データ流れ（このステップではモジュール間の問題に限定している）の複雑さが変ってくる．モジュールとモジュール間の静的構造を設定し，そのときの制御流れ，データ流れ，それぞれの複雑度 (complexity) を定量的に表わせば，これによって設計品質が評価できる．複合設計法，ジャクソン法，ワーニェ法，オートマトン法などの既存技法はいかにして，この3者に関する最適解を与えるかを提案したものである．

上で述べたモジュール空間に関しては，情報隠ぺい (information hiding)，抽象データ型，ADA のパッケージ (package) および多くのモジュールを取り扱った言語[11),34)]などによって思想的確立が行われてきた．

変換 P_3：上でもとめたモジュール空間相互の間のインタフェース（変換するデータ，制御の定義），および構造を明白に文書化するサブステップである．言語を用いる方法と，図を用いる方法がある．さまざまな言語や図様式が提案されている．このサブステップでは，上のモジュール空間は自然にコンパイル単位という概念によって置換される．ひとつひとつのモジュールは個別にコンパイルされることを前提にされるので，リンク方法への配慮が導入される．MESA[10)]，MIL[6)]，DREAM[28)]，INTERCOL[33)]，その他[14),30)]は言語の代表例である．SARA[23)]は図様式の一例である．



説明:
A: モジュール
C: 手続き呼びのインタフェース
D: データ参照のインタフェース
箱の上辺と右辺: 呼びのインタフェース
箱の左辺と下辺: 参照のインタフェース

図-6 ソフトウェア構成を示す図様式の例

言語と図様式が組み合わされたものも多い. 筆者らがSWB システム[17]で実用している図様式の例を参考までに **図-6** に示しておく.

このサブステップで用いる言語の処理系は, 単にモジュール間構造検査, データおよび制御インタフェースの型検査, 同時実行プロセス間の交信方式検査, リンカへの指示などの機能だけをもてばよいのであるが, さらに通常のプログラミング言語のような内部手続き記述に対する処理機能までを包含し, 変換 P_4 までもカバーする大きな処理系として発展させようとしているものに ADA, PEARL がある.

変換 P_4: モジュール空間の中にプログラミング言語を用いて, コードを割り付けてゆく過程である. いくつかの公理 (axiom) が見出されている[19]. この公理を組み合わせた定理 (theorem) を構築し, この定理に従ってプログラムを組むことが, 確実な (secure) プログラムを作るための大前提である. 定理はそれぞれの問題ごとに作り得るが, できるだけ広い範囲で使い得るような定理を見出すことが重要な関心事である. 記述したプログラムがこの定理を正しく満足することを検証する (verify) システムのことをベリファイヤ (verifier) とよんでいる. ベリファイヤの利用は, その繁雑さのためにはかばかしくないが, さらに研究が進められている[21].

連接, 選択, 反復はもっとも初歩的な公理のひとつであるが, これに従ってプログラミングを実施するための技法が数多く提案されている[5]. さらに一般的でわかりやすい定理が発見されれば, 一層正しいプログラムへの道が開けてくると考えられる.

組合せ (製造) 過程A: 変換過程で作られた各モジュールを組み立てる過程であるから, ボトムアップ方向で進めるのが一般的である. 最後の組合せ A_0 では外界機器, 外界システム, 人間要素 (human factor) との組合せが行われる. これらの過程ではツールが重要な役割を果す. インタフェース宣言を付したモジュールは部品としてデータベースに格納される. 結合されるモジュールの接合点インタフェースは変換過程で嵌合条件の検査をすませておかねばならない. 組合せ過程におけるテスト, 確認に関しては本稿に与えられた領域の外であるので述べない.

製造用ツールは, たとえばつぎのような機能をもち, 体系的な組合せ方法を履行せざるを得ないようにしむけていることが多い.

(イ) モジュール間構造に従ってモジュール (ソース, オブジェクトコード) 格納ファイルへアクセスすることができる.

