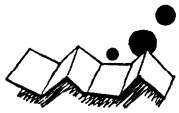
解　説

設計とプログラミングの自動化†



松　本　正　雄††

1. はじめに

ソフトウェアの品質や生産性の向上はもとより，総合技術としてのソフトウェア信頼性工学のさらなる充実発展が要請されている．そうしたなかで，設計と主としてプログラミングの自動化技術は，要求定義，検証技術，テスト技術とともに，ソフトウェア開発の重要技術である．

（1） 要求仕様と設計の関連……要求仕様は，実現したいシステムの現実世界での要件を仕様化したものであり，設計への入力となる．要求仕様は，システム作成依頼者と開発者双方が要件を確認し，合意を成立させる上で必要である．設計は，抽象世界での実現の仕方を解き明かすことである．設計内容は，開発者内部での意志疎通や，審査，インプリメンテーション，検査などの目的のため，仕様化および設計文書化される．

（2） 設計技術……過去10年以上にわたって，設計方法論がいくつか提唱されてきた．大別するとシステムのモジュールへの分解に関係したもの **2.** と，モジュール自身の設計に関係したもの **3.** の二つである．現在も拡張や改良が続いている．たとえば，システムのふるまいを状態遷移の観点から仕様化する技法において，階層性を許すなど拡張が続いている．対象指向（object-oriented）の概念を取り入れた開発法も提唱され始めている[4]．また，アルゴリズム発見を助ける目的で，人工知能を取り入れたアルゴリズム設計も研究されている．

（3） ソフトウェア CAD……コンピュータ，ネットワーク，ワークステーションの普及にともない，設計技法を駆使して，机上でのみ設計作業を行うだけでなく，コンピュータ上で作業してゆけるようソフトウェア CAD (Computer-Aided Design) が提供されるようになった．設計したシステムの動的ふるまいをシミュレートしながら確認できたり，インプリメンテーションの自動化につなげることができるので有用性が増している．図形やアイコンを使った設計を通じて表現の標準化が促進されることが特徴の一つである（**4.** 参照）．

（4） インプリメンテーション自動化……理論として定理自動証明が合成と検証の2面から長らく研究されてきている．これを使った実際の自動プログラミング系も現れ評価が始まった．また，部品を使ったプログラム合成は，いち早く実用段階に入っている．抽象化レベルのより高い仕様言語からのプログラム生成もいくつか研究されている．自然言語や図・表によるプログラム詳細仕様から，知識ベースを用いてプログラムを生成する方式の研究が，その例である．さらに，広スペクトラム超高級言語の開発とそれからの自動プログラミングが現在活発に研究されている．これらについては **5.** で述べる．

以下，設計技法，ソフトウェア CAD，インプリメンテーション自動化について，主なものを取り上げ解説する．

2. システム分解の方法

システムを実現することは，基本的に**分解** (decompose) と**組成** (compose) の二つの過程を踏む．分解は，与えられた条件のもとでシステム全体をいくつかの構成要素に分け，それらの内容を定義するとともに，構成要素間および外部環境とのインタフェースを規定することであり，ソフトウェア設計の主要部分を占める．分解は，モジュールのレベルまでのものと，モジュール内に対するものに大別される．組成は，分解で得た結果をもとに，ソフトウェアを作り上げることである．

システム分解の目的としては，大きく次の二つがある．

- 複雑さの克服：複雑なシステムを単純な部分に分けて理解したり取り扱うことができるようにする．
- 開発作業の分担：いくつかの部分の開発を分担し

† Software Design and Programming Automation: Present and Future by Masao MATSUMOTO (NEC Software Product Engineering Laboratory).

†† 日本電気(株)ソフトウェア生産技術研究所

て，並行に進められるようにする．

このため，モジュールはできるだけほかと独立になっているのが良い．すなわち，インタフェースが単純で，そのモジュールを使うときに必要な知識が少なくてすむことが要求される．

2.1 基礎的技法

モジュール分割の従来から提唱されている基礎的な主要技法としては，次のようなものがあげられる．

- **システムの階層分割**[19),20)]：システム全体から，最終的なプログラムモジュールへと段階的に分割していく技法．この場合，上位のレベルでは，分割の構成要素は，サブシステム，ジョブなど，プログラムモジュールとは直接対応しない概念的なまとまりになる．
- **手続きのトップダウン設計**[25)]：主手続き，それから呼ばれる手続き，さらにそれから呼ばれる手続き……，という順に設計を進める．
- **データ抽象化設計**[9),16),17)]：データとそれに関係するいくつかの機能（データの操作）を一つのモジュールにまとめ，そのデータを使う側に対して，データの実現の仕方などの詳細な情報を隠してしまう（抽象化する）技法である．
- **データフロー設計**[8),12)]：順次データ列の変換のされ方を中心に設計を進める．概念的には，並列処理プロセス要素が順次データ列（datastream）を介して結合される構造となる．

2.1.1 技法の位置づけ

システム全体から最終的なモジュールに至るまでの設計の枠組として，システムの階層分割が使える．また，階層分割の各段階で，どのような構成要素に分割しどのように関係づけるかの技法として，トップダウン設計，データ抽象化設計，データフロー設計などが使える．

トップダウン設計は，呼び出し関係の上位のモジュールが下位を制御する主要な役割をするので**中央分権型**といえ，データ抽象化は，下位の抽象データモジュールが重要になるので**地方集権型**といえよう．またデータフロー設計は，各要素の独立性が高く要素間に制御の上下関係がないので**連邦型**と考えられる．

2.1.2 システムの階層分割設計

システムの階層分割では，まずシステム全体の外部環境（入出力データなど）を明らかにする．次にシステム全体をより詳細な構成要素（サブシステム）に分け，それらの間の関係を定める．この時点で，各構成要素のインタフェースが明らかになる．さらに各構成

