TECHNOLOGY REVIEW

UNISYS 技報

通巻 28

1991年2月発刊 Vol. 10 No. 4

特集：ソフトウェア開発の方法

巻頭言

特集「ソフトウェア開発の方法」の発刊によせて……深堀年弘 1

論 文

ソフトウェア開発の形式的方法……山崎利治 3

分散システム機能仕様図……佐藤 博 24

自然言語仕様からのデータフロー図構成法……大野浩史 39

会話型プログラムの仕様記述に関する提案……木下博文 53

通信ソフトウェア開発効率化の手法……宮坂順之 63

ソフトウェア生産における工程検査の方法と進め方……西島政信 82

CAD/CAM システムにおける UIM の実現……松林 毅 103

OMS/B——直接操作型インタフェース記述システム……川辺治之 114

UNISYS シリーズ 2200/1100 の

Common Lisp 処理系……大田一久 122

ソフトウェア改修作業の生産性と信頼性の実体……林 雅彦 136

形式的記述技法への誘い——記述言語はなぜ必要か……染谷 誠 149

新製品紹介……153

図書紹介……157

掲載論文梗概……表 2, 3

日本ユニシス

ISSN 0914-9996

ソフトウェア開発の形式的方法が産業界でも注目されているが，その数学的側面が産業界での実用のための隘路になっている．山崎利治は，**ソフトウェア開発の形式的方法**の中で，この形式的方法を概観し，その一つである RAISE (Rigorous Approach to Industrial Software Engineering) を紹介するとともに，ソフトウェア開発の現場に形式的方法を導入し普及するための教育過程や訓練計画の議論を行っている．

システムの分析，設計作業を有効に行うためには，設計段階でエンドユーザと標的システムについて同一の理解を確立することが不可欠であるが，現在のところそれを満たす設計仕様記述法がない．佐藤博の**分散システム機能仕様図**は，エンドユーザにも容易に理解できること，分散システムの機能分担・タイミングが表現できること，等の条件を満たす変換図法に基礎を置いた設計仕様記述法を提案している．

近年，ソフトウェア工学の分野において，上流工程の重要性に対する認識が深まってきている．ソフトウェア開発過程において，仕様定義や要求分析が信頼性の高いソフトウェア作成のためのより決定的な要因となっているからである．大野浩史の**自然言語仕様からのデータフロー図構成法**は，各種のソフトウェア設計法の初期ステップで行われている問題理解の過程を支援すべく，自然語の要求記述からデータフロー図作成のための枠組みと手順を提案している．

近年，プロジェクトの大型化，プログラム開発の外注化の傾向が顕著であり，より正確な設計者―プログラム開発者間の意志疎通が求められている．意志疎通の主たる手段はプログラム仕様書であり，その果たす役割は重大である．木下博文は，**会話型プログラムの仕様記述に関する提案**の中で，とくに会話型処理プログラムの仕様記述方法について，この方法の特徴・枠組み，および記述例を記している．

1980 年代末以来，ISO による OSI，CCITT による ISDN の標準化が行われてきた．標準化は下位のレイヤより順次進められており，これまでにこれらの機関が作成した標準ドキュメントの量は 10 万ページを越えている．これらの標準は通信ソフトウェアとして実装される必要があるが，現用の方法により実装した場合，多大のマンパワが必要である．宮坂順之は，**通信ソフトウェア開発効率化の手法**の中で，プロトコル開発言語，テストシステムの自動化等，通信ソフトウェアの開発効率化のためのツールの実現方法について記述している．

ソフトウェア生産においては，生産の各工程で不良の入り込む可能性が高く，後工程になればなるほど不良の発見が困難になり，修正に多大な労力と時間を要する．このような問題の回避には，不良が後工程に送り込まれるのを未然に防止する活動等が必要になる．西島政信は，**ソフトウェア生産における工程検査の方法と進め方**の中で，当社（日本ユニシス）システムプロダクト本部におけるソフトウェアのリリースプロセスを紹介するとともに，ソフトウェア検査の一つの形態としての工程検査の方法と進め方，有効性等について事例をもとに概説している．

CAD/CAM システムのユーザインタフェースは，これまではシステム開発のたびに，同じようなプログラムを多大な開発工数をかけて作成していた．このような問題を解決するために，容易に実現・変更ができ，かつシステム間で共通に利用可能なユーザインタフェース・マネージャを作成した．松林毅は，**CAD/CAM システムにおける UIM の実現**の中で，ユーザインタフェースに関する共通の操作ルールのモデル化と，その表現形式および実現方法について述べている．

近年，ワークステーションを始めとする高精度表示装置とポインティングデバイスを用いた計算機の普及に伴い，この上の対話型システムのユーザインタフェースとして，直接操作型インタフェースが注目されている．川辺治之の **OMS/B**――

特集「ソフトウェア開発の方法」の発刊によせて

深堀年弘

本号はシステム構築の中核を成すソフトウェアの開発に関する特集号である．

1968年，西独のGarmischでNATO Technical CommitteeによりSoftware Conferenceが開催され，ソフトウェアのあり方について討議が行われた．この場で始めて「ソフトウェア・クライシス」という言葉と対を成す形で「ソフトウェア・エンジニアリング」という言葉が使われたことは有名である．

以来20年以上の時が経過し，この間ソフトウェアに関して学問的考察と技術の体系化を目指し，さまざまな事柄が提案されてきた．一般に科学・技術が進歩とともに細分化の道を歩むように，ソフトウェア・エンジニアリングもその例に漏れず，多岐に亘っており，余程の専門家でない限りその全体を把握することがむずかしい状況である．

一方，この間のハードウェア，とくに半導体の進歩は目ざましいものがある．あらゆる製品に組み込まれ，機能の高度化に寄与するのみならず，ロボットによる製造工程の自動化，CADによる設計工程の合理化等，設計，生産の両技術の高度化に果たす役割は大きい．大量生産される工業製品の生産技術に関して日本はトップレベルにあり，逆にその競争力の強さは国際的な問題を引き起こすまでになっている．

しかし，ソフトウェアについてはどうであろうか．大部事情が異なっているようだ．この20年を見れば，バッチ型開発からTSSによる対話型開発へ，アセンブラからコンパイラ言語へ，構造化プログラミングの定着，ウォータフォール・モデルによる工程管理の一般化，またデータディクショナリによる開発関連情報の一元管理を中心とした工程一貫開発支援ツールの採用等いろいろ工夫がされてきている．しかし，生産性を決定するさまざまな要因のうち，開発者の熟練度が依然トップにあるというのが定説のようであり，ソフトウェア作りの現場の実感でもある．ただこの時に考えなければならないのは，1968年当時は予想もしえなかったほどコンピュータのコストパフォーマンスが向上し，当時とは比較にならないほどコンピュータシステムの応用分野が広がった結果，社会のすみずみに浸透し，また企業戦略に占めるウェイトも高くなってきている環境にあるということである．コンピュータが高価だった時代には，コンピュータを効率良く使うよう仕事をコンピュータに合わせるシステム作りをしていたが，現在は現実にコンピュータ側が合わせ，素直に利用者がシステムを使えるような環境実現を開発者に要請している．つまりシステムに対する価値観が機能・効率から，より次元の高い使い勝手とか，変更の容易性に移ってきているのであろう．このことは対象に対する設計技術と，どう作るかという生産技術が明確に分けられる工業製品と異なり，両

者の関係が不明確というより，むしろ一体となっているソフトウェア製品の場合，対象の性格の変化がプロセスのあり方に直結しているといえる．ソフトウェアの開発工程を言語の変換列と考えれば，よりふさわしい言語が常に求められているといえるのではあるまいか．