(ロ) ソースコードの変更および変更履歴の記録が可能で, オブジェクトコードへの変換, リンク, ターゲット計算機へのダウンロードをコマンド指令によって容易に行うことができる.

(ハ) テストケース, テストデータを管理し, これをテストのために, ある体系に従ってモジュールに入力として与える, テスト用ツールへの連繋をとることができる.

(ニ) ターゲット計算機上のオブジェクトモジュールのパッチングを可能とするための分離コンパイル, リンクロードが可能である.

(ホ) ターゲット計算機上のオブジェクトモジュールをアップロードし, そのコードを理解可能のコードへ変換する.

$Form_0$ から $Form_{n+1}$ への変換を総合的に振り返ってみる. $Form_0$ は人間の思想に起因する抽象であるのに対して, $Form_{n+1}$ は計算機のもつ物理的アーキテクチャと密接に関係した具象である. そのため, $Form_0$ から $Form_{n+1}$ への変換には2つの大きな質の変換点がある. 第1の変換点は目的/制約領域 (what phase) から実現領域 (how-to phase) への移行であ

り，第2の変換点は実現領域から資源（リソース：プログラム，データ，ファイルなど）領域への移行である．本稿で述べた変換 P_1 は前者，変換 P_4 は後者に該当する．

第1の変換点では与えられた目的/制約にもとづいて実現のためのアルゴリズム・手段の選択，評価を必要とし，人工知能技術の応用などが検討されている．

第2の変換点では与えられた実現方法に対して，合理的に，かつ効率良く資源を割り付けることが必要となる．このような問題を含む $Form_0 \rightarrow Form_{n+1}$ 変換が完全に形式化されれば，より一層信頼し得るソフトウェアが作られることになろう．

5. むすび

ユーザの要求をソフトウェアによって実現するまでの過程の一部分であるひとつ，または複数のサブステップに関する技法は数多く提案されているが，全過程を統一的思想で貫いた技法はまだ実験段階である．

真に信頼できるソフトウェアのためには，全ライフサイクルに統一的効果をもたらす一貫技法と，それを支援する環境ツールの確立が望まれる．

参考文献

1) Anderberg, M.: Cluster analysis for applications, Academic Press (1973).
2) 新井，田村，岡村：システム計画技術の動向，東芝レビュー，Vol. 35, No. 13, pp. 1128-1131 (昭55 12月).
3) Biewald, J. et al.: EPOS-A specification and design technique for computer controlled real-time automation systems, Proc. 4th International Conference on Software Engineering, Munich (Sept. 1979).
4) Boyer, R. S. et al.: SELECT-A formal system for testing and debugging program by symbolic execution, ACM SIGPLAN Notices, Vol. 10, No. 6 (1975).
5) Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R.: Structured programming, Academic Press, London & New York (1972).
6) DeRemer, F. et al. Programming-in-the-large versus programming-in-the-small, Trans. IEEE, Software Engineering (June 1976).
7) Hartigan, J.: Clustering algorithms, John Wiley (1975).
8) Huff, S.: Decomposition of weighted graphs using the interchange partitioning technique, Technical Report No. 3, Sloan School of Management, MIT (1979).
9) Huff, S. and Madnick, S.: An extended model for a systematic approach to the design of complex systems, Technical Report, No. 7, Sloan Scholl of Management, MIT (NTIS No. A058565) (July 1978).
10) Lauer, H. C. et al.: The impact of Mesa on system design, Proc. 4th International Conf. on Software Engineering, IEEE, pp. 174-182 (1979).
11) Liskov, B. H. and Zilles, S. N.: Programming with abstract data types, SIGPLAN Notices, Vol. 9, No. 4 (Apr. 1974).
12) Mariani, M. P. and Palmer, D. F.: Tutorial: Distributed system design, Chapter 3, IEEE Computer Society, Order # 267 (Oct. 1979).
13) 松本：工学系システムに対するシステム制御技術の適用，東芝レビュー，Vol. 35, No. 13, pp. 1132-1134 (昭55. 12月).
14) 松本：あるソフトウェア生産現場の実際，bit. Vol. 13, No. 11, pp. 4-10 (Oct. 1981).
15) Matsumoto, Y.: A method of software requirements definition in process control, Proc. COMPSAC '77, IEEE Computer Soc., Chicago, pp. 128-132.
16) Matsumoto, Y. et al.: A Method to bridge discontinuity between requirements specification and design, Proc. COMPSAC '80, IEEE Computer Soc., Chicago, pp. 27-31 (Nov. 1980).
17) Matsumoto, Y. et al.: SWB: A Software Factory, Proc. Symposium on Software-Engineering Environments, Geselschaft für Mathematik und Datenverabeitung MBH, Bonn, pp. 16-20 (June 1980).
18) McCabe, T. J.: A complexity measure, IEEE Trans. Software Engineering, Vol. SE-2, No. 4 (Dec. 1976).
19) Mills, H. D.: How to write correct programs and know it, Proc. International Conference on Reliable Software (1975).
20) Musa, J. D.: The measurement and management of software reliability, Proceedings of IEEE, Vol. 68, No. 9, pp. 1131-1143 (Sept. 1980).
21) Neumann, P. G. and Walker, S. T.: Contributions from VERkshop II, ACM SIGSOFT Software Engineering Notes, Vol. 6, No. 3, pp. 1-63 (July 1981).
22) Osterweil, L. J. and Fosdick, L. D.: DAVE-A validation and error detection system for Fortran programs, Software-Practice and Experience, Vol. 6 (Oct.-Dec. 1976).
23) Penedo, M. H., Berry, D. M. and Estrin, G.: An algorithm to support code-skeleton for concurrent systems, Proc. 5th Software Engi-

