解　説

情報処理システムのモジュール化†

前　川　　守††

1. はじめに

本解説は情報処理システムをモジュール化の観点からモデル化する試みである．ここで対象としている情報処理システムはソフトウェアシステムおよびソフトウェア，ハードウェアを共に含むシステムである．

いわゆるソフトウェア工学，さらには情報科学の分野においての根本的問題の1つは情報処理システムのモデルを明確にすることである．モデル構築にはいくつかの方法が考えられるが，情報処理システムの論理の複雑さ等を考慮するとモジュール化による構造モデル構築は欠かせないものであるし，おそらく最も重要なモデル化の方法であろう．システムをモジュール化するという概念はもちろん情報処理システムに限ることなくほぼ総ての種類のシステムに適用されている極めて普遍的な概念である．情報処理システムのモジュール化においては，これら他システムに適用されている概念，方法論，ツールを最大限利用すると共に，情報処理システム特有の部分については過去の成果をできるだけ活用すると共に不充分な部分については新たな研究が必要となる．本解説は過去の成果を構造化，またはモジュール化の観点から横断的に眺め整理するのが目的である．

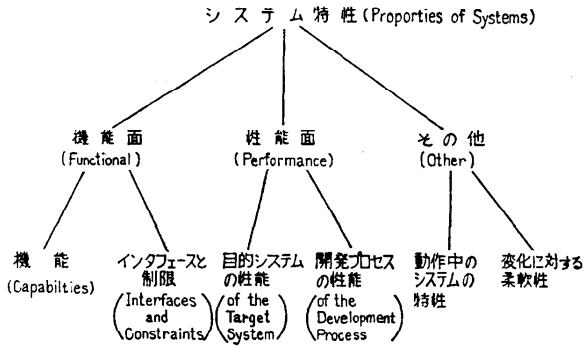


図-1　システム特性の分類

† On Modularazation of Information Processing Systems by Mamoru MAEKAWA (Faculty of Science, University of Tokyo).

†† 東京大学理学部情報科学科

システムをモジュール化する目的としては幾つも考えられるが，システムの特性を明確にすることがまず第1に考えられる．これは要求定義段階においても設計の段階においても共に重要である．Teichroew[34]はシステムの特性として図-1のような分類をしている．システムの特性は機能面，性能面，その他，の3つに分類される．機能面はシステムが果たす機能とシステムの外部との接続様式を規定するインタフェースおよびその制限にさらに分類される．性能面はシステム自身の性能とシステムを開発する過程（プロセス）の性能に分類される．その他の特性としては動作中のシステムの特性，たとえば信頼性，回復性等と，システム環境の変化に対する対応性，たとえば，柔軟性 (flexibility)，適応性 (adaptability)，保守性 (maintainability) 等があげられている．このシステム特性としてさらに追加されるべき項目としては，動作中の性能として，最適性（動的にシステムを最適化する能力），変化に対する対応性としては拡張性（動的にではなく長期間の変化に対してシステムを最適化しうる能力）をあげたい．さらには変化に対する対応性としてはシステム開発をシステムの群 (family) に対して行いうる能力（たとえば Parnas の program families の概念[25]）は極めて重要である．これをもう一歩進めてそのようなことが自動的に行いうる能力も重要である．そのようなシステムとして国井は生命型製品または evolutionary architecture を提唱している[16,17]．

Ross は特性として理解度 (understandability)，変更の容易さ (modifiability) をさらにあげている[28]．このうち，変更の容易さは歴史的に見て最も達成しにくい目標であることがわかる．その理由はシステムの環境の変化や，システムの一部の変更が及ぼす程度を推測することが技術的に困難なためであり，また発生しうる変更をあらかじめ予測することが難しいためである．変更の容易さを得るためには適応性が高く，漸進的設計，標準ブロックの設定，性能の

チューニング機能等が必要である．

モジュール化を行う別の理由としては人間的側面があげられる．それはシステム開発者がシステムをよりよく理解するための手段であり，また設計者間の情報伝達をよくするための手段でもある．さらにはシステム開発が管理可能となるためにはモジュール分割は不可欠である．本解説ではこの人間的側面についてはごく軽くふれる程度にとどめる．

2. モジュールとは

Liskov が初期の論文で指摘しているようにモジュール分割は従来からも用いられているシステム開発法である[18]．これはあらゆる工学において適用されている概念であるが，情報処理システム，特にソフトウェア開発の分野での成功の度合が必ずしも高くないのはモジュール化を行うという方針が悪いのではなく，正しいモジュール化の方法が定まらないからである．Liskov は従来のモジュール化手法ではモジュール化を行ったがゆえに次のような問題が発生するとしている．

（1） モジュールは互いに関係はあるが異なるあまりにも多くの機能が含む形で作成される傾向にあるためモジュール内の論理が複雑となる（新たな機能上の複雑さの導入）．

（2） 共通的な機能を早期に検出できないために，これらの機能が多くのモジュールに分散され，個々のモジュールの論理が明解でなくなる（機能上の複雑さ）．

（3） モジュールが共通データを扱う際に共通データが予期しない形で扱われることが生じる（結合上の複雑さ）．

第1章で述べたシステム特性が仕様を満たし，開発が円滑に行われるためには，モジュール化は唯一の方法ではあるが，モジュール化が正しく行われない限りそれも意味がない．本章ではまず第1にモジュール識別のための基準を数え上げることから始めよう．

2.1 モジュール識別の基準

モジュール化の基本の第1はまずモジュールを識別することである．モジュールの識別には，そのための基準が設けられる必要がある．モジュール識別または分割の基準としては次の項目が考えられる．

（1） 制御の流れの違いによる分類

この基準は時間の次元でもってモジュールを分類する方法である．時間の次元による分類としては同時性，順次性，排他性，選択性が考えられる．

例 同時性：プロセス，コルーチン，CPU 内の並列処理ユニット．
順次性：コンパイラのパス1，パス2等，プロセデュア．
排他性：モニタ[13]，スケジューラ．
選択性：順次性を有するモジュールで条件分岐が行われる場合（例えばメッセージ処理）．

（2） 機能の違いによる分類

これはモジュールの果たす機能の違いにより分類する方法である．

例 給料計算のモジュール，平方根を計算するモジュール，演算ユニット，ディスク制御ユニット．

（3） データ分割

モジュールの基本としてデータをとらえ，データモジュールによりシステムをモデル化する．データモジュールのインタフェースは通常そのデータに影響を及ぼしたり参照したりすることができる“動作”により記述される．後程述べる Ross によるデータ分割[29]，Liskov の抽象データタイプ[19]，Parnas の情報隔離[25]等が広い意味でこの範ちゅうに入る．

（4） トランザクション分割

基準(2)，(3)は処理する側からの分類であるが処理される側(トランザクション)の立場から分類が可能である．たとえば，トランザクションの種類によりモジュール分割することも可能であるし，トランザクション処理の順序性により分割することも可能である．基準(1)の制御による分類と共通点はあるが異なる．SREM の R-net はこの規準を用いている[1]．

（5） インタフェース最小化

これはモジュール間のインタフェースを最小にするようにモジュールを分割する方式で，モジュールの満すべき条件の第1にあげられることが多い．ここでインタフェースとはモジュール間の結合の複雑さを言い，具体的には結線の数，プロトコールの複雑さ，パラメタの数等により定まる．またこれらの変化の度合いも大きな要素となる．

（6） インタアクション最小化

インタフェースは結合の複雑さを示すが，そこに流れる情報の量，頻度をインタアクションと呼ぶこととし，インタアクションを最小にするようにモジュールを分割する．この基準は分散処理システム等で特に重

要であり，一般にシステム全体の性能に大きく影響を及ぼす．

（7） 位置による分類

モジュールの属している位置により分類するもので，ページ，セグメント，トラック，シリンダ等の他にハードウェア実装上の制限からくるモジュール分けがある．また分散処理システムにおいては物理的位置が重要な要素となる．

（8） 権利による分類

各モジュールの有する参照の権利，またはモジュールが他のモジュールに許す参照許可条件によりモジュールを分割する．

例 読み出しのみのモジュール，スーパバイザのみが参照を許されているモジュール．

（9） 実行頻度による分類

各モジュールの実行の頻度によりモジュールを分類するもので，性能の最適化に重要である．通常のプログラムでは，プログラムの極く一部が実行の大部分を占めることが知られているので，その部分を分割し別扱いすることに意味がある．

以上は個々のモジュールの有すべき性質により分類したものであるが，モジュール間の結合または構造によりモジュールを分割することができる．

(10) 層による分類

情報処理システムを階層構造に分割したとき各層をモジュールとしてとらえる．ここで階層構造とは実体のあるモジュール，すなわち抽象モジュールではないモジュール間の何らかの意味での階層構造を意味する．この階層を定める要素としては制御関係，権利関係 (authorization)，保護関係 (protection)，機能の大小関係，等がある．オペレーティングシステムにおいて，たとえばプロテクション機構，プロセスの創出機構，メモリの各プロセスへの創出機構(virtualization)，入出力の仮想化，のように階層化する場合[6]では保護関係と機能の大小関係が階層を定める大きな要素となっている．