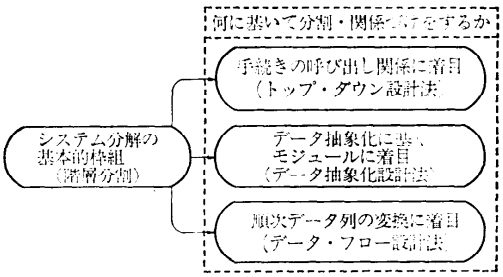


図-1 システム分解の基礎的技法

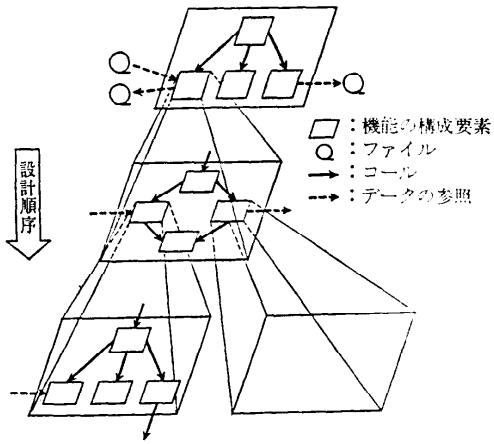


図-2 システムの階層構造化

要素を詳細化する．このように，システム，サブシステム，サブサブシステム，……と分割を繰り返し，最終的には簡単にコーディングできるプログラムモジュールレベルの構成要素に至るまで分割を進める．つまり，逐次階層化を進める設計（**Layer-by-layer Design**）である．このようにして，図-2 に示すようなシステムの階層構造が得られる．このような階層化の概念に基づいた技法として，SADT[19)]，SDMS[20)] などがある．

システムの階層化では，図-2 にも示されているように，上位レベルの構成要素間の関係が下位レベルに正しく引き継がれなければならない．たとえば，システム全体の入力は最終的な要素のどれかの入力になっていなければならない．

構成要素のまとまりとしての強さを**構成要素の強度**，構成要素間の関係の強さを構成要素間の**結合度**と呼ぶと，分割の一般的な指針は，**強度を強く結合度を弱く**するという“複合設計”[12)]の指針が適用できる．

2.1.3 トップダウン設計

トプダウン設計[25]では，各構成要素は一入口の手続きで，要素間の関係は手続き呼び出しを使う．この設計法では，まず主手続き要素を選び，その内部処理を副手続きの呼び出しを含む形で設計する．次に各副手続きについて同様の設計を進めていく．

設計対象構成要素の全体がおよそ見とおせて，いくつかの機能手続きの組み合わせで表現できる場合，この方法は効果がある．しかし，これを安易に進めると，上位レベルの手続きは理解しやすいが，下位の手続きは細切れの機能に対応するものになり，それぞれ単独では理解しにくくなる傾向がある．このため，次に述べるデータ抽象化技法も念頭におくようにすると良いであろう．

2.1.4 データ抽象化設計

トップダウン設計を容易に進めると，下位モジュールが理解しにくく複雑なインタフェースになりがちであるが，データ抽象化ではこのようなことが生じないように，データとそれに関連する，互いに関係の強いいくつかの機能（手続き）を一つのモジュールにまとめる．そうすることによって，単独では理解しにくかった機能が，一つの抽象機械や抽象データに対する一連の操作として理解できるようになる．実現上の詳細な情報を隠して，そのモジュールを使う側にとって必要な機能だけをみせる方式を**情報隠蔽**[16]の法則と呼び，Parnas の考え方の中心概念をなしている．

抽象データの概念を拡張して，同じ構成の抽象データを複数個宣言して使えるようにしたものが**抽象データ型**[9]である．抽象データ型の概念は，データの実体（オブジェクト）にその操作（メソド）が付随しているという，オブジェクト指向型の概念の芽と考えられる[24]．

2.1.5 データフロー設計

データ抽象化はよいモジュール化を得る上で非常に有効であるが，一般には，どれとどれをモジュールの入口操作にするかの選択が難しい．一方，**順次データ列**（順アクセスファイル，キーボード入力，ラインプリンタ出力など）を扱う場面では，構成要素間の制御関係を無視し，要素間を流れるデータに着目すると全体の見とおしがよくなることが多い．

データフローで扱われるデータは順次データ列とし，一つの順次データ列にデータを書き込む処理（生成側プロセス）と読み出す処理（消費側プロセス）は，それぞれ高々一つに限るものとする．要素間の制御関係は，データは読まれる前に作られることという簡単な規則だけで自動的に定まってくる．これは，メッセージバッファを介した並列プロセス（concurrent processes）に制限を加えたものと考えることができる[8]．このようなデータフローの概念は，システム階層の上位レベル（システム→サブシステム分割）だけでなく，下位の詳細レベルにも活用できる．

抽象データから抽象データ型への拡張と同様に，データフローのプロセスに関しても，同様の機能をもつプロセスを複数個宣言して使えるように拡張できる．たとえば図書館システムを考える場合，本の利用や返却を行う“利用者”というプロセス型をもつプロセスが図書館の利用者の数だけ存在して，並列に動作するようなモデルを設定できる．これは，Jackson のモデリング技法[8]の中心的な概念であり，前に述べた抽象データ型の概念と同様に，オブジェクト指向型の概念に近いものである．

データフロー設計法の大きな問題点は，通常のプログラミング言語ではそのままの形で効率よく実現できないことである．最も自然な実現は，各処理を並列プロセスと考え，中間データ列をプロセス間通信のメッセージバッファとする方式である[20]．データフローで設計を行った後，それをプログラムで実現する際に，手続き呼び出しを用いた制御関係に変換する方式として，Myers の複合設計[12]と Jackson のプログラムインバージョンがある．Myers の方式は簡単ではあるが，限られた場合でしかうまくいかない．