ソフトウェアの開発には，まだ解決すべき課題は多い．しかし一方，ソフトウェア・エンジニアリングの成果も現場で十分応用されているとは言い難い．

本号ではソフトウェア開発のむずかしさを考える上で，また本質的な解決を考える上で重要な意味を持つであろう形式的開発法，仕様記述，ユーザインタフェース，保守等について考察を試みた．ソフトウェア開発に携わる皆様方の参考になれば幸いである．

（システム技術本部生産技術部 部長）

ソフトウェア開発の形式的方法

Formal Methods in Software Development

山 崎 利 治

要 約 ソフトウェア開発の形式的方法が産業界でも注目されている．そこでこの形式的方法を概観し，その一つである RAISE 方法を紹介する．ソフトウェアの品質や経済効果の面での形式的方法の利点は自明であるが，その数学的側面が産業界での実用のための隘路になっている．

この小文の目的は，日本のソフトウェア開発の現場に形式的方法を導入し，普及するための教育課程や訓練計画を議論することである．

Abstract There is a growing interest in formal methods for industrial software development. After a brief review of methods is done here, RAISE——Rigorous Approach to Industrial Software Engineering——is sketched as a tutorial introduction to these approaches. It is self-evident that the formal method helps to radically improve software quality and to considerably cut down software production costs. Regrettably, however, the mathematical aspect of formality has been a bottleneck to its widespread industrial use.

The article is intended to discuss what we ought to do so as to accept these methods and train software people in an attempt to make them exoteric.

1. はじめに

ソフトウェア開発の形式的方法がとくに欧州を中心に産業界にまで普及してきている．通信プロトコルなどのための形式的仕様記述言語 LOTOS が国際規格になってから，形式的方法それ自体までを規格化する動きも活発である．この小文は形式的方法の一端を紹介し，ソフトウェア開発の現場へその導入と普及を図るための一試論である．

1章で形式的方法についての最近の状勢を述べ，2章で形式的方法一般についての概略とともに固有名をもつ方法のいくつかを挙げる．これは注釈付き文献表の域を出ない．3章で，とくに産業界での実用のために欧州で展開中の RAISE の仕様記述言語 RSL をややていねいに説明する．4章はまとめであり，日本における開発現場でのこのような形式的方法の適用の道を考える．

次は，C.ジョーンズがその VDM の教科書[40] に引用している B.オークリィの言葉である．

「アルベィ・ソフトウェア工学計画の一番の業績は学問の世界から「形式的方法」を引き出し産業界で利用させるという難事業に成功したことである．この業績はいくら高く評価してもしすぎることはない．というのも形式的方法こそがより良いソフトウェアをつくる大道であり，経済効果も大きい．つまり，19世紀における土木工学の革命に匹敵するものであるからである．」

また，次はデンマーク工科大学の D. ビョルナーが行った InfoJapan'90 での招待講

演からの引用である．

「プログラミング，課題のモデル化，ソフトウェア工学での形式的方法の利用はソフトウェアの品質向上，経費の低減，開発期間の短縮に著しく寄与する．これは欧州での経験によって今では自明になっている．……形式的方法の使えないソフトウェアハウスはそのつくるソフトウェアの低い品質や長い開発期間によって客を失う危機に瀕している．爆弾発言と聞こえるかもしれないが，これがここしばらく，つまり今世紀の間は続く欧州のプログラミングやソフトウェア工学での趨勢なのである．……ソフトウェア開発の最も重要な分野であるプログラミング方法論において欧州は指導的立場にあると信じている．構造的プログラミング，JSP/JSD，VDM，Z といった具合である．……米国や日本がこの面をもっと強力に推進していないのが，不思議である．」

研究面で米国や日本が形式的方法に無縁であったわけではないが，両国とも形式的方法を広く実用に供していないのも事実である．ごく最近，米国では電気電子技術者学会 IEEE が，その 3 誌，Computer，Software，Transaction on Software Engineering(1990 年 9 月号)に形式的方法についての特集を組み，欧州生まれの Z などを大きく紹介している．プログラム言語 Ada の普及とともに形式的方法も着実に進展するだろう．さて日本の場合，そのソフトウェア開発は憂慮すべき状況にある．開発現場の多くは 70 年代のソフトウェア工学の成果も享受していない．ここへ形式的方法を全面的に導入しようというのは白昼夢に近い．そこで，ソフトウェア開発の実務家がこの形式的方法をどう考え，どうすべきであるかの一案を考えたい．これがこの小文の目的である．

ソフトウェア開発の形式的方法は「正しいソフトウェア」をつくることである．正しいソフトウェアとは，ソフトウェアの仕様があり，ソフトウェアがその仕様を満たしているという意味である．

形式的方法は形式的と方法という二つの用語を含んでいる．形式的とは，対象に関わる議論がその内容や意味に言及しないで形式だけに頼ってできることをいい，方法とは，一時的な術や方便ではない客観的な認識と論理による目標達成への道程をいう．ソフトウェア開発の形式的方法は，したがってソフトウェア作成のための多くの原理(理論)をもち，その原理の適用手段である技術が存在し，さらにその適用を円滑にする道具を用意し，これらを選択し利用する手順を提供するものである．形式化は一般に対象についての理論展開，すなわち対象がもつ性質の発見，解析，検証をやさしくするのであり，さらに，それらの事実，ソフトウェア開発の場合であれば，ソフトウェアやその一部についての仕様や説明やコードなどであるが，その再利用や検証などを計算機によって支援できるようにするものである．

散見する形式化，つまり，形式的な扱いや形式的方法による事例のいくつかを紹介する．

1) Algol 60 の構文……プログラム言語 Algol 60 の具体構文を厳密で簡潔に記述したバッカス記法は仕様の形式的記述の最古の例である(実は，紀元前 5 世紀のインドの大文法学者パーニニが，古典サンスクリットの連声法の記述に同様の方法を採用している)．

2) プログラム言語の意味記述……多くのプログラム言語に対して，属性文法や表

示的意味論による形式的な仕様記述が行われている[6],[8].

3) 実時間操作系の核……X線診断装置に埋め込む計算機のための実時間操作系の核を，仕様記述言語Zによって再構築した例がある[55]．操作系の核は，プロセスをスケジューリングし実行管理する部分であるが，埋込み計算機をAからBへ変えるためにその核をつくり直す必要があった．そこで操作系についての文書とコードから核の仕様をZによって記述した．このとき，その核がデッドロックを起こすことのあることが判明した．これはZによる記述のもつ数学的性質から直ちに発見できたという．実際の利用例では，計算機のもつタイムアウト機能と，利用プログラムのプロセスの実行形態から，患者にX線を長時間にわたって照射する危険は皆無であったというのだが．

4) オシロスコープの仕様……ブラウン管上に信号の時間波形を表示するオシロスコープも，現在ではアナログ装置ではなく計算機によるディジタル制御によって，極めて高度な機能をもっている．この仕様をやはりZで記述している[16]．

5) 浮動小数点演算マイクロコード……IEEE規格の浮動小数点演算コードを，トランスピュータのためにZによって開発した例がある[2]．この場合も非形式的方法によるコードよりも，はるかに短期間で正しいコードを開発している．

6) LOTOS……通信プロトコルは，自然言語や状態推移図などによって記述するのには複雑すぎるので，その厳密で適切な表現手段として形式的仕様記述言語LOTOSが開発された．これは通信制御装置の実現の検査設計にも効用をもっている．