(11) 拡張性その他による分類

システムの望ましい性質としては拡張性，高信頼性，変化に対する柔軟性，動的再構成性，等があげられる．これらを満たすようにモジュールを分類し結合の方式を定めることができれば大変望ましい．これらは目的ではあるが，モジュール分割の基準と考えることもできる．これら上記の性質を満たすべきシステムの条件には共通するところも多く，まとめてまたは個々に多くの研究開発が行われている．分散処理システムの目指すところ，データベースシステムの3層スキーマ構成の目指すところ等がこれら目的と合致する．

以上はモジュール間結合または構造に主眼を置いたモジュール分類基準であったが，モジュール化のもう1つの大きな主眼は設計者の思考過程の記述であり，いわゆる抽象化の概念である．抽象化の過程では上位モジュールは設計者により作られ，設計書として存在するが実体はなく下位モジュールにより置き換えられる．

(12) 抽象化の階層による分類

システムをより抽象化の高いレベルから実体へと分割していくときに各レベルで生じるモジュールを抽象モジュールとして分類する．この抽象モジュールは各レベルにおいてモジュール化基準(1)～(9)を用いて抽出される．このモジュール分割法は，機能を中心として分割する方法とデータを中心に分割する方法の2つに大きく分類することができる．

(13) 階層的機能分割法

これはおそらく最も一般的なモジュール分割法であると思われるが，機能を順次詳細化していくモジュール分割法である．各レベルにおいて他の基準を適用することができる．

(14) 情報隔離 (information hiding)

Parnas の情報隔離の原則は，各モジュールにおいてそれの実現に必要な情報とモジュールの外部に必要な情報を分離する方式を与えるもので変化に強い構造を有することができる[25]．

(15) 抽象データタイプ (abstract data types)

データタイプの概念を拡張し，データのタイプをそれに参照できる作用素で定めると共に実現法は分離して記述できる．これについても多くの研究がある[18]．

(16) 階層的データ分割法

データ中心に階層的に分割を進める方法で，機能分割ではデータがモジュール間を結合するものであったが，データ分割では動作がモジュールを結合するものとなる．

モジュール分割または統合に際してはこれらの基準を必要に応じて組み合せ，できるだけ最適の形でモジュールを定める．しかし，これらの基準の中には互いに相反するものもあり妥協が必要となる．

2.2 モジュールの分割，統合，対応

システムはモジュールから成るという概念は極く普

遍的と思われるが，プログラムの分野では必ずしも全面的に受け入れられているわけではない．たとえば段階的プログラム開発法 (stepwise refinement, では基本的姿勢はあくまでも全体プログラムを“いきなり”プログラムするという姿勢である．違いはプログラムを書くプログラム言語がより高級であり，したがってプログラムされた結果が人間が直接見て理解できる範囲におさまるということにすぎない．この方法は従来の“プログラミング”の概念から一歩も出ず，ただプログラミングを容易にすることを考えているに過ぎない．そこにはシステムの構造をとらえる姿勢とモジュールの概念が欠如している．これでは成果物の品質は依然プログラマの質に大きく依存し，システムの特性を評価することも難しい．しかも1章であげたような要求を満たすとはとうてい考えられない．これらの要求を満たすにはモジュールを明確にすることが重要である．Parnas は段階的プログラム開発法に対してそれを補うものとしてモジュールの概念をより明確に出している[25]．これは Parnas によれば Dijkstra[7]の発展であるとしている．上記の段階的プログラム開発法も Dijkstra[7]から発展している．Parnas でのモジュールは外部から認識できる動作を記述した“仕様 (specifica tions)”であると定義されている，モジュールを構成するにあたってはプログラムの群 (family) を作成するのに必要な，すなわちこれら群に共通でない設計上の決定事項を隔離するように行われる．これは情報隔離の原則によっている．この Parnas の方法は後に述べるモジュール分割 (module decomposition) の一方法である．段階的開発法の他の欠点は逐次処理を最初から仮定し開発を進めていることである．したがってこの方法は極く限られた分野においてのみ効果を発揮しうるであろう．

システム開発にはモジュール化が不可欠であるとの立場に立てば，システム開発の方法として，より大きな(抽象的な)モジュールから小さなモジュールにと分割していくモジュール分割法 (module decomposition)，その逆であるモジュール統合法 (module synthesis)，そして階層を考えないモジュール対応法 (module mapping) とが考えられる．

3. システム記述とモジュール

3.1 要求記述とモジュール

システムをモジュールに分割し，それによりシステムをモデル化する方法は本質的にシステムの構造を取り扱うことになるため，システムの内部構造（たとえそれが実際の実現の方法とは異なったとしても）を例示することになる．このことはモジュール化によるモデルでは要求のすべてを記述しえないことを意味している．また場合によってはモジュール化記述が望ましくない場合もある．コンパイラ等はおそらくその典型例であって，コンパイラの仕様記述には言語仕様を主体とするのが望ましく，モジュール化モデルはコンパイラの他の側面，たとえば性能を規定するのに用いられる．

このモジュール化によるシステムモデルがシステムの内部構造を例示するということは別の困難さを生じさせる．それは例示されたシステムは要求に対する十分条件の1つを通常示すにすぎず，必要条件を必ずしも示していないことである．たとえば，あるモジュールが記述されているとするとき，そのモジュールの果たす機能の必然性は必ずしも保証されない．他のより本質的な問題は例示が最適である保証が無いことであるが，これは設計の問題として残される．

このようにモジュール化によるモデルではシステムの必要最小条件を記述することは困難であり，これは何らかの別の方法で記述される必要がある．これは自然言語，図，数式等により示されることとなろう．要求記述にはこれら記述をもサポートする道具立てが必要である．

3.2 システム理解とモジュール化

モジュール化が議論されるときの大きな要素は人間にとっての理解の容易さである．そもそもシステムをモジュールに分割する動機は，大きな問題を小さな理解しやすい問題に分割するためである．この理解度増進のためのモジュール化規準として重要なのがインタフェース最小化および物理的位置と他基準，たとえば制御とか機能により分割された構造との一致である．さらに重要なのはインタフェースとモジュール本体の分離である．このためにはHarada と Kunii によるように記述法としてインタフェースを分離する必要がある[11),12)]．

3.3 システム解析とモジュール化

第1章で述べたシステム特性のいずれもその解析には構造が重要となる．機能の解析に構造が重要となるのは実際に広く認識されており，多くのシステム記述がこの方法で行われていることでもわかる．性能の解析はやはり構造モデルが基本となり，その解析はモジュール分割，統合と同じくシステムのサブシステムへ

の分割，それらの統合が基本的解析法である[4),30)]．信頼性についてもやはりその解析は構造を扱うことにより行われる．柔軟性，拡張性，適応性，保守性等いずれをとっても構造モデルが大きな役目を果すことは明らかである．これらのことがモジュール化によるシステムモデルが情報科学（工学）の根本的理解を進める基本であり，現実の諸問題を解決する原動力となると考える根拠である．このように考えるときソフトウェア工学を含めての情報科学，情報工学発展の特効薬が突如見つかるということはなく地道な努力が絶えまなく行われる必要があることがわかる．

4. モデルの多次元性

システムのモデルを構築するのにはいくつもの理由があるが，通常その各々に対して異なったモデルを作成する必要がある．さらには，1つの目的に対してもそれを達成するためには幾つものモデルが必要になることが多い．しかし，現在のところ2.1節で述べた各々のモジュール化基準に対しても満足なモジュール記述の方法が無く，ましてや，それらの組合せに対しては無力に等しい．この複合モジュール化基準の研究は極めて重要であるにも拘らず研究が行われていない重要な分野である．

2.1節のモジュール化基準(1)～(9)はいずれも相矛盾する基準ではなく，実際のモジュール化にあたってはどれを選び，どのような組合せでモジュール化を行うかが重大な設計上の決定事項となる．例えば，SADTでは機能分割とデータ分割を互いに相補うものとして両方が行われることを推奨している[29)]．抽象データタイプはプロセデュア等の機能，制御抽象化と相補うものとしてとらえられる．

インタフェースの複雑さに対するモジュール化の影響としては機能分割とデータ分割に対して，ソフトウェアの規模が小さいうちは機能分割が望ましく，規模が大きくなるとデータ分割が有利との観察がある[3)]．これについてはより実証が必要であるが極めて重要な観察結果と言える．

プログラムの性能の最適化（tuning）に関しての機能分割とデータ分割の関係については実験の報告がある[21)]．この実験結果によると最適化に関しては両者共に同程度の成果を与えることができるが最適化の対象となるプログラムの部分が互いに相反することが報告されている．

5. モジュール分割の一般原則

システムの設計において最も知恵を搾る必要があるところはモジュールの分割である．種々の言語や支援システムはあくまでも補助であり，一旦モジュール分割が定まればそれを記述し整理するのに役立つがそれらは直接モジュール分割を行ってくれるものではない．モジュールの分割という作業は最も創造的で困難な作業である．現在までに提案されている種々の方法論や言語のうちモジュール化の指針を示しているのは極めて少ない．その理由は明白で，モジュール分割の一般則を作ることが極めて困難なためである．実際すべての情報処理システムに適用可能で，かつ有効性の高い原則を見い出すことは不可能と思われる．一般的なものは効果が少なく，効果の大きいものは汎用性に欠けるであろう．本章では過去に提唱されている原則のうち，比較的適用性の高いと思われる原則について述べることとする．