2.2 応 用

ソフトウェア設計のさまざまな側面を要約すると次のようになる：

- ソフトウェア設計は，仕様を基に設計考察をし，さらに詳細な仕様を作成しながら進む（**段階的詳細化**）．
- その設計プロセスは，階層に従ってシステムの各構成要素の特質および構成要素間のインタフェースを反映していく．
- 機能以外の要求についても，設計のすべてのレベルで取り扱われなければならない．
- 設計意志決定内容は，かならずしも完全に形式的言語／記法で表現されるとは限らない．
- ソフトウェアライフサイクルは，入れ子サイクルの集合である．

以下に，実時間制御システムとデータ処理システムの二つの場合について，システム分解の具体的方法を

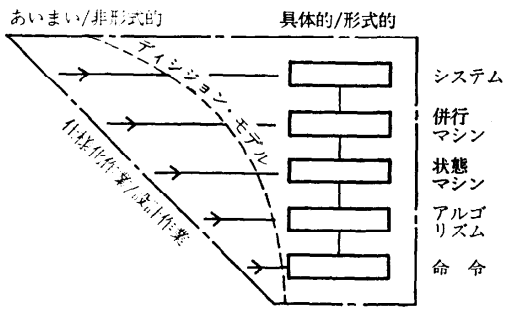


図-3 実時間システムの設計の進め方

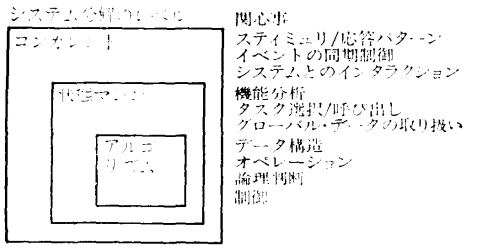


図-4 実時間システムの分解例

示す.

2.2.1 実時間システムの分解

(1) 機能設計のためのモデル

設計者が問題の解を見つける仕方は，純粋に**演繹的プロセス**を経てはいない．ファジィで非形式的解から完璧で形式的な解へ，順次進んでいくのが通例である(**図-3** 参照)．設計の上位レベルにおいては，より非形式的であり，下位レベルにいくにつれ，より形式性を増す．

(2) 機能以外の要求の取り扱い

設計時には，時間／空間性能，実装，信頼性などについても考察される．より下位レベルの設計向けの仕様を導くためディシジョンがくだされる．それら内容は，ディシジョン・モデルに組み込まれる．

(3) 実時間システム向きの階層

実時間システムに適用される階層レベルと各レベルでの主要関心事を**図-4**に示す．実時間システムを特徴づけている属性（ポテンシャルな属性）に，同期性／非同期性，決定論的／非決定論的，対話的／非対話的などがあり，これら属性は，ソフトウェア構造を解明する手掛りを与えてくれる．実時間システムは，次の4つのレベルに分け得る．

●**コンカレント・レベル**……システムは，コミュニケーティング・シークエンシャル・プロセスの集合とみなせる．各プロセスは，自治体的であり，ほかのプロセスまたは環境と非同期的にインタラクトする．

●**シークエンシャル・ステート・マシンのレベル**……有限状態マシンであり，イベント／メッセージによって起動されるアクションを決める．

●**アルゴリズム・レベル**……個々のアクションまたはプログラムで，入力データ構造を出力データ構造へ変換する．

●**命令レベル**……基本的なプリミティブで，各データ要素やそれに対するオペレーションを表している．

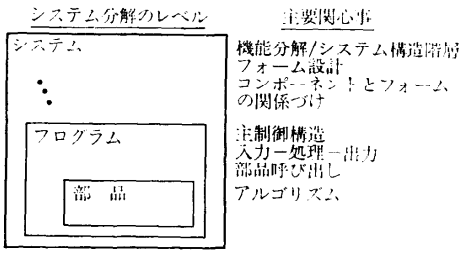


図-5 データ処理型システムの分解例

2.2.2 データ処理システムの分解

一般に，ビジネス・システムのようなデータ処理型システムの分解は，前述の実時間制御型システムのそれと，特質の捉え方が同等ではない．機能分解を行いながら，ファイル処理との関連を明確にする．システム全体は，いくつかの業務遂行上の作業単位に分けられる．さらに作業単位は，プログラミングの単位へ分解される．プログラムの単位は，主制御構造と細部処理手続きとから構成される．これらの分解のレベルと各レベルの主要関心事を，**図-5** に示す．

2.2.3 設計の段階的洗練化

前述の大きな設計階層レベルに共通した設計の進め方のモデルについて述べる．各レベルの中は，通常さらに詳細ないくつもの段階に分けて設計が行われよう．そうした詳細な段階を踏んで設計は洗練化される(**図-6** 参照)．

設計の中核は，ディシジョン・モデルである．これは，提示された要件を満足すべく，問題の分解 (decomposing) を行ったものである．分解されたものは，より低い段階に位する部分となる．構造，ふるまい，信頼性，性能などについて形式的モデルが作られ，要件を満たしているかどうかの検証が行われる．要件を満たしていなければ再考されねばならないし，与えられた要件が成立しない（たとえば，性能とふるまいに関する要件をともに満たし得ないような）場合は，前