2. 形式的方法

ここで形式的方法を概観しておきたい．

形式的でない，つまり非形式的方法によっても，あるいはまったく方法によらないでもソフトウェアは開発できる．D.E.クヌースによるTEXは方法によらない一例である(もっとも，クヌースはこの仕事を通して文芸プログラミングを提唱しているが)．この場合には，もちろん，もし必要なら，いつでも問題をいくらでも形式化して解析し算法を導き出しコード化する能力がなければならない．事実，クヌースは算法についての大部な百科辞書を書き続けているくらいだから，それができTEXもつくれるわけである．

形式的と非形式的との中間に位置する，いわば，半形式的方法も存在する．構造的分析[17]や，ジャクソン・システム開発法[38]などである．これらの方法も，その利用者側の事情によって半形式的にしているといえる．たとえば，前者の小仕様をPrologで，後者の仕様をLOTOSで書いたりすれば立派な形式的方法になるわけである．

ここで概観したい形式的方法は，ソフトウェアの形式仕様をつくり，実現したソフトウェアがその仕様を満たすことを形式的に立証するものである．この意味で形式的方法は，仕様記述と実現の検証という二つの側面をもっている．

2.1 形式仕様

用語「仕様」は大方の同意が得られた概念ではないが，仕様とプログラムとを分ける理由は次による．現在の多くのプログラムは実行効率のために代入文をもつ命令型

言語によって実現する．この命令型言語は参照の不透明さ，つまり式の一部をそれと等価なものに書き換えると全体の意味が変わってしまう性質をもっている．したがって，命令型言語によるプログラムは本質的にわかりにくいものであり，プログラム文の他にプログラムのもつ性質を明示する仕様が必要になるのである．そしてこの仕様がプログラムの正しさを議論する根拠になるわけである．

仕様を形式的に書くのは，もちろんプログラムのもつべき性質を厳密にまた簡潔に記述するための楽な方法であり，プログラムの正しさの立証のしやすさのためでもあるが，仕様自体をも数学的な厳密さで議論したいからである．仕様に矛盾がないという整合性は，もし仕様が矛盾を含めば，どんなプログラムでもその仕様を満たすから重要な主題である．また同一入力に対して異った出力を出す二つのプログラムがともに仕様を満たすことを引き起こさないという完全性も同様である．

以上のような議論に対して形式性の利点は，神学の僕であった論理学が形式化され記号論理学が生まれたことからも明らかである．仕様に対するさまざまな要請，たとえば必要な情報を含み，余分な情報を含まない最小性，簡明さ，読みやすさ，書きやすさ，変更のしやすさなども形式化によって達成しやすくなる．日本語などの自然言語に図や数式を補なった記述は一面読みやすいかもしれないが，あいまいさをもったり，過不足や不整合にもなりやすい．図書館問題に対する形式仕様[56]や，行編集問題の検証[26]に見られたように形式仕様にさえ間違いや問題が発生する．しかし，形式仕様はこのような問題点や誤りを早期にしかも楽に発見させることも事実なのである．

2.2 モデルと仕様記述単位

形式的方法の仕様記述の側面には仕様対象をどう見るかというモデルと，その表現手段，すなわち，記述言語と，仕様記述単位と対象全体との関係によって必要となる仕様の構造化機構などの問題がある．

仕様記述単位のモデルとしては状態機械がよく利用されてきた．それは，より抽象的には抽象データ型になる．抽象データ型とは，一つまたはそれ以上の集合と，それらの間で定義した演算群との組を考え，その表現や実現の詳細を捨象した抽象実体のことで，数学的には多種代数(many-sorted algebra)である．

抽象データ型は，したがって次を与えれば定義できることになる．

1) いくつかの集合．種ともいい，それ自体が抽象データ型である．
2) 抽象データ型を定める演算群の演算名とその型(定義域と値域)．
3) 演算の定義．演算を一つづつ直接的に定義したり，いくつかの演算間に成立する関係などによって間接的に定義したりする．

抽象データ型の具体的な記述手段については次項で触れるが，その表現方法に二通りがあって次のようにいうことがある．一つは，その抽象データ型の台(career)を直接に示すモデル指向(model-oriented)記述と，二つめは台を直接に示さない性質指向(property-oriented)記述である．

対象の挙動そのものを直接的に取り扱いたい場合のモデルとして並行プロセス，とくに交信プロセスがある．これについては3章のRAISEの仕様記述言語RSLで触れる．

2.3 仕様記述言語

仕様記述単位としてのモデル，つまり抽象データ型を表現するためには参照の透明な言語を用いればいい．次のような言語がそのために使用されている．

1) 論理言語……正しい推論形式を論理といい，その研究が論理学である．この論理学の形式面による研究のために形式的体系が考えられているが，その言語がこの論理言語である．ふつう，この言語は，論理結合子($\vee$，$\wedge$，$\sim$，$\Rightarrow$，$\Leftrightarrow$)，量化記号($\exists$，$\forall$)，個体変数や個体定数，関数記号，述語記号などの上に，項，素論理式，論理式を構成するものである．仕様記述のためには，適切な体系を選択して言語を設定すればいい．仕様記述言語としては最も一般的なものである．

2) 論理型言語……ある論理体系の論理式をプログラムと考え，そのプログラムが意味する計算がその体系での証明になるという言語である．ホーン節集合を計算モデルとするProlog族はこの代表例である．

3) 代数型言語……抽象データ型のもつ演算が満たさなければならない関係を等式論理によって記述しようという考えのための言語である．具体的には等式系によって記述することになり，例を後で述べる．この言語によって書いた仕様は，ある性質を満たせば項書換え系によって実行できるようになる．

4) 関数型言語……プログラムを関数定義と関数適用によって構成する言語である．この意味はラムダ算モデルや項書換え系によって与えられる．関数は帰納的に定義することが多く，構造をもつデータもその構造全体を一つの値として扱え，関数を関数の引数や値にでき，簡明で強力な表現が可能である．MLやMirandaなどがこの例である．

2.4 構造化機構

比較的小さな仕様単位のいくつかを合成して大きな仕様を構成したり，一つの仕様単位に対して構造を付け加えたり，既存の構造の一部を取りはずしたりしたい．このような操作を許す機構は仕様の再利用にも役立つので，仕様記述方法に含める必要がある．

2.5 実現の正しさの立証

実現したプログラムが正しいという主張のために，

1) プログラムの検証

2) 正しいプログラムの作成

3) プログラム変換

などが考えられている．

1)は，一つの論理体系を定め，仕様とプログラムから検証条件をつくり，それを体系の中で証明することによって，プログラムの正しさを主張するものである．後出のVDMの項で再度触れる．

2)は，初めから正しいプログラムをつくり，プログラムが完成した時には正しさが保証されているというものである．ダイクストラの方法(2.7節)や構成的プログラミング(2.8節)がこの例である．

3)は，仕様から意味を変えない変換を反復して実用的なプログラムをつくろうというものである．ジャクソン・システム開発法の実現段階はこの例である．プログラム

変換とは，もともと，実行効率や記憶効率の改善のために意味を変えないでプログラムを書き直すことをいい，関数型言語に対して研究が進んでいる．関数型言語によるプログラムは参照の透明さをもち，それを仕様と考えてもいいが，この仕様のプログラム変換によって，より実用的なプログラムが得られるわけである．この場合の変換としては，再帰定義の効率化・除去，並列化，非決定性の除去，部分評価などがある[24]．

2.6 支援ソフトウェア

形式的方法に不慣れなソフトウェアの実務家はそれを秘儀的と受け取る傾向をもち，受け入れやすくするための方策として，ソフトウェア・ツールを求めることがあった．しかし，実際には形式的であるためにツールは用意しやすい．つまり，プログラム言語に対するソフトウェア・ツールを広範囲に拡大できるわけである．次のようなツールが考えられている．