Rossは一般に適用されるべき原則としてモジュール化（modularity ここでのモジュール化は本解説でのモジュール化より限定された意味で用いられている），抽象化（abstraction），局所化（localization），隔離（hiding），一様性（uniformity），完全性（completeness），確認可能性（confirmability）の7原則を上げている[28)]．Rossはシステムは階層的に分割が行われて開発される（top-down）という前提に立っている．

まずモジュール化（modularization）については，Rossはいくつかの定義を与えることから議論を始めている．Dennis, Goos, Myersの定義はそれぞれモジュール化の目的による定義で，Dennisはモジュール化の目的は，個々のモジュールの正当性をその使用の条件（context）とは無関係に証明できることであるとし，Goosはモジュール性とは個々のモジュールをその内部構造を知ることなくより大きなシステムを構築できる能力としている[5),9)]．Myersはモジュール化とは単に大きなシステムを小さく分割することではなく，各モジュールが互いに極めて独立であるように分割することであるとしている[23)]．構造面からモジュール化が定義されることも多く，Liskovは別個にコンパイル可能で，かつ他モジュールと結合されるものとしている[18)]．Rossはモジュール化とは特定の目的を達成するのを助けるためのシステムの構造であり，モジュール化とは目的に応じた構造化であると定義している．Liskovの狭い定義を除いてはこれら定義はすべ

て正しいが Ross の定義が一番広くかつ本解説での趣旨と合致する．モジュール化の定義は上記のように目的と関連してが行われることが多いが，これら目的を達成するためにモジュール化を推進する一般的方法はシステムの構造に何らかの制約を設けることである．簡単な例は構造化プログラミングで強く提唱された GOTO-less プログラミングである．GOTO 文を禁止することにより各文の実行の条件をより明確にしプログラムの理解を助けることとなる．

抽象化 (abstraction) の概念はモジュール化の概念（上に述べた狭い意味での）と同様重要な原則である．抽象化の原理は不要な詳細は無視し，真に必要な基本情報のみを抽出することである．階層構造のモジュール化は抽象化の原型であると言える．もし抽象化の原則が完全性の原則と併用されれば各レベルはシステムを完全に記述することとなり，各レベルでシステムの完全理解が可能になると共に，各レベルでの構造を他システムに移転することも容易となる．いずれにしても抽象化はモジュール化と併用される必要がある．

局所化は物理的距離を問題とするものである．サブルーチン，アレイ，レコード，ページ等が局所化の例である．さきほどの GOTO 文の排除は制御構造を局所化する方法である．

隔離は Parnas 提唱の情報隔離で知られている[24]．これはトップダウン方式において物の対応 (binding) の決定をできるだけ遅らせることおよび抽象化と関連しており，隔離原則では抽象化および対応の決定を遅らせることを行うのみならずそれを明確に強制化する．これにより“いかに (how)”の部分が抑制され不要な詳細が表われることが抑えられる．

一様性 (uniformity) とは不統一や矛盾を無くすことである．この保証は困難であるが対象を絞って（たとえば形式的記述に対してのみ）適用することが可能であるし，他の原則との併用も重要である．

完全性 (completeness) は明らかに必要な原則である．この原則は記述（モデル）が必要なことはすべて明確に述べ，不必要なことは一切排除されていることを要求するものである．この原則もやはり証明が困難である．

最後に確認可能性 (confirmability) は要求された目標が達成されるか否かを確認できることを要求するものである．これも難しい条件であるが確認可能性はすべての条件が明確に (implicit でなく) 述べられていることが前提となる．

6. システム開発行程の形式モデル

システム開発の方式としてはトップダウン (top-down)，ボトムアップ (bottom-up)，段階的開発法 (stepwise refinement)，内外開発法 (inside-out)，外内開発法(outside-in)，重要部分第1開発法 (mostcritical-component-first) 等がある[8]．これらの定義はいずれも概念的であり厳密ではない．このいずれの方法においてもそこで行われる人間の知的活動として Freeman は活性化 (operationalization)，抽象化 (abstraction)，詳細化 (elaboration)，検定 (verification)，意思決定 (decision-making) の5つをあげている[8]．活性化とはあいまいな条件が述べられたときにそれを具体的な条件文に置き換えることを意味する．たとえば“システムは迅速に応答しうること”という条件に対しては，“システムの応答時間は端末80個稼動時で平均1秒以内，最高3秒を上廻らないこと”のようにテストの可能な条件に置き換えることを意味する．抽象化については既に述べたので明らかであると思われる．詳細化はほぼ抽象化の逆である．検定はシステムまたは各モジュールが要求を満たしているか否かのテストである．意思決定は設計の中心であって種種の可能性から1つを選択する行為である．

これらの定義はいずれも極めてあいまいで非形式的である．このシステム開発行程を少しでも自動化するためには形式性を導入する必要がある．システム開発行程の形式性を追求したモデルは極めて少なく，わずかに Harada と Kunii の形式化ぐらいである[11]．Harada と Kunii は設計プロセスを階層的分割としてとらえ，その過程を表現するのに便利な表現法として再帰グラフ (Recursive Graph) を定義し行程のモデルの基本としている．再帰グラフはグラフのノードにグラフを含むことを許すグラフであって階層的分割法の表現に適している．各レベルでの設計結果はデザインスキーマ (design schema) として定義されグラフで表現される．全体の設計は再帰グラフの集合として表わされる．デザインスキーマから別のデザインスキーマに移るのは再帰グラフの操作として表わされる．

デザインスキーマはモジュールの階層構造を表現するが，そのためモジュールを示すためのノードとモジュール間の関係を示すための連係 (association) とが基本となる．連係は対等の関係を示すもの，包含関係を示すものの他にモジュールとポートの関係を示すものとがある．ポートはモジュールのインタフェースを

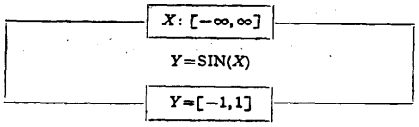


図-2 ポート

とるためのサブモジュールであって外部はこのポートを通じてのみモジュールにアクセスできる．例えば図-2 に示すモジュール（sin 関数）は入力ポートとして X，出力ポートとして Y を許している．デザインスキーマには更に各モジュールおよび連係の記述を行うためのレコードが対応づけられている．

再帰グラフの操作のためには 8 個の基本命令が用意されておりグラフの種々の操作が行える．図-3 にこれら操作の例を示す．Harada と Kunii はこれを更に発展させて機械でチェックを行い種々の解析結果を出力するためのシステムとその言語を提案している．

この Harada と Kunii による形式化は種々の機能分割設計法の記述を機械でチェック可能なレベルまで形式化すると共にインタフェースを分離し階層構造法に合った操作命令を設けたものと言えることができる．この方式は現在なお発展途上にあり拡張が計画されている[12]．

7. システムおよびモジュール記述言語

7.1 モジュール化と記述言語

システムおよびモジュールを記述するにはそのための記述言語が必要となる．記述言語はモデルとの結束度合により極めて一般的なものから特定のモデルに限定されたものまである．最も一般的な記述言語はおそらく自然言語であり，モデルの存在を一切仮定していない．自然言語はあらゆる種類のシステムに適用できるかわりにモデルがないため記述は冗長，不正確になる．一方，モデルが確定している記述言語は自由度が極めて限られる代償として記述は簡潔で正確になる．この種の言語としてはパラメタ指定の各種の言語，たとえばオフィスコンピュータ用の簡易言語等，がある．

この両極端の言語の間にモデルとの結合度合に応じていくつもの言語が考えられる．まず第一の群は第 2 章で述べたモジュール化基準のいずれかに着目して言語を定める方式で，主なものとしては機能に着目した言語，データに着目した言語，トランザクションに着目した言語，性能評価のためのサーバ中心の言語，制御中心の言語，正当性証明のための記述言語等がある．またこれら以前の言語として概念設定のために用いられる言語がある．これらは構造を議論する以前に用い

表記法 / グラフ要素関数	グラフ表現	式表現	備考
ノード関数	R1: 1:buffer	$N_{R_1}=\{1\}$ $nsR_1(1)=$buffer	buffer システム
副ノード関数	R2: 1:buffer, 2:sender, 3:receiver	$N_{R_2}=\{1,2,3\}$ $snR_2(1)=\{2,3\}$ $snR_2(2)=snR_2(3)=\phi$	buffer は sender と receiver を有する
ポート関数	R3: 1:buffer, 2:sender, 4:syn, 3:receiver, 5:syn	$N_{R_3}=\{1,2,3,4,5\}$ $ptR_3(2)=\{4\}$ $ptR_3(3)=\{5\}$ $ptR_3(1)=ptR_3(4)=ptR_3(5)=\phi$	sender と receiver は syn をポートとして有する
アーク関数	R4: 1:buffer, 2:sender, 4:syn, 1:synchronize, 3:receiver, 5:syn	$A_{R_4}=\{1\}$ $asR_4(1)=$synchronize $afR_4(1)=[4;\ 5]$	連係 synchronize が receiver の syn から sender の synにある
副アーク関数	R5: 1:buffer, 2:sender, 4:syn, 6:in, 7:out, 1:synchronize, 2, 3, 3:receiver, 5:syn, 8:in, 9:out	$A_{R_5}=\{1,2,3\}$ $saR_5(1)=\{2,3\}$ $saR_5(2)=saR_5(3)=\phi$ $afR_5(2)=[9;\ 6]$ $afR_5(3)=[7;\ 8]$	連係 synchronze は互いに逆方向の副連係から成る

図-3 再帰グラフの操作

られる言語で関係図や表が主たる手段となる.