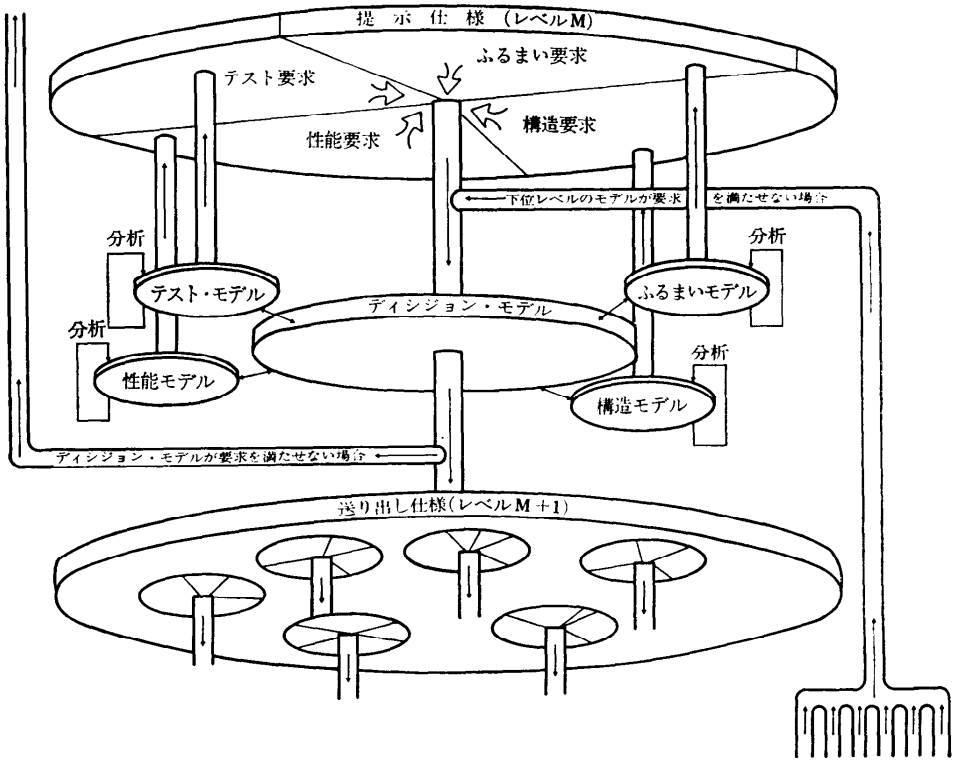


図-6 段階的洗練化

の段階へ戻らなければならない．逆に，解が良好であれば，次の段階への仕様を形づくる．

3. モジュール内の設計法

モジュール分割の設計を終わるとシステムはすでにマネージ可能なモジュール（およそのプログラム規模は50～200行くらい）の集まりに分割されている．システムの全体構造，モジュール間の関係，各モジュールの機能やパラメータなど，ソフトウェアの設計に関する多くの重要な決定はすでに終わっている．

モジュール内の設計では，モジュールの機能やインタフェースを理解し，その実現の処理方式を定めていく．このレベルでの設計法としては，制御構造に着目した構造化プログラミング，データ構造に着目したJackson法，Warnier法，処理をいくつかの機能断片に分け，それらを結合していくPAM（問題分析手法）[2]，状態遷移図を使った手法，場合分けを表形式にした決定表を使った方法などがある．ここでは，代表的なものとして，Jackson法とプログラム構造仕様法について説明する．

（1） Jackson法

構造化プログラミングでは基本制御構造を規定したが，それをどのように選んで処理を組み立てるかについてはほとんど述べていない．Jackson法では，そのモジュールが扱う入出力データの構造をもとに処理の構造を定める方法を提案している．入力と出力の間に，明確な対応関係がない場合は，両方のデータ構造と対応関係をもつような新たな中間データ列を導入し，その中間データ列を介したデータフロー関係のモジュール分割を行う．このように，入力と出力の間に対応関係がつけられない場合を，**構造クラッシュ**と呼び，それをデータフロー関係で解決する方式がJackson法の一つの特徴である．

（2） プログラム構造設計法

データ処理ソフトウェアの開発熟練者は，システム階層，ファイル入出力を含むフォーム仕様，作業単位仕様が確立していれば，プログラム単位への機能要求を基に，処理の展開を考える．ここで，展開を円滑に進めるために，プログラム単位において行われるべき処理手続きの骨格（プログラムの主制御構造）をまず

決め，次に詳細機能を割り付けるアプローチがとられる．プログラムを木にたとえれば，骨格は幹に，詳細機能は枝葉に相当する．これら仕様を**プログラム構造仕様**と呼ぶ．通常，枝葉に相当する機能は，あらかじめ用意されているコーディングがあれば，それを検索して呼び出すようにしている．プログラム構造仕様は，後述するソフトウェア CAD に支えられて，作成，修正，審査を行うことが容易になっている．また，プログラム単位に関係する他の仕様（インタフェース仕様，使用するフォーム自体の仕様など）とともに，実行可能言語レベルへ変換可能であることが大きな利点となっている．

4. ソフトウェア CAD

これまで述べたように，設計技法がいくつか提案されてきた．それらを理解した上で使用すればそれなりの効果は得られるが，良質の設計をさらに手際良く行う，設計審査を容易化する，修正を容易化する，当事者の意志の疎通を容易化する，以降のインプリメンテーションを自動化する目的で，設計のコンピュータによる支援（ソフトウェア CAD）が，重要な位置を占めるようになった．設計技法の多くは，ソフトウェア CAD 化されている．ソフトウェア CAD は，設計対象の**モデル化技法**，**設計記述言語**，**設計支援ツール**，設計に必要な**情報ベース**より成り立っている．また，ソフトウェア CAD では通常，設計内容を図形化して扱える．図形は数十，数百語で表現されるものに匹敵する．ここでは，代表的なものとして，状態に基づくものと，機能階層とフォーム使用関係に基づくものの二つを紹介する．

4.1 状態に基づくソフトウェア CAD

複雑性をもった組み込みシステムの設計のためのソフトウェア CAD について述べる．組み込みシステムは，**反応型**であり，その**動的ふるまい**は，サブシステム間およびシステムとそれを取り巻く環境の間のインタラクションに基づいている．反応型のシステムは，通常，イベント，アクション，信号，条件，入力，出力の組み合わせで成立している．反応型のシステムの