1) 構文ツール……仕様，詳細化，コードなどの作成時や修正時の具体構文に依存したエディタがこの代表である．他にも文面だけから判断できるツール，たとえば，型検査，名前の参照表作成，検証時の証明図を見やすく画くツールなどがある．
2) 意味ツール……仕様や詳細化の検査のためにいろいろなツールがある．実際のデータを与えて動かすアニメーション・ツール，プログラム変換を行うツール，詳細化時の検証条件の生成用ツール，検証条件である論理式を簡約するツール，あるいは，Ada，C++などのプログラムへの翻訳ツールなどである．
3) 検査ツール……検査用データの生成ツールがいろいろ考えられている，とくに，システムの動的特性を直接取り扱う仕様記述に対して考えられている．
4) 実用ツール……仕様，詳細化，コードなどのライブラリが開発中につくられていく．このために，版の管理，構成管理，開発記録，修正変更とやり直し，状況報告などが必要でこれらのツールがこの範疇に属する．

2.7 ダイクストラの方法[18],[19],[21],[27],[47]

プログラムを書いた後でその正しさを立証するのは面倒だから，初めから正しいプログラムをつくるべきだと主張するE. W. ダイクストラは次のような方法を考えた．

この方法の骨子は，形式的な仕様をもとにその仕様を満たし必ず停止するプログラムをダイクストラ算という形式的な計算を用いて作成することである．

プログラム S の仕様を，ここでは前件 P と後件 Q との組であると考える．前件 P とは，プログラム S に対する入力データについて成立しなければならない条件を述語として書いたものであり，後件 Q とは，入力データと出力データに関して成立しなければならない条件をやはり述語として書いたものである．プログラム S を前件 P のもとに実行を開始すると必ず停止して，そのとき後件 Q が成立することを，

$$\{P\}S\{Q\}$$

と書く．プログラム S と後件 Q とを固定したとき，$\{P\}S\{Q\}$ となる前件 P は一般にはいくらもある．つまり，$\{P'\}S\{Q\}$ に対して $P \Rightarrow P'$ が真になるような P なら，どんな P に対しても $\{P\}S\{Q\}$ が成立する．$\{P\}S\{Q\}$ が成立する最も弱い前件 P を S の Q に対する最弱前件といい，

$$P = wp(S, Q)$$

と書く．この最弱前件がダイクストラの方法の鍵となる重要概念である．

最弱前件はプログラム言語によって定まるが，この方式でプログラムがつくりやすい言語も考えられ，ダイクストラは「ふたつき命令」を提案している．それによる最弱前件の計算は次のようになる．

$wp(S, false)=false$

$wp(S, P)\Rightarrow wp(S, Q)$　ただし $P\Rightarrow Q$ のとき

$wp(S, P)\wedge wp(S, Q)=wp(S, P\wedge Q)$

$wp(S, P)\vee wp(S, Q)\Rightarrow wp(S, P\vee Q)$

$wp(skip, Q)=Q$

$wp(abort, Q)=false$

$wp(x:=e, Q)=Q[e/x]$　($Q[e/x]$ は Q に出現するすべての x を e で置き換えたもの)

$wp(S_1\ ;S_2, Q)=wp(S_1, wp(S_2, Q))$

$wp(if\ \ B_1\rightarrow S_1\square B_2\rightarrow S_2\square\cdots\square B_n\rightarrow S_n\ fi, Q)$

$=(B_1\vee B_2\vee\cdots\vee B_n)\wedge(B_1\Rightarrow wp(S_1, Q))$

$\wedge(B_2\Rightarrow wp(S_2, Q))\wedge\cdots\wedge(B_n\Rightarrow wp(S_n, Q))$

$wp(do\ \ B_1\rightarrow S_1\square B_2\rightarrow S_2\cdots\square B_n\rightarrow S_n\ \ od,\ \ Q)$

$=H_0(Q)\vee H_1(Q)\vee\cdots$

ここで，$H_0(Q)=Q\wedge\sim(B_1\vee\cdots\vee B_n)$

$H_k(Q)=wp(IF,\ \ H_{k-1}(Q))\vee H_0(Q)$

IF は $do\cdots od$ を $if\cdots fi$ に書き換えたものである．

$H_k(Q)$ は S_i のうちの一つを少くとも k 回選択実行して停止し，そのとき Q が成立する最弱前件である．ところで，このままではいつ停止するかが判明せず不便である．いま t をプログラム変数上の整数値関数とし $wdec(S, t)$ を S の実行が停止し，t の値を少くとも 1 は減少させる最弱前件とする．そこでは次が成立する．

$$(Q\wedge B_1\Rightarrow wdec(S_1, t)\wedge\cdots\wedge(Q\wedge B_n\Rightarrow wdec(S_n, t))$$

ならば，$BB\equiv B_1\vee\cdots\vee B_n$ として，

$$Q\wedge BB\Rightarrow wdec(IF, t)$$

そして，

$P\wedge BB\Rightarrow wp(IF, P)\wedge wedc(IF, t)\wedge t\geq 0$ が成立すれば，

$$P\Rightarrow wp(DO, P\wedge\sim BB)$$

も成立する．また次も主張できる．

$P\wedge BB\Rightarrow wp(IF, P)$ ならば $P\wedge wp(DO, true)\Rightarrow wp(DO, P\wedge\sim BB)$

ダイクストラによるプログラムの作成は，

$$\{P\}S\{Q\}$$

を満たす S を見つけ出すことである．このために，

1) プログラム案 S を立てる．

2) $wp(S, Q)$ を求める．

3) $P\Rightarrow wp(S, Q)$ を確かめる．そうであれば S が求めるプログラムである．

4) 3)が成立しなければ，別案 S' を立て，それを S として 1)へ戻る．

問題は反復を必要とする場合である．反復中の不変条件や上の t を見つけることが

大切である．これらについての説明は省略する．

2.8 構成的プログラミング[15],[31],[36],[49]

この佐藤雅彦(東北大学)の命名によるというプログラムの作成方法は，次のようなものである．ある論理体系を設定すると，プログラム仕様をその体系の中での定理として述べ，その証明からプログラムが自動的に抽出できるようになる．この場合プログラムは，必ず停止し仕様に対して正しいことが保証される．体系によっては，計算可能関数のすべてがそのようにして得られるとは限らないが，十分広い範囲の関数がそうして得られることがわかっている．このような体系は一般に直観主義論理のもので，P. マルチン・レフの構成的型理論や，T. コカンと G. ユエらによる構成理論などが現在盛んに研究されている．また実際のシステムとしては，林晋(龍谷大学)の PX や R. コンスタブル(コーネル大学)らの Nuprl(ニューパール)などがある．いますぐにこの考えによって実用に供する巨大システムを開発できる段階にはないが，正しいプログラムの実現方法として興味深いものである．

2.9 形式的方法のいろいろ

ここで固有名をもつ形式的方法のいくつかを恣意的にあげる．他にも，もちろん，多くの方法がある．

ACT ONE/ACT TWO[22],[23]

ACT(Algebraic specification techniques for Correct and Trusty software system)は，ベルリン工科大学の H. エーリッヒらによる代数型言語である．ACT ONE は抽象データ型の仕様記述言語であり，ACT TWO は ACT ONE の拡張で抽象データ型を参照するとき何を参照するか，また何の参照を許すかなどのいわゆるモジュール化機構や，抽象データ型の定義が実際には何を意味するのかについて制御できる機構などを付加したものである．ACT ONE は国際規格となった形式的仕様記述言語 LOTOS のデータ型定義にそのまま利用されている．

CIP[3],[13],[14],[51]