上記のモジュール化基準はシステムの特定のもの,または特性に着目するため,それに基づく記述方式がより適しているシステムとそうでないシステムが生じる.しかし,これら記述方式の適用範囲は広範であり汎用言語と呼ばれるべきものである.

次のレベルはモジュール化基準を複数個組み合せた記述言語である.このレベルの記述言語の考え方としては1つの言語で複数基準に適合するように定める方式と1つの記述から他の記述を導く考え方とがある.このレベルの言語も種々の努力が成されているがまだ十分な成果は得られていない.

さらにモデルとの関係を深めるためにはシステムの種類(アプリケーション)を限定するか,解析対象であるシステム特性を限定するかである.このレベルは本解説では扱わない.

以上の議論を要約すると**図-4** のようになる. 全体システムの記述や概念設定レベルの記述は現在のところ自然言語や特定の形式にとらわれない表や図が圧倒的に用いられている.これを形式化するには人工知能の分野で研究されているセマンティックネット(semantic nets)[31]やファジーシステム(fuzzysystem)[39]の適用が考えられる.情報処理システム記述に対してのこれらの適用例は極めて少いなが試みる価値のある分野と考えられている.

情報処理システムの特性記述として Ramamoorthy と Ho は次のことが最低行われなければならないとしている[26].

(1) 情報処理システムとその外界(環境)との間で作用するすべての実体を明確にすること.

(2) システム入力に対して期待される機能的な応答を明らかにすること.

一般性	特性
より一般	
自然言語	モデルなし,極めて一般
概念設定言語	関係,表等により記述,概念の設定,整理
1個のモジュール化規準による言語	特定のモジュール化規準を基本とする記述
複数個のモジュール化規準による言語	複数のモジュール化規準のすべて,またはそれらの関係を記述する
アプリケーション別またはシステム特性別言語	システムの種類,特性等に応じて多くの言語,モデルがある
パラメトリック言語	モデル固定,パラメータ値の変更のみ
より限定	

図-4 言語の汎用性/専用性

(3) システムの性能に関して,性能を定めるパラメタの明確化とそれらパラメタによるシステム性能の記述.

(4) システムの他の特性(たとえば柔軟性)をシステムの各要素によって記述すること.

このうち(2)～(4)は第1章で述べたシステムの3つの特性に対応しており,(1)はそれ以前にシステムを形成するまたはシステムに関係する"物(entity)"を明確にすることである.物とそれらの関係を記述する言語としては二項関係を記述する言語[35]が適している.この言語が形式化されていれば PSL のように機械処理が可能となる.この章では概念設定の次のレベルの言語,すなわち,何らかの意味で構造を有する言語と記述方式について詳しく眺めていくことにする.

7.2 機能中心のモジュール化と記述法

情報処理システムの記述で最も広く用いられている方法は,まとまった機能を1つのモジュールとし,それらモジュール間の構造を記述する方式である.ここで機能とはシステム(またはモジュール)の成す行動によって記述するか,または入力と出力の関係(関数)が定まるならばその関数を明示することによって示される.すなわち何らかの行動または操作である.

機能中心の記述として知られている方法には SADT[29], HIPO[32], PSL/PSA[35] 構造化設計[33], Harada と Kunii の形式化[11),12)]等がある.記述としてはいずれもシステムを階層構造に記述するが,形式性,モジュール定義の正確性,階層構造定義の正確性等に違いがある.これらいずれの方法においてもモジュールの決定,分割,統合は設計者,解析者の仕事となっており,その結果を記述する方法にすぎない.もちろん,記述方式にはいくつかの制限,指針がありモジュール分割が正しい方向で行われるよう作用するが絶対的なものではない.

さて,まず1個のモジュールに対する記述法について考えることにしよう. 1個のモジュールの特性を定める記述能力として次のような能力が必要とされる[27].

(1) モジュールの外部から見たときの動作またはモジュールの果たす機能の記述.

(2) モジュールの外部との関係(インタフェース)を記述する能力.

(3) モジュール内部の記述.これはモジュール実現の際に必要となる情報である.

(4) 射影(projection)の機能.これはモジュール

の特定の側面をうきぼりにするための記述能力である．

（5） 非手続的な記述．内部のアルゴリズムの指定なしに記述できる能力．

（6） 非規定的な記述．モジュール特性の記述を非規定的に述べる能力．ここで非規定的とはモジュール実現の機構，アルゴリズム，戦術等が自由に選べることを意味する．

（7） 複数の視点から見た記述が可能であること．

（8） モジュールの内部構造に直交する形の記述が可能であること．直交する記述はモジュールの構成要素間の関係を規定するのに役立つ．

（9） 記述自身がモジュール構造であること．このことによりモジュール特性が独立に指定され，モジュール特性自身とモジュール特性間の関係が別個に指定される．

(10) 分析が可能な記述であること．すなわち，与えられた記述からシステムの種々の特性を解析，評価できること．

これに加えて次の項目も必要と思われる．

(11) 実現機構の記述．実現機構の記述は内部記述で他と分離されているべきである．

(12) タイプ（またはクラス）とそのインスタンス（またはオカランス）を指定する能力．この能力は記述を簡単化するのに役立つ．

(13) 記述の形式性．これは記述を機械で扱うのに必要な条件である．

(14) 記述の理解の容易さ．記述は設計者が見て理解しやすいものでないと要求記述，設計の段階での情報伝達の言語としては望ましくない．

(15) モジュール記述システムの操作性．これはモジュールの直接の特性とは言えないかもしれないが，モジュール記述を操作するには大きな要素である．

モデルが階層構造から成ると考えるときには階層構造の表現に対する記述能力も必要となる．

(16) モジュールおよびインタフェースの明確な包含関係が明示されること．

(17) 機能と実現機構の階層構造をそれぞれ分離して表現可能であること．

つぎに，これら個々の条件についてどのような解決法が用意されているかを眺めてみることにしよう．

まず機能の記述についてはその果たす機能が非常に小さいものでない限り自然語で書かれる場合が多い．PSL, SADT, Harada と Kunii の方法，HIPO 等い

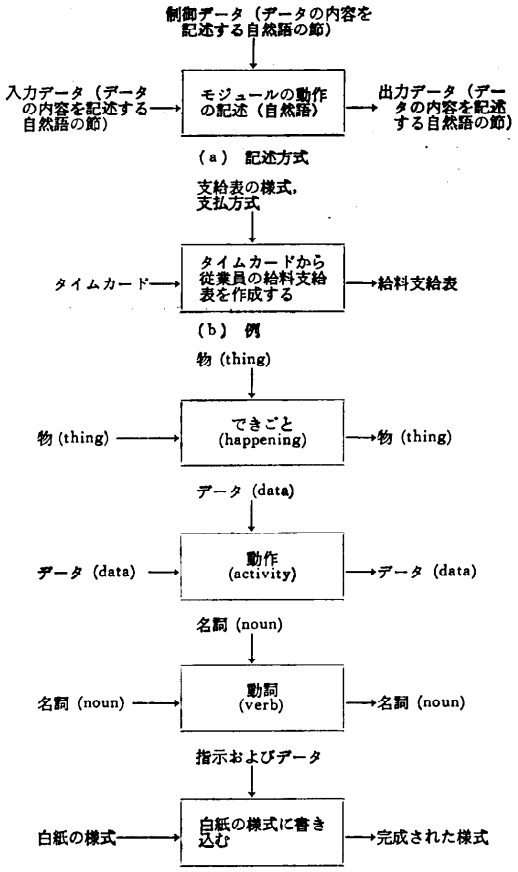


図-5 SADT モジュールの記述

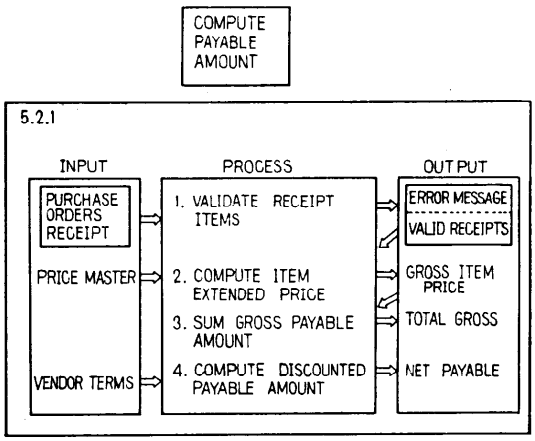


図-6 HIPO モジュールの記述例

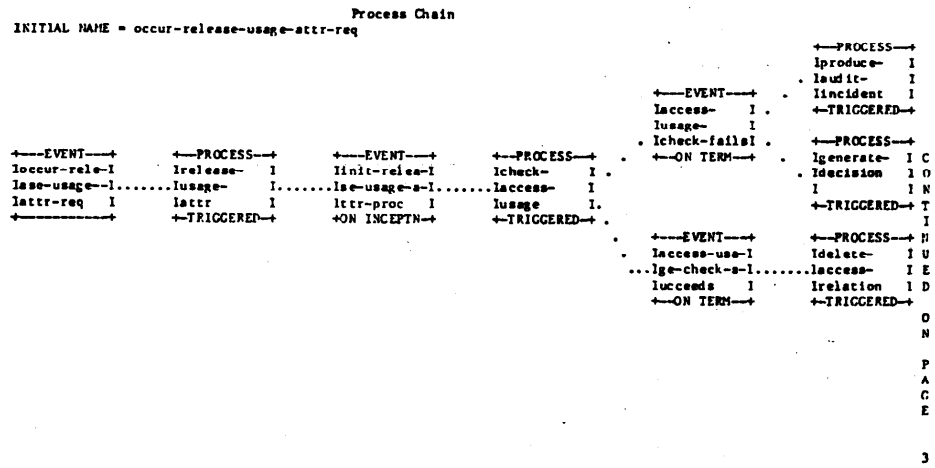
ずれも機能の記述は自然語による．自然語以外の記述は機能が数学的関数で記述可能で，かつ形式的で簡単な場合や手続き型言語で記述が可能な場合に限られる．これは現在の技術水準からしてやむをえないと考えられる．手続き型言語による記述については後ほど述べる．