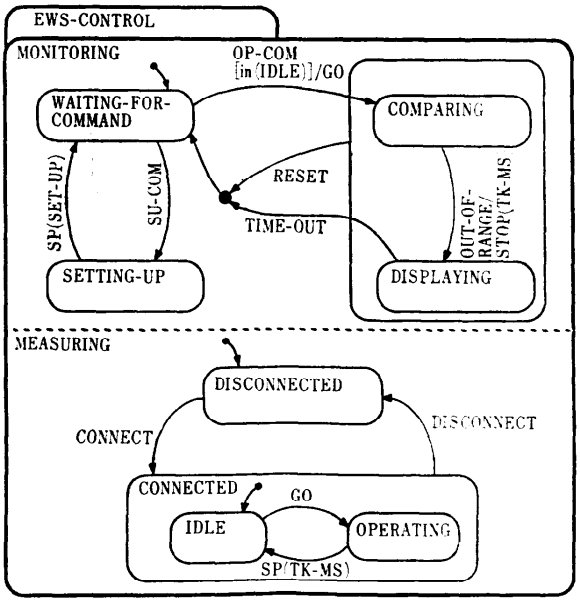


図-7 状態図（簡単な例）

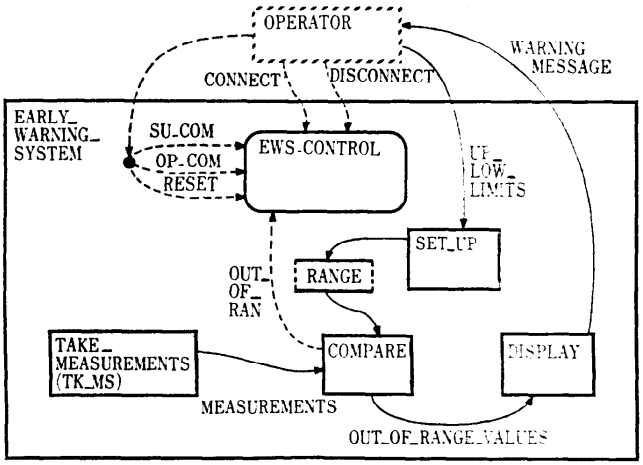


図-8 アクティビティ図（簡単な例）

仕様を明晰，理解容易，正確に記述するため，対象システムを「**ふるまい**」，「**機能**」，「**構造**」の三つのビューで捉える．記述言語は図形的であるとともに形式的であり，コンピュータを使った分析，シミュレーションを通じて，設計結果を評価できるようにしている．

（1） 対象システムの三つのビュー

ふるまいのビュー：これは，システムのふるまいの状態およびある状態から他の状態へ遷移させる条件と

イベントを示す．状態のレベルは，階層を有し，高位のものから低位のものまである．表現形式としては，図-7 に示すような状態図が用いられる．

機能のビュー：これは，システムの機能やアクティビティを示す．また，機能やアクティビティ間を流れる（あるいは，システムと環境との間に介在する）信号やデータも示す．表現形式としては，図-8 に示すようなアクティビティ図が用いられる．

構造のビュー：これは，システムの実際のモジュール，コンポネント，サブシステムといった構成要素およびチャネルを介した相互接続を示す．表現形式としては，図-9 のようなモジュール図が用いられる．

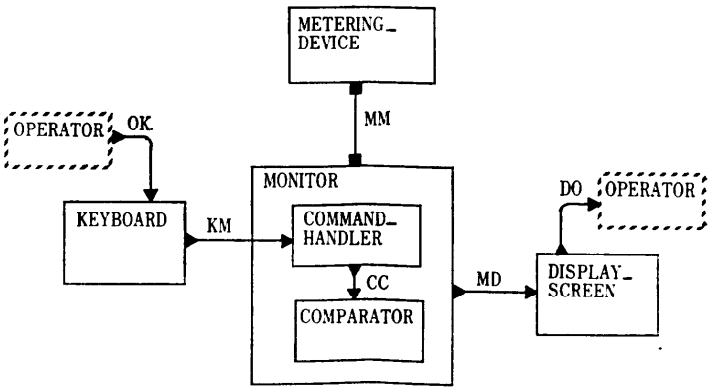


図-9 モジュール図（簡単な例）

（2） ビュー間の関連

ふるまいと機能の二つのビューは，互いに補間関係にある．つまり，状態に関係するイベントや条件は，アクティビティに影響を与えるし，アクティビティの開始または終了は，状態に影響を与える．主要なアクティビティは，いくつかのサブアクティビティへの分解が行われ，それらの間の関連づけをする．状態図に示された制御は，システム内のアクティビティとデータの流れを支配し，信号が状態図側にフィードバックされることを想定している．構造のビューは，より具体的コンポネントとそれらによりシステムがいかに構成されるかを示している．

（3） 状 態 図

状態図 (statechart) は，David Harel によって開発されたもので，従来の状態・イベントのパラダイムを出発点としているが，いくつか拡張が試みられている[6]．従来の状態遷移図は，平面的であり，構造性を有していない．状態図は，これら性質を保持しつつも，大規模で複雑な反応型システムの設計にも向くよう，

- 状態によるふるまいを階層的に指定できるようにした (Superstate, Substate).
- 独立性の高い状態コンポネントを識別することにより，ふるまいをモジュール化できるようにした．
- 現在のシステムのふるまいを時間的に過去のそれに基づけるようにした（履歴，時制論理を使用）．

などの拡張をしたものである．

（4） ソフトウェア CAD としての特長

状態図，アクティビティ図，モジュール図などの状態ベースの言語とシミュレータ，エディタ，レポート生成などのツールを用いて，設計対象システムをモデル化し，動的なふるまいを，種々の角度から目視確認できるようにしている点が，この CAD の特長といえる．