F. L. バウエルを中心とするミュンヘン工科大学の CIP(Computer-aided Intuition-guided Programming)グループは，ミュンヘン計画 CIP を 1968 年に発足させ，言語やそれによるプログラミング・スタイルの差に無関係な多くの言語に共通する意味的基盤を求め，その上に将来のプログラム言語が合理的につくられるよう研究してきた．その成果が言語 CIP-L とプログラム開発支援系 CIP-S である．

CIP-L は仕様から実現までの広範囲を包含する言語で，論理型，代数型，関数型，操作型，並行性などの様式による記述を許すものである．この言語は，具体構文にある程度の自由を残すために抽象構文によって定義し，その意味は核言語を用意しそれへの変換によって与える．

CIP-S は，プログラム変換によって行うプログラム開発を支援するもので，これ自体も CIP-L で記述されている．

HDM[43]

スタンフォード研究所の L. ロビンソンらによって米国海軍のために 1970 年代に開発された形式的方法である．この種のものとしては最も古い．ダイクストラの抽象機械，パーナスのモジュール，ホアのプログラムの検証などを，一つのプログラ

ム作成方法としてまとめたものといえる．仕様記述言語SPECIAL(SPECIfication and Assertion Language)と HSL(Hierarchy Specification Language)をもち，パーナス・モジュールとそれらの間の関係とを記述する．このような仕様を順次詳細化してプログラムをつくるのであるが，そのためにILPL(Intermediate Level Programming Language)を用意している．

IOTA システム[48]

1976年から1983年にかけて京都大学の中島玲二らによって開発された仕様化から実現まで，さらに検証をも含む形式的方法である．もともとは，引数をもつ抽象データ型の定式化やそれによるプログラムの作成およびその検証についての研究であったが，仕様記述や設計・実現のための言語を設定し，多種一階述語論理に基づく検証系を考え，総合的なソフトウェア開発方法とそれを支援する環境を用意することになったものである．

LOTOS

対象の動的な挙動を直接的に記述する目的をもった形式的な仕様記述言語で国際規格(ISO 8807)である．静的なデータの記述はACT ONEの抽象データ型に準拠し，動的な側面の記述は交信プロセス理論であるR. ミルナーのCCS[46]とC. A. R. ホアのCSP[34]を基礎にしている．

me too[1]

英国のアルベイ計画の支援を受け，P. ヘンダーソンを中心とするスコットランドのスターリング大学とSTC Technology Ltdとの4年間にわたる共同開発の成果がこの *me too* である．これは，

形式仕様＋実行可能性＝プロトタイピング

という考えによる．開発方法はモデル化，仕様化，検査の段階をもつと考える．形式仕様はモデルと仕様の二部から構成する．モデルとしては抽象データ型の種と演算の型，および演算が利用者から見える，見えないの区別を与える．仕様としては，モデルに与えた種の具体的表現と演算の関数型表現を与える．この形式仕様を *me too* shellによって実行し，モデルと仕様の検査を行う．

OBJ2[25],[36]

電子技術総合研究所の二木厚吉らによって開発された．項書換え系という実行機構をもつ代表的な代数型言語である．独自の工夫により次のような特徴をもっている．

① 引数付きモジュールを許す．
② 関数を関数の引数や値にできる．
③ 種の間の包含関係が定義でき，それによって例外処理や関数の多重継承が可能になる．
④ モジュールをそのまま，制限，拡大などを伴なって参照できる．

VDM

VDM(Vienna Development Method)の歴史は古い．誕生は1972年，場所はIBMウィーン研究所である．そこでは操作的仕様記述言語VDLによってプログラム言語PL/Iの記述を行ってきたが，新計算機のためにPL/Iの処理系を表示的意味記

述によって開発しようとし，VDL に代わる VDM を生むことになった．

ここで VDM 自体も形式的方法として成熟を見せるが，新計算機は世に出なかったのでやがて舞台はウィーンを離れ，デンマークと英国で VDM は温存され適用事例を豊富にすることになる．デンマークでは D. ビョルナーを中心にさまざまなプログラム言語の仕様記述に使われ[6],[8]，英国では C. ジョーンズがモデル指向の形式的方法として普及を図っている[40]．やがて，非形式的方法によるソフトウェア開発の欠点に気付いた産業界は VDM に注目する．産業界への普及のためには，しかし，VDM の整備が必要であった．英国流とデンマーク流の記法を統一し，さらにモジュール化機構を付加したり，段階的詳細化に伴なう検証用論理体系の設定，産業界向きソフトウェア・ツールの用意，教育プログラムや教育資料の充実などである．

1987 年になると欧州共同体委員会の後援によって VDM 利用者団体である VDM 欧州グループが結成され，年間 3, 4 回の会議や 1 週間にわたる研究集会が行われるようになる[7],[9],[10]．ここでは VDM に関する経験，教育，ツール，方法論，数学的基礎，規格化などが話題になる．英国標準局はまた VDM の仕様記述言語の規格化を検討し，また国際規格にもしようとしている．

形式的方法としての VDM は，デンマーク派と英国派でアプローチが異っていたように見える．デンマーク派は表示的意味記述の方法をそのまま採用する．すなわち，構文領域，抽象構文，文脈条件，意味領域，意味関数を定義して仕様とし，できるだけそのままの実現を図る．領域は領域方程式系によって帰納的に定義する．特別に工夫して継続意味論を避け素朴意味論的な扱いをする．デンマーク派の課題は主として言語処理系であったからこれはごく当然であった．英国派は抽象データ型をモデル指向で記述し，詳細化によって実現しようとする．このとき，仕様そのものに対して，また詳細化に対して必ず検証を義務付ける．次にこの検証について触れておく．

英国派 VDM の仕様記述は抽象データ型を仕様単位とした．この抽象データ型は状態機械である．つまり，型は状態 S であり，型の初期値，すなわちその型を生成した時の値である初期状態 s_0 をもつ，この型の演算 ω は一般に，

$$\omega : I \times S \to S \times O$$

の形をしている．これを前件 pre-$\omega(i,s)$，後件 post-$\omega(i, s, s', O)$ の形で定義する．前件とは $I \times S$ 上の述語であり，後件とは $I \times S \times S \times O$ 上の述語である．演算が部分関数である場合の取り扱いのために 3 値論理を採用し，その自然演釈系を用意し，この上で形式的な証明を行うものとする．

① 演算に対して，前件が成立するデータに対しては必ず後件を満たすデータが存在することを証明する(実現の可能性)．

$$\forall i \in I \forall s \in S \bullet \text{pre-}\omega(i, s) \Rightarrow \exists s' \in S \exists o \in O \bullet \text{post-}\omega(i, s, s', o)$$

② 演算を直接的に定義した時には，前件を満たすなら必ず後件を満たすことを証明する．

$$\forall i \in I \forall s \in S \bullet \text{pre-}\omega(i, s) \Rightarrow \text{post-}\omega(i, s, \omega_s(i, s), \omega_o(i, s))$$

③ 抽象データ型(上位) S をより具体的な抽象データ型(下位) S' に詳細化して実現に近づけるが，その時，次を証明する．(ret:$S' \to S$ を考えて)

・上位型のすべてのデータ(状態) s に対して，それを具体化した下位型のデータ s' が存在すること．

$$\forall s \in S \quad \exists s' \in S' \bullet \mathrm{ret}(s') = s$$

・とくに上位型の初期状態 s_0 に対応する下位型データ s'_0 が存在すること．

$$\exists s'_0 \in S' \bullet \mathrm{ret}(s_0') = s_0$$

・下位型データに対応する上位型データが上位型の前件を満たせば，下位型データが下位型の前件を満たすこと．

$$\forall s' \in S' \bullet \text{pre-}S(\mathrm{ret}(s')) \Rightarrow \text{pre-}S'(s')$$