第2のインタフェースについてはモジュール間でやりとりされるデータにより記述される．PSL, SADT, HIPO 等ではデータの非形式的な記述，すなわち，データの名前，種類等を記述する．データは一般に入力データと出力データに分けられるが SADT では制御データをさらに分類している．PSLはデータの構造を包含関係として記述可能である．システムの変更を考えるとき重要な要素としてインタフェースと

```
Parameters:  DS=-EXEDS  NAME=hourly-employee-processing  NOINDEX  NOPUNCHED-NAMES  PRINT  SHIFT
  NOPUNCH  SMARC=5  WMARC=20  AMARC=10  BMARC=25  RMARC=70  CMARC=1  NMARC=60  MODESIGNATE
  SEVERAL-PER-LINE  DEFINE  COMMENT  NONEW-PAGE  NONEW-LINE  NOALL-STATEMENTS
  COMPLEMENTARY-STATEMENTS  LINE-NUMBERS  PRINTEOF  DLC-COMMENT

 1 PROCESS                                          hourly-employee-processing;
 2      /*  DATE OF LAST CHANGE - JUN 20, 1976, 13:56:44 */
 3     DESCRIPTION;
 4          this process performs those actions needed to interpret
 5          time cards to produce a pay statement for each hourly
 6          employee.;
 7     KEYWORDS:        independent;
 8     ATTRIBUTES ARE:
 9          complexity-level
10                        high;
11     GENERATES:       pay-statement, error-listing,
12                      hourly-employee-report;
13     RECEIVES:        time-card;
14     SUBPARTS ARE:    hourly-paycheck-validation, hourly-emp-update,
15                      h-report-entry-generation,
16                      hourly-paycheck-production;
17     PART OF:         payroll-processing;
18     DERIVES:         pay-statement
19       USING:         time-card, hourly-employee-record;
20     DERIVES:         hourly-employee-report
21       USING:         time-card, hourly-employee-record;
22     DERIVES:         error-listing
23       USING:         time-card, hourly-employee-record;
24     PROCEDURE;
25          1. compute gross pay from time card data.
26          2. compute tax from gross pay.
27          3. subtract tax from gross pay to obtain net pay.
28          4. update hourly employee record accordingly.
29          5. update department record accordingly.
30          6. generate paycheck.
31         note: if status code specifies that the employee did not work
32            this week, no processing will be done for this employee.;
33     HAPPENS:
34          number-of-payments TIMES-PER pay-period;
35     TRIGGERED BY:  hourly-emp-processing-event;
36     TERMINATION-CAUSES:
37                    new-employee-processing-event;
38     SECURITY IS:   company-only;
39
40 EOF  EOF  EOF  EOF  EOF
```

(a) PSL によるプロセスの記述例



(b) プロセス関係図の例

図-7 PSL によるモジュール（プロセス）記述例

モジュール本体の切り離しがあり，インタフェースを異なったモジュールとして認識する方式もある．HaradaとKuniiの方式ではインタフェースとモジュール本体を異なったモジュールとして認識している．これらモジュールの記述例および記述方式を図-5～7に示した．またHaradaとKuniiによる記述例は既に図-3に示した．この4種の言語の中PSLとHaradaとKuniiによる言語とが機械処理に適した(または目指した)言語である．SADTも機械処理が行われつつあるが，HIPOに対してはそのような動きはないようである．PSLは記述は図-5に示すように文章によるが表示は図で示すことができ，これら言語がいずれも図式言語によるのは興味深い．これは(14)の記述の理解の容易さに関連することで，このレベルのシステム記述には欠かせない要素と言えるかもしれない．

さて，つぎにモジュールの内部記述について見てみることにしよう．これは(11)の機構の記述と関連する．内部記述は3つの方式があると考えられる．1つはモジュールの果たす機能，役割のみを記述する方式でモジュール内には内部記述は一切行わない方式である．この方法ではモジュールが詳細化されることにより，その詳細モデルが内部記述を直接または間接に表わすこととなる．第2の方式はモジュール内に直接内部記述（実現方式）を記述する方式でプログラム言語またはその拡張でシステム記述が行われる場合，たとえば抽象データ型を用いての記述，または更に非形式的ではあるがHIPO等の方式に見られる．第3は機構を区別する方式である．SADTとHaradaとKuniiによる方法がこの第3の方式に属する．PSLは第1の方式であり，HIPOは原則的に第1の方式であるが第2の方式の併用も可である．第3の機構の記述を許す方式は設計段階における必要モジュールの検出，ボトムアップ設計に有用である．

条件(4)の射影の能力は(9)の記述自身がモジュール構造であることと関連するが，現在の記述方式ではこの条件は十分に満足されているとは言い難い．これを実現するにはインタフェース，機構を分離したように必要な側面ごとにある程度のモジュール分割が必要であろう．

条件(5)および(6)の非手続き的，非規定的な記述というのはプログラミング言語によらない記述ではおおむね満足されている．しかし，これらを厳密に保持するための，特にチェック可能な規則を作成するのは困難でガイドラインの規定程度に現在のところとどまっている．

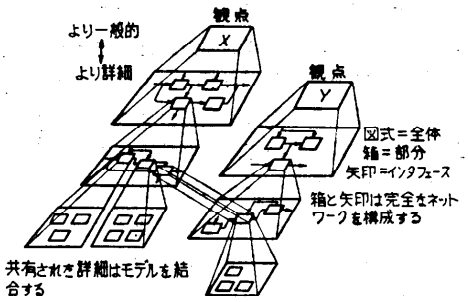


図-8 SADTの複数の観点からのモデル化

条件(7)の複数の視点からの記述はSADTで特に強調されており SADTの特徴の1つとなっている．その基本概念を図-8に示す．他の方式では複数観点からのモデル化は必ずしも強調されていない．

条件(8)の直交する記述というのは形式化または定量化するのが困難な条件であるが，多くの場合この条件は間接的な形で満足されていると考えられる．

条件(10)の分析可能な記述というのは極めて重要な条件であり，システム特性間の関係を記述できることを要求している．これについては紙数の制限もあり本解説では触れないこととする．

条件(12)のタイプとそのインスタンスを指定する能力は条件(11)の機構の記述とも関連するが，設計またはインプリメンテーションを考える際には重要な条件である．機構は個々の実体ごとに定義されるのに対してタイプ（またはクラス）は同種の実体を包含する機構の種類を示す．

条件(13)の形式性は機械処理に特に重要である．PSL, HaradaとKuniiの方法，SADTは機械化に可能な形式性を備えている．これに関連して操作性の面ではこれらいずれも出力は図で示す形を取っているが入力の際に若干問題がある．原則的にはこれらいずれも図を直接操作できる機能を備えることが望ましい．

条件(16)，(17)の階層構造の記述はいずれも包含関係を記述する方式をとっている．階層構造の記述は設計行程の記述でもありHaradaとKuniiによる方法が最も厳密に表現を行っている．

7.3 データ中心のモジュール化と記述法

機能中心のモジュール分割では機能がモジュールを成し，それらを結合するものとしてデータがあった．これに対応する形としてはデータをモジュールとして認識し，データモジュール間を接続するものとして動作を認識する方式がある．一方，データ構造を先に認

識し，それにより処理（機能）の構造を定めていく方式が考えられる．さらに別の方式としては機能とデータを併行してモデル化（モジュール化）していく方式が考えられる．この節では最初の2つの方式について述べ第3の方式については次節で述べることにする．

さて，まず第1にデータ構造を先に認識し，それにより処理の構造を定めていく方式としては Jackson 法，Warnier-Orr 法等がある．Warnier-Orr 法ではデータ構造を**図-9**のようにまず記述する．ここで (1, n) はその項が最低1回，おそらくは多数回出現することを示す．記号⊕は排他的な条件にあることを示す．この図はデータの包含関係の構造を示すわけである．つぎに出力に現れるデータの項目をデータ構造と関連づける（**図-10**）．最後にこれを見ながら処理（プロセス）の構造を定めていく．この場合の全体プログラムの名前を REPORT PROGRAM とすると，そのプログラムにおいて各データ項目に必要とされる項目