4.2 機能階層とフォーム使用関係に基づくソフトウェア CAD

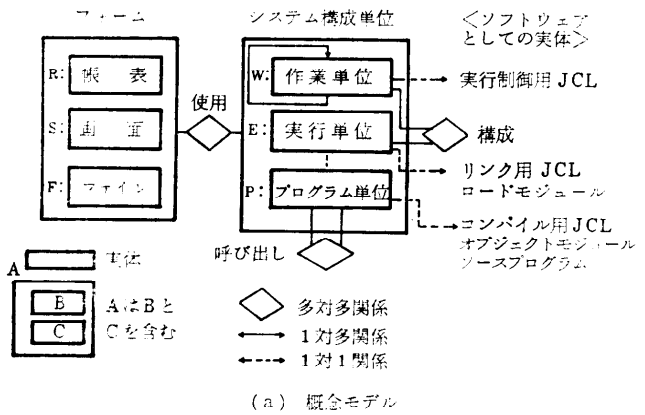
（1） 設計対象システムのモデル化

今日開発されているデータ処理システムは，機能役割面からの階層化に加えて，入出力ファイル，対話画面などのフォームを，どの構成要素が使用するかの関係づけが設計の主要な位置を占めている．**実体—関係** (entity-relationship) **概念**に基づいて，システム構造モデルを扱うようにすれば，設計，保守，転用をするのに都合が良い．図-10（a）は概念上のモデルを，（b）はその実際の構造モデルを示している．

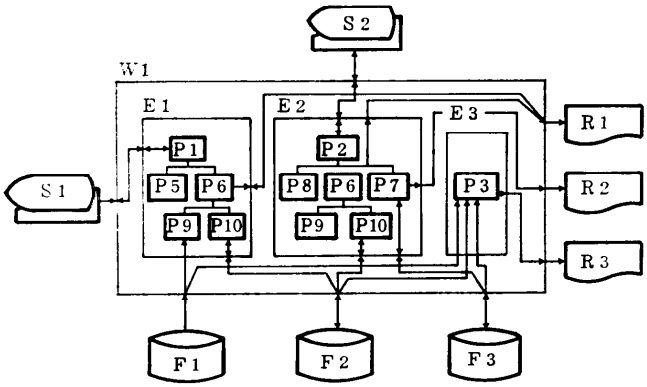
（2） 設計仕様中心のパラダイム

システム構造仕様，フォーム仕様，プログラム構造仕様によって，設計の主要点を適確に捉えることができるので，次の利点がある．

- 設計内容が，より**視覚化**，**標準化**されることにより，良質の設計が行いやすくなるのみならず，審査や修正も容易となる．
- ソフトウェア CAD を使って得た設計仕様から，プログラムが自動生成されるので，インプリメンテーションの**生産性**が上がる．
- 設計とインプリメンテーションが直接つながっているので，設計の妥当性確認のため実行してみることや，変更，保守をプログラム言語レベルでなく，設計



(a) 概念モデル



(b) 実際の構造モデル

図-10 システム構造の設計

仕様レベルで行える.

●類似システムの開発に際し，前に作った設計要素を再利用でき，なお一層開発が容易化される

このように，ソフトウェア開発パラダイムを従来のソース・コード上での保守・改良型から設計仕様を基軸としたスタイルへ移行させる原動力になっている．この例として，ライフサイクル支援システム SEA/I[10] ほかがある.

5. 自 動 化

ここまでは，自動化とは，入力されたものを，抽象化レベルの異なるものに（質的）変換するシステムがあり，人間がこれまで手作業で行っていたことを機械で代行できるようにすることを意味する．自動化は，仕様からのプログラム生成，ソース・コードなどからのプログラム理解用文書生成，実装設計仕様からのジョブ制御言語生成，要求仕様・各種設計仕様からのテスト・データを含むテスト環境の生成など多岐にわたっている．ソフトウェアの品質や作業の生産性を向上させる手段として，自動化はますます重要視されよう．本章では主にプログラミングの自動化について述べる.

5.1 自動化の範囲

時代とともに，自動プログラミングの意味する範囲が次の二つの点で変わってきている.

●垂直方向のレンジ…抽象化レベルのどこからのものを入力（仕様）としてアクセプトできるか

●水平方向のレンジ…いかなる問題領域に対応できるか

従来のソフトウェア・ジェネレータは，財務管理システムとか販売管理システムといった個々の問題領域にかぎって，ジェネレーション制御言語を設け，その指定に基づいて，あらかじめ用意してあるプログラム(群)を特化するものであった.

コンパイラの能力を拡張し，自動化の主として垂直方向のレンジを上げる努力もはらわれた．さらに，自動化の程度を向上させるため，いくつかの開発が行われた．主なものは，次のようである.

●自然言語によるプログラム仕様からのプログラム生成…自然言語で書かれたプログラム詳細仕様を解釈し，その解釈に基づいてプログラム生成を意図している.

●ソフトウェア仕様からのプログラム生成…前章までに述べたデータ構造（フォーム）仕様，プログラム構造仕様，インタフェース仕様に基づいて部品合成を通じてプログラミングの自動化を行う．この方式は，現在実用に具されている.

5.2 問題解決のアプローチ

完全自動プログラミングは，要求定義以降を単一のステップで解決しようとするには，あまりに大き過ぎる問題である．この問題を攻略するため，いくつかのアプローチが試みられてきている．入力言語の観点からすると，一つは，「仕様からのコード自動生成」に関する研究であり，事例は数多くある．特筆すべき研究は，“超高級言語 (Very High Level Languages)”である[21]．また，“超高級言語をサポートするシステ

ム”に関する研究もある[15]．

ほかのアプローチは，「自由度の高い自然言語から形式的仕様への変換」に関するものである[2]．この関連で要求定義における知識表現についての研究も行われた[5]．

対応領域の設定観点からは大きく二通りある．一つは，**'Vertical Slice'** アプローチと呼ばれるもので，十分狭い領域を設定し，完全自動プログラミング・システムを構築しようとするものである．このアプローチでは，特殊目的問題指向言語を設計し，それをコンパイルする特殊目的プログラム・ジェネレータを用意している．