・下位型データが後件を満たし，対応する上位型データがその前件を満たせば，それが後件をも満たすこと．

$$\forall \overleftarrow{s'}, s' \in S' \bullet \text{pre-}S(\mathrm{ret}(\overleftarrow{s'}),\ \mathrm{ret}(s')) \wedge \text{post-}S'(\overleftarrow{s'}, s')$$
$$\Rightarrow \text{post-}S(\mathrm{ret}(\overleftarrow{s'}),\ \mathrm{ret}(s'))$$

・演算の具体化は演算の合成によって行うが，これが正しいことをホア流の論理によって証明する．

Z[2],[16],[20],[32],[37],[53],[54],[55]

仕様記述言語 Z は，1970 年代よりオックスフォード大学プログラミング研究グループによって育成され，最近では産業界でも採り上げられている．教育的資料，事例ともに豊富に存在し，その点では VDM と人気を二分している．抽象データ型を，型をもつ集合とそれらの上の演算と，それらが満たすべき公理をスキームという視覚的な枠組を用意して記述する．そのスキームを組み合わせて大きな仕様を書きやすくする機構をもっている．またスキームに引数を許す．

3. RAISE[11],[28],[29],[30],[44],[45]

形式的方法の味見のために RAISE を採り上げる．それは，これが Rigorous Approach to Industrial Software Engineering の名をもつからである．RAISE は VDM を出発点としている．しかし VDM は長期間にわたって大勢の人々によって拡張され，修正され再定義・再解釈されてきている．実際，VDM が産業界に普及し始めて以来，さまざまな問題が指適されていた．たとえば，

1) VDM はソフトウェア・ツールをもっていなかったが用意するべきである．
2) VDM はモデル指向であったが，性質指向の良さも採用したい．
3) VDM は並行性の記述に冷淡であったが，これを採用したい．
4) VDM は仕様の構造化に注意を払ってはいなかった．モジュール化機構は重要である．
5) VDM の記法は必ずしも一定していない．

以上のような要望を入れ，形式的意味をもち，検証のための形式的体系をもつ新言語を設定すべきである．

このような声から，欧州共同体委員会の支援を受け，デンマークと英国の大学・研究機関と民間企業とによる研究開発組合を結成し，RAISE 計画が実施されたのである．目標は数学的基礎をもったソフトウェア開発方法の構築であり，その言語や開発環境を支援するソフトウェアを用意することであった．

1985年から5年間，115人年の努力が次のような成果として結実した．

1) RAISE方法
2) 仕様記述言語RSL
3) ソフトウェア・ツール
4) RAISE文書
5) 教育訓練コース

仕様記述言語RSLを眺めよう．そのために次のような問題を考える．

「選良であるA氏の有能な秘書は選挙区の有権者の誕生日帳をつくり，いつも更新している．目的は有権者の誕生日に男性なら酒一升，女性ならバラの花束を届けることにある．これを計算機で処理しようと秘書は思いついた．そのシステムを考えよ．」

RAISEの仕様記述言語RSL(RAISE Specification Language)による仕様もVDM流に考えればいい．つまり，抽象データ型「誕生日帳」を考える．新しい空白の帳面に，有権者の誕生日を記録していく．なかには選挙区から引越しする有権者もいるだろうから，記録を消す必要もあるだろう．日付によって，その日を誕生日とする人の表が欲しいから，日付から有権者を選び出す演算も必要である．あの有権者の誕生日が知りたいという要望もあるかもしれない．誕生日帳に記録すべきデータは本質的には有権者名とその誕生日だろう．このような考えから抽象データ型誕生日帳が見えてくる．実際の課題は，酒とか花束とかもう少しの仕事が必要であるが，誕生日帳ができていればもう問題ではないだろう．

RSLは，操作型，関数型，並行型の記述が可能であり，しかもモデル指向，性質指向の両者を許す．操作型の記述は一見奇異であるが，段階的詳細化を行う場合に便利であるからという理由でVDMから引き継いでいるものである．このうちのいくつかを試みてみよう．

まず，関数型でモデル指向として書いてみよう(図1)．

class…*end* をクラス式といい，…のところにさまざまな宣言を書き，それらが定義するモデル族を表すと考える．*object* BIRTHDAYBOOK：はクラス式が定義したモデルのうちの一つを選び，それにBIRTHDAYBOOKと名付けることを意味する．*type*…は型宣言である．VoterやDateはそれ以上型の内容を明示していないが，やはり型であることを示す．Bdb=Voter$\xrightarrow[m]{}$Dateは型BdbがVoterからDateへの写像の型であることを表す．写像型$F=D\xrightarrow[m]{}R$の値fに対しては，定義域Dの値dに対して写像fを作用させて値$f(d)$を得る演算，写像fに対してそのdの値をrに更新する演算$f\dagger[d\mapsto r]$，写像fの定義域Dからdを制限除去する演算$f\backslash\{d\}$などが付随している．型を定義する式で$A\times B$は直積型を，*A-set* はA型の値の集合，つまりAの巾集合の型を表す．$A\to B$は定義域をA，値域をBとする関数型を表す．*value*…は値の宣言で，その型を型式によって与え，さらにその値自体を関数型，操作型，並行型，公理的などによって記述して与える．

抽象データ型BIRTHDAYBOOKの台はBdb型の値である．これらを特徴付ける演算がnew，add，…である．newは空白の誕生日帳をつくる演算である．addは，有

```
object BIRTHDAYBOOK :
  class
    type
      Voter, Date
      Bdb = Voter →(m) Date
    value
      new : Bdb = [],
      add : Voter × Date × Bdb → Bdb
        add(v,d,bdb) ≡ bdb†[v ↦ d],
      delete : Voter × Bdb → Bdb
        delete(v, bdb) ≡ bdb\{v},
      find : Voter × Bdb → Date
        find(v,bdb) ≡ bdb(v)
          pre v ∈ dom bdb,
      remind : Date × Bdb → Voter-set
        remind(d, bdb)
          ≡ dom[bdb|v:Voter ● bdb(v) = d]
end
```

図 1　関数型，モデル指向

Fig.1　Functional, model-oriented version

権者とその誕生日を誕生日帳に与えて新しい誕生日帳をつくる．delete は与えた有権者の誕生日を誕生日帳から取り除く．find は与えた有権者の誕生日を見つけ出す．remind は与えた日付を誕生日とする有権者の集合を与える．*dom*[bdb|v:Voter ● bdb(v)=d] は，写像 bdb で，bdb(v)=d を満たすものに制限し，その定義域をとったものである．

図 2 は同じ対象を関数型性質指向によって書いた例である．

基本的には抽象データ型の代数的記述法である．型 Bdb を表現していないので歪のない仕様になっているといえる．この場合，値宣言には，関数の型だけを記し，関数値，すなわち，関数の本体は *axiom*…に間接的に定義する．誕生日帳 Bdb の一つの状態 bdb は，new に add を反復したものである．

$$bdb \approx new \cdot add \cdot add \cdots \qquad (*)$$

より正確に書けば，

$$bdb \approx add(v_n, d_n, add(v_{n-1}, d_{n-1}, \cdots add(v_1, d_1, new)\cdots)$$