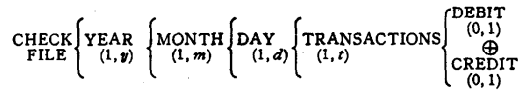


図 9 データの論理構造

を考えることにより**図-11** のような処理の構造を定めることができる．

Jackson 法は同様の方法でモジュール構造を定めていくもので，まずデータ構造を**図-12**のように定める．ここでデータ構造はプログラムの場合と同様に，列 (sequence)，繰返し (iteration)，選択 (selection) の3つの基本構成要素で示される．それらに対する Jackson の表記法を**図-13**に示す．図-12のようにデータ構造が定まると，個々のデータに対して処理が大体1対1に対応するので処理の構造を**図-14** のように定めることができる．

さて，これら Jackson 法，Warnier-Orr 法はソフトウェアのモジュールの考え方として大きな指針とな

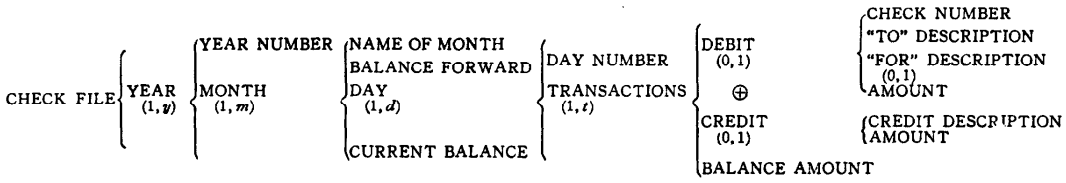


図-10 出力に現れるデータ項目とデータ構造との関連

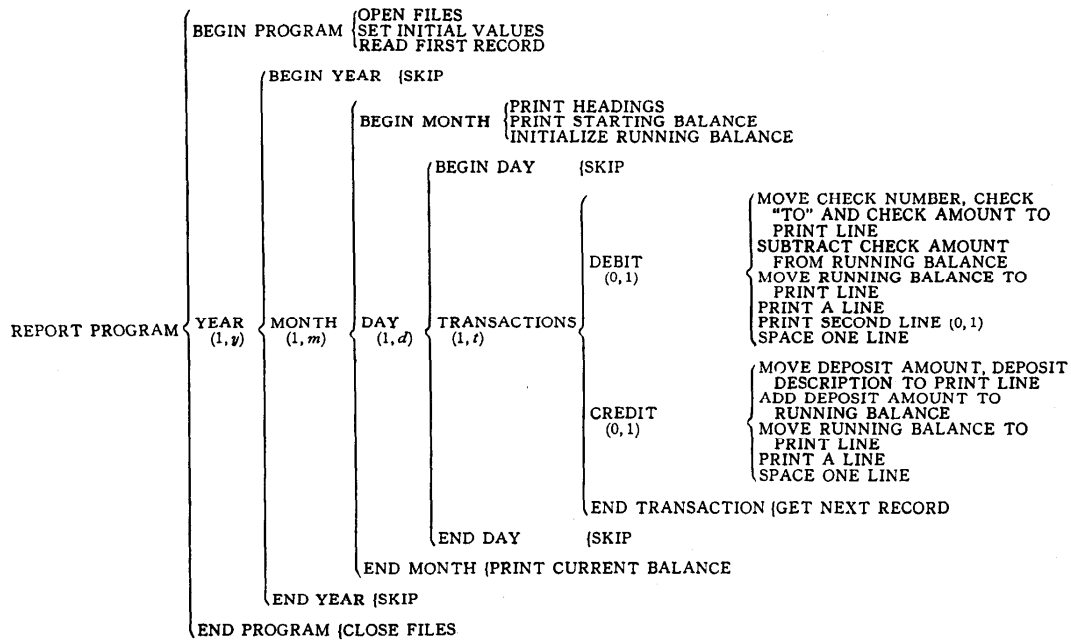


図-11 処理の構造

るものであるが，そこに提案されている表記法は極めて原始的なものであり，7.2 節で上げたモジュール記述の満たすべき条件という点から見ると不満足な点が多い．これらについては改善または他記述法の併用が考えられる．

データ中心の記述法の第2の方法としては，データ中心のモジュールに対して動作はモジュールを結合するものであるととらえる考え方がある．SADT におけるデータダイアグラムがその例で，記述法としては同じ SADT の動作ダイアグラム (activity diagram) の丁度裏返しである．図-15 に図-5 に対応するモジュールの記述規則を示す．データダイアグラムと動作ダイアグラムは記述法の点で丁度裏返しになっているが書かれるモデルは通常裏返しではなくそれぞれ異なった側面，この場合は当然ながら動作面とデータ面をうきぼりにする．この SADT のデータダイアグラムによるモジュール記述法についての評価は動作ダイアグラムによるのとほぼ同様である．

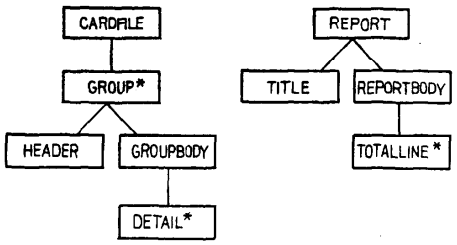


図-12 Jackson 法におけるデータ構造の表現

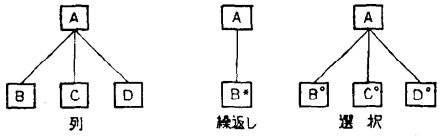


図-13 Jackson のデータ構造の記述法

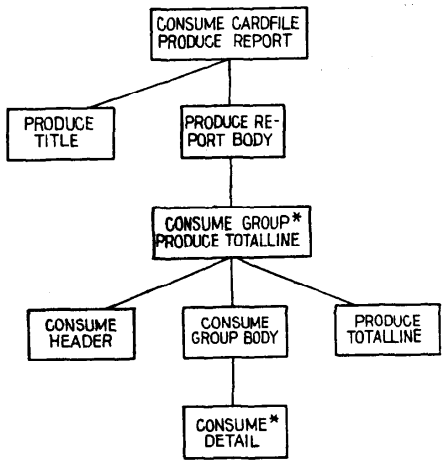


図-14 Jackson 法における処理の構造

7·4 トランザクション分割

トランザクション分割は工場のアセンブリラインのようにトランザクションが順次流れ処理される形態を記述するもので，トランザクションが各ステップまたはモジュールのトリガーとなりトランザクションは別のトランザクションへと変形されて別のステップへ移る．こういった形のモデル化はトランザクションのようにユーザにとって重要なデータについて記述を行い内部のデータとの区別を明確にするため，よりユーザ側に立った視点からモデル化できる利点がある．

トランザクション分割では各ステップは機能により分割されるため，その意味では機能分割である．7.2 節で述べた機能分割と異なる点はモジュール間の関係がトランザクションで定義されることである．トランザクションには工場のアセンブリライン等におけると同様に“流れる”という概念が伴ない，そのためその

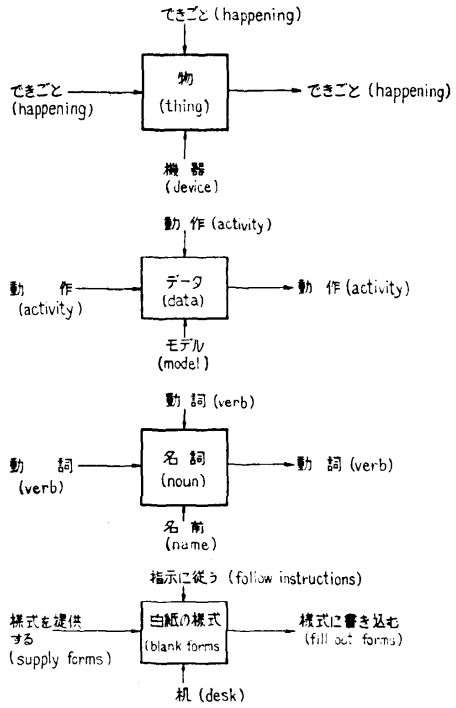


図-15 SADT データモジュールの記述原則

流れの制御が表面に表われることが多い．このような意味でプログラムの制御の流れと一見似た記述となるが，これら両者は制御されている対象が片やトランザクションであり，片や実行の点であるところが大きく異なる．

トランザクション分割の代表例は SREM における RSL (Requirements Specification Language) とそのグラフ表現である R-net (Requirements-net) である．図-16, 17にその例を示す．RSL は PSL に比べて細かい点で多くの改善が成されているが7.2節で述べたようなモジュール記述の評価規準からすれば同傾向の言語である．RSL の特徴はトランザクションを直接扱っているためトランザクションの処理時間が陽に評価しやすいことであろう．

7.5 制御の流れによるモジュール化と記述

制御の流れの違いを表わすモジュール化の単位として最も一般的なのがプロセスである．プロセスをモジュール化単位とする記述法として十分なものは少ないが，Brinch Hansen の分散プロセス (distributed processes) の概念および記述法，Hoare による記述法，Wirth の Modula 等がある．Brinch Hansen の記述法はプロセス間の通信をプロセデュアコール (procedure call) と guarded regions により行うもので表現法としてプロセデュアコールに対しては，call Q.R (expressions, variables)によりプロセスPがプロセスQのプロセデュアRを呼ぶことを許す．Guarded regions は

when $B_1 : S_1 | B_2 : S_2 | \cdots\cdots$ end

cycle $B_1 : S_1 | B_2 : S_2 | \cdots\cdots$ end

の形を取り when 式では条件 $B_1, B_2, \cdots$, のいずれか1つが成り立つまで待ちそれに対応する文が実行される．cycle 文ではそれが繰返される．もし，複数個の条件が成り立っている場合にはいずれか1つが任意に実行される．どれが実行されるかを予知することはできず非確定的な要素が表現される．