もう一つのアプローチは，対応領域を広く設定するかわり処理を数段階に分け，**部分的自動化** (partial automation) を提供するものである．要求分析，インプリメンテーション，テストといったいろいろなタスクに対して自動化の程度は異なる．Designer/Verifiier's Assistant[11]，Programmer's Apprentice[22]，Knowledge-Based Software Assistant[7] などが，その例である．本アプローチの魅力は，いろいろな程度の自動化の段階的進歩をただちに享受できる点である．

5.3 定理証明テクニック

定理の証明を自動的に発見する（または，発見するのを助ける）テクニックは，ソフトウェア工学や AI の研究の頭初からのアクティブな領域であった．プログラム言語が，Floyd ほかによって，数学的基礎を与えられるや，プログラミング作業に定理自動証明テクニックを応用しようとする試みが始まった．**演繹的合成法**と**プログラム検証法**の2大方法によって進展してきた．

演繹的プログラム合成は，構築的（構成的ともいう）証明はプログラムと等価であるという観察に基づいている．**構築的定理証明系** (Constructive Theorem Prover) は，仕様からプログラムを導出するのに，次のような手順を踏む：導出されるプログラムは，入力 x を取り，出力 y を作り出すと仮定する．precondition $P(x)$ が入力に関して成立し，postcondition $Q(x,y)$ が出力に関して成立すべきであると記述される．この仕様は，次の定理に変換される．

$$\forall x \exists y [P(x) \rightarrow Q(x,y)]$$

この定理は，**定理証明系**に与えられる．証明のプロセスは，条件 $P(x)$ が真になるような x に対して $Q(x,y)$ が真であるような y が存在することを証明し，**証明の副産物**としてプログラムが作り出される．このアプローチは，自動プログラミングという難しい問題を，ほかの難しい問題つまり定理自動証明に置換しているともみれるだろう．演繹的合成は，いろいろな例題に対して，機能することが示されている[23]．しかし，現状は，論理型言語で簡単に形式化しうるような仕様に対して小規模なプログラムを生成する程度に止まっている．次第に判明してきていることは，検証も合成もプログラム開発プロセスの中で，いくつかの部分問題に適用されるべきだという点である．たとえば，プログラマの生産性向上のために，できるだけ早期段階で，プログラム設計の個々の性質を検証しようとするのは理にかなっていよう．

5.4 トランスフォーメーショナル・アプローチ

自動プログラミング研究のなかで最も活発な領域は，プログラム構築を支援する目的の**トランスフォーメーション**に関するものである．プログラム・トランスフォーメーションとは，プログラムの部分を取り上げ，新たな部分（トランスフォームされた部分）と置換する操作を示す．プログラム・トランスフォーメーションの重要な特長は，Correctness-preserving（**正しさの保持**）である．新しい部分は，旧部分の働きと全く同じである．トランスフォーメーションの目的は，部分に着目し，それを改善することである（たとえば，より効率を高めるなど）．X**2 を計算するのにベキ乗のサブルーチンを使うより，X*X を計算する手続きに置換するなどは，簡単な実例である．

通常，プログラム・トランスフォーメーションは三つの要素を有している：

① トランスフォーメーションをプログラムのどの部分に適用するかを決定するのに使うパターンを有している．

② 適用条件の集合を有す．この条件集合によって，トランスフォーメーション適用箇所をさらにせばめる．いままで開発されたトランスフォーメーション・システムの多くは，適用条件を論理言語で表現し，所与の状況下で適用条件が満足されるか否かを決定するために定理証明を使っている．

③ アクションを有す．アクションは，通常，手続き型である．パターン・マッチングがとれた箇所を置換すべくプログラムの新しい部分を創出するのがアクションである．

5.5 仕 様 技 術

自動プログラミングの主要課題は，プログラム記述

のために，いかなる言語を使うようにするかである．自然言語，超高級言語，ふるまいの例示の三つがユーザ・インタフェースとして，研究上着目されている．

究極的には，制限のない自然言語でユーザに入力させるようにすることであろう．Protosystem-I[13] ほかは，この方向での初期の研究である．初期の研究成果から学んだことは，自然言語による仕様表現の本質的困難さは，自然言語自身がもっているシンタックスやセマンティックスについてではなく，意志疎通をはかるときに人が使う **informality** についてである．

本質的問題は，たとえば，

> the RATS transmission times are entered into the schedule.

といった場合，それは

> Each RATS clock transmission time and transmission length is made a component of a new transmission entry which is entered into the transmission schedule.

の意味に解釈できるかである[1]．

以上の判明結果を元に，自然言語による仕様の研究は，自然言語処理の課題から informality 問題を切り離す方向に進んでいる．自動プログラミング研究の焦点は，自然言語よりも，もっと新たな形式言語（シンタクティック処理はストレートフォワードだが，セマンティック informality を許せるような言語）の設計へと移った．

早い時期に開発された超高級言語の一つに SETL があり，サポートしているのは Algol ふう言語の標準コンストラクトの大部分である．さらに，便利なユニバーサル・データ構造—tuple と set—をサポートしている．たとえば，写像は 2-tuples の set として扱う．また全称と存在記号を使えるようにしている．集合 S のすべての要素 x に対してなんらかの計算を行うための SETL ステートメントは，

$(\forall x \in S)$……end ∀；

と書く．このような表現手段を提供する目標は，プログラマがデータ構造の詳細設計について考慮しなくても済むようにしようとする点である．

5.6 要求仕様からの自動プログラミング・システム

要求仕様を入力として受け入れる自動プログラミング・システムで汎用のものは，現在ない．特殊用途自動プログラミング・システムではあるが，興味深いもの二つをあげるとすれば，Draco システム[14]とΦNIX システム[3]がある．入力は要求仕様に近いものであり，ドメイン知識を内蔵している点が特徴である．