もちろん，経過としてはこの中に delete や find，remind も挿入されているはずである．delete はそれ以前の add を除去するものだから整理したと考えよう．また，find や remind は bdb の状態を変えないから，結局(*)が成立していると考えていい．このような bdb の状態に対して，それぞれの関数がどう作用するかを考え，そこで成立する関係を等式として表現し公理とする．たとえば，

```
object BIRTHDAYBOOK :
  class
    type
      Voter, Date, Bdb
    value
      error : Date,
      new : Bdb,
      add : Voter×Date×Bdb → Bdb,
      delete : Voter×Bdb → Bdb,
      find : Voter×Bdb → Date,
      remind : Date×Bdb → Voter-set
    axiom forall
      v, v1 : Voter,
      d, d1 : Date,
      bdb : Bdb ●
      delete(v, new)≡new,
      delete(v, add(v1, d, bdb))
        ≡if  v=v1 then bdb
          else add(v1, d, delete(v, bdb))end,
      find (v, add(v1, d, bdb))
        ≡if v=v1 then d else find(v, bdb) end,
      find(v, new)≡error,
      remind(d, new)≡[],
      remind(d, add(v, d1,bdb))
        ≡if d=d1 then remind(d, bdb)∪{v}
          else remind(d, bdb) end,
end
```

図 2　関数型，性質指向

Fig. 2　Functional, property-oriented version

$$\mathrm{find}(v, \mathrm{new}) \equiv \mathrm{error}$$

は，bdb の new という状態に対して有権者 v の誕生日を尋ねると error であるというのである．これに対して，

$$\begin{aligned}&\mathrm{find}(v, \mathrm{add}(v_1, d, \mathrm{bdb}))\\ &\equiv \mathit{if}\ \ v=v_1\ \ \mathit{then}\ \ d\\ &\mathit{else}\ \mathrm{find}(v, \mathrm{bdb})\ \ \mathit{end}\end{aligned}$$

は，ある誕生日帳の状態 bdb に対して，有権者 v_1 の誕生日が d であると登録したとき，その有権者の誕生日を尋ねれば，その誕生日は d であり，そうでない時にはそれ以前に登録した有権者の誕生日を調べる，ということを主張している．これは，(*) を右から左に順に探すといいといっているのである．文面上は，しかし，find と add との間の一つの関係を表現しているにすぎないけれども，気持ちは上のような説明である．気持ちがわかればあとの説明は不要だろう．

図3は，操作型モデル指向による記述である．

```
object BIRTHDAYBOOK:
  class
    type
      Voter, Date
      Bdb=Voter→m Date,
    variable
      bdb : Bdb,
    value
      new : Unit → write bdb Unit
        new() ≡ bdb : =[],
      add : Voter×Date → write bdb Unit
        add(v, d) ≡ bdb : =bdb†[v ↦ d],
      delete : Voter → write bdb Unit
        delete(v) ≡ bdb:=bdb\{v},
      find : Voter ⇝ read bdb Date
        find(v) ≡ bdb(v)
          pre v∈dom bdb,
      remind : Date → read bdb Voter-set
        remind(d) ≡ dom[bdb|v:Voter ● bdb(v)=d]
  end
```

図 3　操作型モデル指向

Fig. 3　Operational, model-oriented version

操作型は抽象データ型を状態機械と考えている．そこでこの状態を変数として保存し，それへの代入文(assignment)とその逐次合成による効果を期待し仕様をつくる．代入文を仕様記述にも採用したのは，最終的には命令型言語によって実現するだろうから設計のある段階(詳細化の一段階)でこの様式によって書くのは便利だろう．また，状態機械モデルはそれに関与する演算あるいは関数の引数が，その状態分だけ節約できることなどからであるとRAISEはいう．

さて，そこでこの様式での演算は，

$$I \times S \to S \times O$$

の型をもつと考える．I や O をとくに必要としない場合にも単位型 Unit がそれであるとして記述する．単位型とは()だけを値とする形式上導入した型である．演算が状態を更新するとき *write* を→の右辺に書く．また，更新しないとき，つまり状態を参照するにとどまる場合には *read* を添える．たとえば，

new : *Unit* → *write* bdb *Unit*
add : Voter×Date → *write* bdb *Unit*
remind : Date → *read* bdb Voter-*set*

のようにである．

図4は，並行型モデル指向の場合である．

```
object BIRTHDAYBOOK :
  class
    type
      Voter, Date,
      Bdb=Voter →m Date
    variable
      find-res-var : Date,
      remind-res-var : Voter-set
    channel
      new-c : Unit,
      add-c : Voter×Date,
      delete-c, find-c : Voter,
      find-res-c, remind-c : Date,
      remind-res-c : Voter-set
    value
      error : Date,
      bdb-p : Bdb ~ in new-c, add-c,  delete-c, find-c,  remind-c
                  out find-res-c,  remind-res-c  Unit
        bdb-p(bdb) ≡
          new-c? ; bdb-p([])
          □
          let (v,d)=add-c? in  bdb-p(bdb†[v ↦ d]) end
          □
          let v=delete-c? in  bdb-p(bdb\{v}) end
          □
          let v=find-c?  in
            if v∈dombdb
            then find-res-c!bdb(v) ; bdb-p(bdb)
            else  find-res-c!error; bdb-p(bdb) end end
          □
          let d=remind-c? in
            remind-res-c!dom [bdb|v:Voter ● bdb(v)=d];bdb-p(bdb) end
    value
      new : Unit → out new-c Unit
        new() ≡new-c!(),
      add : Voter×Date → out add-c Unit
        add(v,d)≡add-c!(v, d)
      delete : Voter → out delete-c Unit
```

```
    delete(v)≡delete-c!v,
  find : Voter→out find-c  in find-res-c  write find-res-var  Unit
    find(v)≡find-c!v; find-res-var : =find-res-c?
  remind : Date→ out remind-c  in remind-res-c  write remind-res-var  Unit
    remind(d)≡remind-c!d; remind-res-var : =remind-res-c?
end
```

図 4　並列型モデル指向

Fig. 4　Parallel, model-oriented version

これは並行プロセス間に一対一の通信路をもち，その同期によって交信するモデルを考えている．通信路 chan に対して，送信側は chan!msg によって通信文 msg を chan に送り出す．受信側は chan?によって msg を受け取る．この機構はホアの CSP から引き継いだものである．誕生日帳の例では，それを管理するプロセス bdb-p があり，これが new, add, delete, find, remind に相当する msg を受信し，管理している状態 bdb を更新，あるいは要求された答えを見つけ出し送信する．$e_1 \square e_2$ は選択演算を示すもので，そのとき外部状況によって e_1 か e_2 のどちらかが選択されるのである．

RAISE によるプログラム作成の第一歩は，仕様記述である．その仕様はだいたい以上のようなものである．仕様記述のために，組込み型としては，論理型，自然数型，整数型，実数型，文字型，単位型，文型などをもち，直積，関数(部分関数)，巾集合，リスト，有限写像，部分型，連合型などを構成できる．モジュール化機構を備え，輸出入，拡張，隠蔽，名前替えなどができるようになっている．またモジュールはオブジェクトかスキームである．オブジェクトは，それが示すモデルの単体であるかまたはその配列である．スキームは記述単位の文面自体を引用するためにクラス式に名前を付けたものをいう．RAISE ではこれらの記述をさまざまに利用する．最初の仕様として，また段階的詳細化のそれぞれの段階の仕様(設計仕様)としてである．RAISE は VDM の延長上の方法であるから，詳細化に対して一つ一つ検証を要求する．ここではこの検証についても省略する．

4. おわりに　　aude et fiet

ソフトウェア開発の形式的方法をどう考え，どう対処するかについて意見を述べたい．まず，どんな形にせよ形式的方法は受け入れるべきである．形式的方法はもちろん万能薬ではないし，この利用によって開発したソフトウェアが完璧なものになるわけでもない．ただソフトウェア開発を科学の規範にのせ教育可能な工学とし，ソフトウェア技術者などといいたいのであれば，形式的方法の採用は避けられない．