Hoare の表現法は互いに関連するプロセスの群を1つのコマンドで表現できる

```
ORIGINATING-REQUIREMENT: SENTENCE-2.
   DESCRIPTION: "DEFINES ANALOG DEVICE MEASUREMENTS".
   TRACES TO: MESSAGE DEVICE-REPORT.
MESSAGE: DEVICE-REPORT.
   PASSED THROUGH: INPUT-INTERFACE FROM DEVICE.
   MADE BY: DATA DEVICE-NUMBER, DATA TYPE-MESSAGE,
            DATA DEVICE-DATA.
   TRACED FROM: SENTENCE-2.
DATA: DEVICE-DATA.
   INCLUDES: DATA PULSE, DATA TEMPERATURE,
             DATA BLOOD-PRESSURE, DATA SKIN-RESISTANCE.
ENTITY-CLASS: PATIENT.
   ASSOCIATES: DATA PATIENT-NUMBER,
               DATA SAFE-FACTOR-RANGE FILE
               FACTOR-HISTORY.
DATA: SAFE-FACTOR-RANGE.
   INCLUDES: DATA LOW-PRESSURE, DATA HI-PRESSURE,
             DATA LOW-TEMPERATURE, DATA
             HI-TEMPERATURE,
             DATA LOW-SKIN-RESISTANCE,
             DATA HI-SKIN-RESISTANCE.
   TRACED FROM: SENTENCE-4.
FILE: FACTOR-HISTORY.
   CONTAINS: DATA MEASUREMENT-TIME, DATA HPULSE,
             DATA HTEMPERATURE, DATA HBLOOD-PRESSURE
             DATA HSKIN-RESISTANCE.
   TRACED FROM: SENTENCE-3.
```

図-16 RSL 記述例

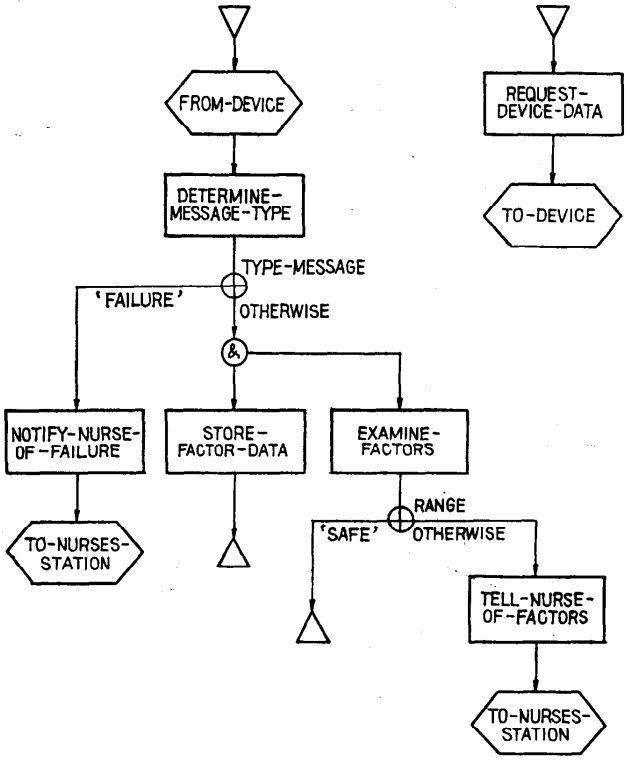


図-17 R-net の例

ようにし，プロセス間の通信を表現するために入力と出力のコマンドを別途設けたものである．たとえば，次の例

[cardreader? cardimage‖lineprinter! lineimage]

は2つのプロセス cardreader と lineprinter が並列に動作し全体は両方共に終了したときに終了することを示す．特殊記号‖はプロセスを区別するのに用いる．この例の前半は入力コマンドの例で特殊記号?によりデータを生成する生成元 (source process) のプロセスとそのデータが格納される変数名を指定する．この例の後半は出力コマンドを表わすもので特殊記号!の後に指定されるデータが!の前に指定された送り先 (destination process) に送られる．

この Brinch Hansen と Hoare の概念および表記法は従来のプログラミングにおける種々の概念，たとえば，プロセデュア，コルーチン，モニタ，セメフォア等を包含した形で表現し扱うことができる点で大きな意味があり，モジュール化の表現法としても将来の発展の基礎になりうると考えられる．

7.6 抽象データタイプ

プログラムにおけるモジュール化の単位としてのプロセデュアの果たす役割は大きく，そのためプロセデュアに対しては多くの研究が成されてきた．データにおいてプロセデュアに対するものとして認識されてきたのが抽象データタイプであり，近年その研究が盛んに行われた．プロセデュアについては多くの研究が行われ，ここでそれらを再録する必要を認めないが，抽象データタイプとの対応として眺めてみるとプロセデュアの有する利点として少なくとも次の8点を上げることができる[14]．

(1) 記述の繰返しを排除できること．
(2) モジュール化
(3) 構造化プログラミングへの基礎
(4) プログラム理解の概念的単位
(5) 明確に定義されたインタフェース
(6) 保守および改善の単位
(7) 言語拡張の機構を与える
(8) 別コンパイルの単位

抽象データタイプはデータに対してこれらと同等の利点を生じさせてくれることが期待されるわけで，記述法もそれを満足する形でなければならない．このためには抽象データタイプ（種々の名前で呼ばれている．たとえば SIMULA の class, CLU の cluster, ALPHARD の form）はその形式として

(1) 名 前
(2) インタフェース指定
(3) 本 体

が与えられなければならない．そして，プロセデュアと同様，抽象データタイプの名前のスコープ (scope) の範囲内で新しいインスタンス (instance) を宣言できる必要がある．この宣言にはパラメタを指定できることが望ましい．インタフェースとしては各インスタンスに対して適用可能な操作 (operations) を指定する．本体は抽象データタイプ実現のために必要な局所変数，インタフェースの操作実現のための局所プロセデュア，初期化のプログラムを含んでいる．抽象データタイプはモジュール化の大きな武器の1つであり，単にデータ構造を規定するものととらえるのは望ましくない．プロセデュアが詳細の制御構造を隠蔽するように，抽象データタイプも詳細のデータ構造を隠蔽するものでなければならない．プロセデュアは制御構造 (do, if-then-else, case 等) の増加を行うものではなく新しい命令（高い概念を示す）を増加させる．同様に抽象データタイプはデータ構造の追加ではなく新しいデータタイプ（高い概念を表わす）の追加となるべきである．

抽象データタイプの例としてここでは CLU の cluster と Riddle の表記法を示すにとどめる[20),27)]．図-18 に CLU の cluster を示す．

Riddle の表記法 (DDN: Dream Design Notation) は大規模システムを念頭に置いており，その点でプログラミングの色彩の濃い CLU 等とは異なる．サブシステムは外部とはメッセージ交換をもって情報の交換を行うと規定されており，したがってインタフェースはメッセージの流れを規定することにより行える．DDN ではこの通信用のチャネルをリンクとして表現する．このリンクはそれ自身サブシステムでありモニタ (Hoare[13)]の意味で) の性格を有する．リンクはメッセージを蓄積し，転送することができる．リンクはサブシステムにポート (port) により接続される．そしてこのポートは方向性を有しており入力と出力の方向がある．図-19 の例ではポート“note”はそれに対応するバッファ“notice”を有しており，それによりメッセージの形式が指定される．

サブシステムの動作の指定は図-19 の例では“control process”により行われる．この動作の規定は抽象化された形で行われており，たとえば動作指定はポートとそのバッファを用いての記述に限定されてお

```
wordbag=cluster is
  create,    % create an empty bag
  insert,    % insert an element
  print,     % print contents of bag
  rep=record [contents: wordtree, total: int];
create=proc ( ) returns (cvt);
        return (rep ${contents: wordtree $create ( ), total: 0});
        end create;
insert=proc (x: cvt, v: string);
        x. contents=wordtree $insert (x. contents, v);
        x. total=x. total+1;
        end insert;
print=proc (x: cvt, o: outstream);
        wordtree $print (x: contents, x. total, o);
        end print;
end wordbag;
```

図-18 CLU の cluster

```
[noticer]: SUBSYSTEM CLASS;
  QUALIFIERS; #_under_surveillance END QUALIFIERS;
  note: ARRAY [1:: #_under_surveillance] OF OUT PORT;
    BUFFER SUBCOMPONENTS; notice OF [datum_change]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITIONS; notice=change END BUFFER
      CONDITIONS:
    END OUT PORT;
  observer: ARRAY [T:: #_under_surveillance] OF CONTROL
    PROCESS;
    MODEL; ITERATE
      see_it: SET notice (MY_INDEX) TO change;
            SEND note (MY_INDEX);
            END ITERATE;
      END MODEL;
    END CONTROL PROCESS;
  END SUBSYSTEM CLASS;
```

図-19 DDN におけるサブシステムの記述

り，内部の変数を用いていない点とか，高概念の命令を用いて詳細を隠すとかの方法で行われている．DDN では 7.2 節で述べたモジュール記述の果たすべき条件が比較的よく実現されている．DDN は図を用いての比較的非形式的な表記法と CLU 等の比較的詳細プログラムの記述に適している言語との中間を埋めるものとして興味ある存在である．