Draco システムは，いわばプログラム・ジェネレータ・ジェネレータである．このシステムは，トランスフォーメーショナル・フレームワークを有しており，そのフレームワークの中で特殊目的プログラム・ジェネレータを定義できる仕組みになっている．あるドメインに対するプログラム・ジェネレータを定義する仕方は，“ドメインに精通している設計者”が，

① まず，ドメインのプログラムを記述するのに好都合な**問題向き言語**を設計する．

② 次に，その問題向き言語で記述されるプログラムを，どうジェネレートするかを指定する．

Draco システムの貢献は，プログラム・ジェネレーション過程を**宣言的に指定**することを可能にしたことである．

6. おわりに

設計技法，ソフトウェア CAD およびインプリメンテーションの自動化について主なものをできるだけ紹介してきたが，紙数の制約で割愛せざるをえなかった点も少なくない．ソフトウェアは，きわめて広範な分野にわたっていて，その特質も一様ではない．したがって，そうしたソフトウェアの設計・実現パラダイムや支援システムは，実践レベルではソフトウェア分野ごとに最適化されよう．技法の進歩のみならず，ソフトウェア CAD も実務上でさらに真価を発揮できなければならない．視覚的要素を含む操作性の改良のみならず，設計作業を支援する機能（性能解析，視聴覚に訴える動的ふるまい確認など）を進歩させることが課題である．

設計・自動化技術は，今までの基盤の上に，特に仕様言語とそれに関連したトランスフォーメーションや知識獲得の面において，さらに発展することが期待されよう．

謝辞 本稿の執筆にあたり，日本電気(株)ソフトウェア生産技術研究所 紫合治氏，北川博之氏ほか多くの方々から貴重なコメントなどをいただいた．ここに感謝する次第です．

参考文献

1) Balzer, Robert M., Neil Goldman and David Wile: Informality in Program Specifications, IEEE Trans. on Software Eng. SE-4(2), pp.

94-103 (1978).
2) 二村良彦：問題分析法 PAM によるプログラムの設計と審査，情報処理学会論文誌，Vol. 23, No. 4, pp. 451-458 (1982).
3) Barstow, David R.: A Perspective on Automatic Programming, AI Magazine (Spring), pp. 5-27 (1984).
4) Booch, G.: Object-Oriented Development, IEEE Trans. on Software Eng. SE-12 (2), pp. 211-221 (1986).
5) Borgida, Alexander, Sol Greenspan and John Mylopoulos: Knowledge Representation as the Basis for Requirements Specifications, IEEE Computer (April), pp. 82-90 (1985).
6) Harel, D.: Statecharts: A Visual Formalism for Complex Systems, Weizmann Institute Technical Reports CS 84-05, CS 86-02 (Mar. 1986).
7) Green, C., Luckham, D., Balzer, R., Cheatham T. and Rich, C.: Report on a Knowledge-Based Software Assistant, Rome Air Development Center, Technical Report RADC-TR-83-195, pp. 1-50 (1983).
8) Jackson, M. A.: System Development, Prentice-Hall (1983).
9) Liskov, B. H. and Zilles, S. N.: Programming with Abstract Data Types, SIGPLAN Notices, Vol. 9, No. 4, pp. 50-59 (1974).
10) Matsumoto, M.: The SEA/I-Software Productivity System, International Conference on Software Development, Methodologies, Tools and Alternatives (July 1983).
11) Mcriconi Mark S.: A Designer/Verifier's Assistant, IEEE Trans. on Software Eng. SE-5 (4), pp. 387-401 (1979).
12) Myers, G. J.: Reliable Software through Composite Design, Petrocelli/Charter, 1975. 邦訳（久保未沙，国友義久），高信頼性ソフトウェアー複合設計，近代科学社 (1976).
13) Ruth, Gregory R.: Protosystem I: An Automatic Programming System Prototype, AFIPS Conference Proceedings, National Computer Conference, Anaheim, CA, pp. 675-681 (1978)
14) Neighbors, James M.: The Draco Approach to Constructing Software from Reusable Components, IEEE Trans. on Software Eng. SE-10 (5), pp. 564-574 (1984).
15) Kant, Elaine: On the Efficient Synthesis of Efficient Programs, Artificial Intelligence 20, pp. 253-306 (1983).
16) Parnas, D. L.: Information Distribution Aspects of Design Methodology, Tech. Report, Dept. Computer Science, Carnegie-Mellon Univ. (1971).
17) Parnas, D. L.: On the Criteria to be Used in Decomposing Systems into Modules, CACM, Vol. 15, No. 12, pp. 1053-1058 (1972).
18) Parnas, D. L.: Designing Software for Ease of Extension and Contraction, IEEE Trans. on SE, SE-5, No. 2, pp. 128-138 (1979).
19) Ross, D. T.: Structured Analysis (SA): A Language for Communicating Ideas, IEEE Trans. on SE, SE-3, No. 1, pp. 16-34 (1977).
20) 紫合 治，岩元莞二，藤林信也：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, pp. 528-538 (1980).
21) Schonberg, E., Jacob T. Schwartz and Micha Sharir: An Automatic Technique for Selection of Data Representations in SETL Programs, ACM Trans. on Prog. Lang. and Sys. 3 (2), pp. 126-143 (1981).
22) Waters, Richard C.: The Programmer's Apprentice: A Session with KBEmacs, IEEE Trans. on Software Eng. SE-11 (11), pp. 1296-1320 (1985).
23) Manna, Z. and Waldinger, R.: A Deductive Approach to Program Synthesis, ACM Trans. on Prog. Lang. and Sys. 2(1), pp. 90-121 (1980).
24) 米澤明憲：オブジェクト指向型プログラミングについて，コンピュータソフトウェア，Vol. 1, No. 1, pp. 29-41 (1984).
25) Yourdon, E.: Top-down Design and Testing in Yourdon, E.: Managing the Structured Techniques, Yourdon Press, pp. 58-87 (1979).

（昭和 62 年 6 月 10 日受付）