neering Conf., pp. 125-134 (1981).
24) Ramamoorthy, C. V. and So, H. H.: Software requirements and specification: status and perspectives, Tutorial: Software methodology, IEEE, Cat. No. EHO 142-0. pp. 43-164.
25) Ramamoorthy, C. V. and Bastani, F. B.: Software reliability-Status and perspectives, submitted to the IEEE Trans. on Software Engineering (1980).
26) Ramamoorthy, C. V., Mok, Y. R., Bastani, F. B. and Chin, G.: Application of a methodology for the development and validation of reliable process control software, Proc. of COMPSAC '80, pp. 622-633 (1980).
27) Randell, B., Lee, P. A. and Treleaven, P. C.: Reliability issues in computing system design, Computing Surveys, Vol. 10, No. 2, pp. 124-165 (June 1978).
28) Riddle, W. E.: An assessment of DREAM, Software Engineering Environments, North-Holland Pub. Co. (1980).
29) Ryder, B. G. et al.: The PFORT verifier, Computer Science Report No. 12, Bell Laboratories (July 1975).
30) 紫合 治他：統一的設計方法論に基づくソフトウェア設計システム，情報処理. Vol. 21, No. 5 (May 1980).
31) Teichroew, D. and Hershey III, E. A.: PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems, Trans. IEEE, Software Engineering (Jan. 1977).
32) Teichroew, D.: Overview of META system and the Generalized Analyzer, ISDOS Technical Memo, No. META-1, Univ. of Michigan (Sept. 13 (rev.), 1978).
33) Tichy, W. F.: INTERCOL user manual, Carnegie-Mellon Univ., Dept. of Computer Science (1979).
34) Wirth, N.: Modula: A language for modular multiprogramming. Software-Practice and Experience, Vol. 7 (1977).
引用文献の記載のないものについては，つぎのような文献を参照されたい.
35) Ramamoorthy, C. V. and Yeh, R. T.: Tutorial: Software methodology, IEEE Catalog No. EHO 142-0 (1978).
36) 松本：ソフトウェアの考え方・作り方，電気書院 (昭56).
37) Bergland, G. D. et al.: Software design strategies, IEEE Cat. No. EHO 184-2 (1981).

(昭和56年11月19日受付)