抽象データタイプに属するモジュール化法として重要なものに Parnas のモジュール化法がある[24]．これは抽象データタイプがプロセデュア等の抽象化（モジュール化）と併存または相補う形で用いられるのに対して，全体のモジュール化をよりデータ依存に傾斜させたものである．SADT のデータダイアグラムはその極端例でモジュール化を完全にデータ依存で行う．したがって Parnas のモジュール化法は抽象データタイプと完全なデータ依存のモジュール化法の間にあると言えよう．

さて以上で，機能，トランザクション，制御，データを基準とするモジュール化とその記述法について述べたが，第2章で述べた他の多くの基準に対しては現在のところほとんど研究が行われておらず，したがって本解説からも除外した．

8. モジュール発見のプロセス

システムのモデル化には，それが構造モデルであれ，他のモデルであれシステムの深い理解が必要であり単に記述法を整えるだけではすまない．したがって，各システムの研究，理解が先行しなければモデル化が不可能なのは明らかであり，そのための地道な努力が必要であることは言うまでもない．

7章で述べた記述法はいずれもある程度モジュール化（モデル化）の概念と結びついている．しかし，その重要部分はまだまだ魔法の領域と言えよう．従来からの機能分割法では経験の蓄積以外に大きな進歩が期待しがたく分析者，設計者の能力に大きく依存する．

Jackson 等のデータ構造を定めてそれにより処理構造を定めていく方式はデータ構造の方がより現実のシステムを忠実に反映しているという主張が正しければ機能分割よりも成功する可能性がある．これもやはりシステムの種類等により左右される．事務処理システム等に多く適用され成功しているようである．

Constantine 等の構造化設計[33]は従来ややもするとあいまいであったモジュール間の結合の度合を定義したという点で進歩である．Coupling と Cohesion の概念および定義は従来のプロセデュアによるモジュール化が基礎となっておりその点で適用しうるシステムの範囲は限られるが1つの有用な指針である．

モジュール発見のプロセスは前述のように大部分魔術とも言える段階であり多くを解説できないのは残念である．

9. おわりに

本解説ではモジュール化に関してその基準の明確化と具体的な記述法について概説した．情報処理システムのモデルを作成することは極めて重要なことと認識されているが，その成功の度合は現在までのところ極めて低いと言わざるをえないであろう．情報処理システムの構築は多くの場合，経験と勘によっており系統的な方法が確立されていない．

情報処理システムのモデル確定にはこれから地道な努力が要求されるが次の3つの分野からの研究が必要である．

（1） モジュール化基準の相互の関係

個々のモジュール化基準の優劣のみならず，それらの相互関係を明確に，特に定量的に明らかにする必要がある．

（2） アプリケーションまたはシステムの種類ごとのモデル化

情報処理システムは多くのシステムに共通な部分も多いが，システムまたはアプリケーションごとに異なる面も大きい．より具体的なモデル化のためにはアプリケーションごとのモデル化が明確にされなければならない．

（3） システム特性ごとのモデル化

性能，変更の容易さ等のシステム特性はアプリケーションの違いにもよるが多くの場合において各システムに共通な横断的な特性である．したがって，システムにかかわらずこれら特性評価のためのモデルが望まれる．これら各々について今後積極的な研究が行われることを希望する．

参考文献

1) Alford, M. W.: A requirements engineering methodology for real-time processing requirements, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 60-69 (Jan. 1977).
2) Caine, Stephen H. and Gordon, E. Kent: PDL—a tool for software design, Proc. AFIPS NCC, (1975).
3) Chroust, G.: Data interfaces versus control interfaces; a half-backed conjecture, ACM SIGARCH Computer Architecture News, Vol. 5, No. 4, pp. 32-38 (Oct. 1976).
4) Denning, P. J. and Buzen, J. P.: The operational analysis of queuing network models, Computing Surveys, Vol. 10, No. 3, pp. 225-262 (Sep. 1978).
5) Dennis, J. B.: Modularity, Advanced Course on Software Engineering, F. L. Baur, Ed. Springer-Verlag, pp. 128-182 (1973).
6) Dijkstra, E. W.: The structure of "THE" —multi-programming system, Comm. ACM, Vol. 11, No. 5, pp. 341-346 (May 1968).
7) Dijkstra, E. W.: Structured programming, Software Engineering Techniques, J. N. Buxton and B. Randell, Ed., Brussels, Belgium: NATO Scientific Affairs Division, pp. 84-87 (1970).
8) Freeman, Peter: The nature of design, Tutorial on Software Design Techniques, IEEE Computer Society (April 1977).
9) Goos G.: Language characteristics: programming languages as a tool in writing system software, Advanced course on Software Engineering, F. L. Baur, Ed., New York: Springer-Verlag, pp. 47-69 (1973).
10) Harada, M., Kunii, T. L. and Saito, M.: RGT: the recursive graph theory as a theoretical basis of a system design tool 'DESIGN-TOOL'—with an application to medical information system design, Proc. International Symposium on Medical Information System, Osaka (1978).
11) Harada, M. and Kunii, T. L.: A design process formalization, Proc. COMPSAC '79, Chicago (1979).
12) Harada, M. and Kunii, T. L.: IDMS: an interactive design management system based on the recursive graph formalism, Proc. of the 13th IBM Computer Science Symposium, Amagi, Japan (Nov. 1979).
13) Hoare, C. A. R.: Monitors: an operating system structuring concept, Comm. ACM, Vol. 17, No. 10, pp. 549-557 (Oct. 1974).
14) Horning, J. J.: Some desiable properties of data abstraction facilities, Proc. Conference on Data Abstraction, Definition and Structure, Salt Lake, Utah, March 22-24 (1976).
15) Jackson, M.: The Jackson design methodology, "Structured Programming", Infotech State of the Art Report, Infotech International.
16) Kunii, T. L., Browne, J. C. and Kunii, H. S.: An Architecture for evolutionary database system design, Proc. COMPSAC '78, Chicago (Nov. 1978).
17) 国井利泰：「生命型」製品による産業構造の転換，昭和54年電気4学会連合大会講演論文集.
18) Liskov, B. H.: A design methodology for reliable software systems, Proc. AFIPS Conference (1972).
19) Liskov, B. and Zilles, S.: Specification techniques for abstract data types, IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 7-19 (Mar. 1975).
20) Liskov, B., et al.: Abstraction mechanisms in CLU, Comm. ACM, Vol. 20, No. 8, pp. 564-576 (Aug. 1978).
21) Maekawa, M.: Performance adjustment of an APL interpreter, Proc. of the EUROMICRO '79, North Holland Pub. Co. (1979).
22) McGowen, C. L. and Kelly, J. R.: Top-down structured programming techniques, Petrocelli/Charter, New York (1975).
23) Myers, G. J.: Characteristics of composite design, Datamation, Vol. 19, pp. 100-102 (Sep. 1973).
24) Parnas, D. L.: On the criteria to be used in decomposing systems into modules, Comm.

ACM, Vol. 15, No. 12 (Dec. 1972).
25) Parnas, D. L.: On the design and development of program families, IEEE Transactions on Software Engineering (Mar. 1976).
26) Ramamoorthy C. V. and So. H. H.: Software, requirements and specification: Status and perspectives, COMPSAC Tutorial.
27) Riddle, W. E., Wileden, J. C., Sayler, J. H., Segal, A. R. and Stavely, A. M.: Behavior modeling during software design, IEEE Transactions on Software Engineering, Vol. SE-4, No. 4 (July 1978).
28) Ross, D. T., Goodenough, J. B. and Irvine, C. A.: Software engineering: process, principles, and goals, Computer (May 1975).
29) Ross, D. T.: Structured analysis (SA): a language for communicating ideas, IEEE Transactions on Software Engineering (Jan. 1977).
30) Simon, H. A. and Ando, A.: Aggregation of variables in dynamic systems, Econometrica Vol. 29, No. 2, pp. 111-138 (April 1961).
31) Simmons,R. F.: Semantic networks: their computation and use for understanding english sentences, Computer Models of Thought and Language (ed. Schank, R. C.), W. H. Freeman and Company (1973).
32) Stay, J. F.: HIPO and integrated program design, IBM Systems Journal (1976).
33) Stevens, W. P., et al.: Structured design, IBM Systems Journal, Vol. 13, No. 2, pp. 115-139 (1974).
34) Teichroew, D.: Improvements in the system life cycle, Proc. IFIP Congress, 1974, North-Holland Pub. Company.
35) Teichroew, D. and Hershey, E. A.: PSL/PSA: A computer-aided technique for structed documentation and analysis of information processing systems, IEEE Transactions on Software Engineering (Jan. 1977).
36) Wirth, N.: Program development by stepwise refinement, Comm. ACM, Vol. 14, No. 4, pp. 221-227 (April 1971).
37) Wirth, N.: Systematic programming: an introduction, Prentice Hall (1973).
38) Wirth, N.: Toward a discipline of real-time programming, Comm. ACM, Vol. 20, No. 8, pp. 577-583 (Aug. 1978).
39) Zadeh, L. A.: Toward a theory of fuzzy systems, Aspects of Network and Systems Theory (eds. Kalman, R. E. and DeClaris, N.), Holt, Rinehart and Winston, Inc. (1971).

(昭和54年12月21日受付)