ソフトウェアの実務家といっても，いろいろな職種があるだろう．形式的方法を受け入れなければならない職種は，産業構造審議会情報産業部会の情報化人材対策小委員会がいう「プロダクション・エンジニア」である．それは次のように性格付けられている．「現在および今後実用化が予想される広い範囲でのアプリケーションに関して，共通に必要となる基礎的なソフトウェア技術を，単に経験の集積や特定のシステムやアプリケーションに依存したものとしてではなく，理論的な基礎の上に立った工

学として身につけたエンジニアである」[12]．プロダクション・エンジニアの名は周知ではないかもしれない．しかし，このようなエンジニアは実際に必要だろう．この「理論的な基礎の上に立った工学」の一部分に形式的方法があると考えるのである．

話題として三点を選びたい．

1) どのようにして普及するか．
2) 企業へ全面的に導入するならどうするか．
3) どんな勉強をするか．

4.1 普及のために

形式的方法の普及のためには，まず核をつくる必要があるだろう．たとえば主任プログラマ制の主任プログラマ，あるいはプロジェクトリーダとその話し相手に形式的方法を使用してもらう．設計レビューなどでそれが秘儀的でもいいから話し合われるといい．結果として誰の目にも好結果が得られたと写るといい，そこでは名前をもった方法を整然と使うのではない．抽象データ型や交信プロセスが使われれば十分である．その記述は操作型でいい，もちろん，MLやMirandaのような関数型言語やPrologなどの論理型言語が使えればさらにいい，それは，いわゆるアニメーションやプロトタイピングにつながるからである．

抽象データ型と交信プロセスは本質的に重要である．「オブジェクト指向」というキーワードをよく耳にするが，同時に，何をオブジェクトにしたらいいかがわからないとの声も聞くからである．抽象データ型と交信プロセスはこのような状況を一掃するはずである．この二つの概念はブレーンストーミングの鍵になり，非形式的な方法に比べて対象についてより多くの側面を検討でき，分析能力を高める．実現に依存しない仕様が書けることは，たとえば，通勤経路に対して地図の作成を強いることになり，より広い展望が得られるに違いないからである．形式的な検討の結果，自然語で仕様を書いたとしてもそれは確実に改善されているはずである．

ソフトウェアを集団で開発している場合リーダたちの秘儀は必ず要員に盗まれることになるだろう．そこで組織的系統的な教育訓練を実施すればいい．その教育の意味を体験によって知っているからである．もちろん同業の仲間うちにエトスがあっての話であるが．

4.2 組織的な導入のために

一般に新方法の導入のためには，

1) 管理者の理解
2) 実務家の教育訓練
3) 集団作業のための標準の設定
4) 現行方法との共有と移行の方法

などが必要である．すべての項目において困難で否定的な要因があるだろう．管理者の無理解，消極性，低い実務家の職業意識，もともとシステム開発がもっているむずかしさ，新方法の導入計画自体がシステム開発計画以上にむずかしいこと，要員不足や現行作業の遅れから発生する時間のなさ，…挙げれば切りのない要因の数々である．しかし，ビョルナーの言葉を思い出すべきである．形式的方法の使えないソフトウェアハウスは客を失うだろう．そこで次のような実験を行うことを提案する．

目的は形式的方法を採用できるかどうかを評価することである．そのために，

1) ひとりのプロジェクト・リーダを選び，15人月程度の課題を提示してもらう．できればその開発集団の経験した課題がいい．このリーダは顧客を代行し，課題の説明，質疑応答を行う．

2) プロジェクト・リーダと副リーダを二組選び，同じ課題を開発する(仕様作成で止めてもいい)．この二組の要員は形式的方法の未経験者とし，一組には形式的方法の知識・経験のある人を顧問として常時支援を受けさせる．つまり，一組は現行方法によって，もう一組は形式的方法によってシステム開発を行うわけである．

3) オブザーバを選び，二組の開発作業の進行と成果を観察し，評価報告を行う．この実験によって，現行と形式のそれぞれによる開発成果が得られるはずで，それに基づいて形式的方法を評価し，将来計画や管理者への勧告などを得ようというのである．また実験参加者全員による反省会なども興味がもてるだろう．

4.3 何を勉強するか

勉強によって見えなかったものが見え，気付かなかったものに気付くのであれば，何を勉強してもした甲斐があるものだろう．ソフトウェア開発の形式的方法を活用するための勉強といえば，計算科学や情報工学またソフトウェア工学という規範がすでにあり，その教程を学べばいい．それは，しかし，もう一度大学へ戻って学べというのと同じで現実的ではないだろう．現場の実務家のために考えた学ぶべき内容の一例が文献[12]である．しかし，当座のものとしてはまだ多すぎるかもしれない．もっと絞りたい．次はそう考えて選択したものである．

1) 基礎としての数学と論理……仕様作成や正しさの立論のためには，集合と関数，論理と形式的体系についての知識・利用経験が大切である．これらについては数多くの演習が必須である．文献[37],[50]は参考になるだろう．

2) プログラム言語の仕様書……Algol 60, ISO Pascal, Adaなどの仕様書のどれか一つは是非読んでおきたい．それはプログラム言語のより深い理解と仕様書の記述方法についての反省が得られるからである．

3) 関数型/論理型プログラミング……仕様記述のために，このパラダイムによるプログラミングは経験しておきたい．プログラミング教育の最初は関数型で行うべきであるとさえ考える．文献[4]はとくに推薦したい．

4) プログラムの検証……プログラム理論の一部として学ぶよりも，ダイクストラ流ないしはVDM流のプログラミングの教科書から学ぶ方が即物的であると思う．文献[21],[42]は受験参考書風の独習書である．この両者は1)を含んでいる．

5) VDM, Z, RAISE……これらの形式的方法はそのためのいい教科書をもっている．4)などで独習したあとであれば，それらを読破するのはやさしいはずである．再度いいたい．「爲者常成行者常至」．

参考文献 [1] H. Alexander and V. Jones, Software Design and Prototyping using *me too*, Prentice Hall, 1990.

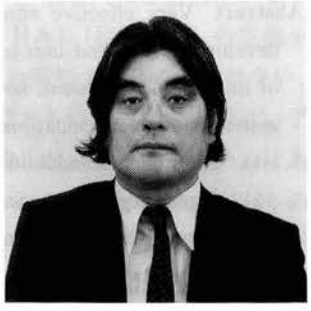
[2] G. Barrett, Formal methods applied to a floating point number system, IEEE Trans. Soft. Eng. SE-15, 611-621, 1989.

[3] F. L. Bauer and H. Wössner, Algorithmic Language and Program Development, Springer-Verlag, 1982.
[4] R. Bird and P. Wadler, Introduction to Functional Programming, Prentice Hall, 1988.
[5] D. Bjørner and C. B. Jones, The Vienna Development Method:The Meta-Language, LNCS 61, Springer-Verlag, 1978.
[6] ibid., Formal Specification and Software Development, Prentice Hall, 1982.
[7] D. Bjørner, et al. (eds.), VDM'87:VDM-A Formal Method at Work, LNCS 252, Springer-Verlag, 1987.
[8] D. Bjørner and O. N. Oest(eds.), Towards a Formal Description of Ada, LNCS 98, Springer-Verlag, 1980.
[9] R. Bloomfield, et al. (eds.), VDM'88:VDM-The Way Ahead, LNCS 328, Springer-Verlag, 1988.
[10] D. Bjørner, et al. (eds.), VDM'90:VDM and Z-Formal Methods in Software Development, LNCS 428, Springer-Verlag, 1990.
[11] S. Brock and C. W. George, RAISE Method Manual, CRI, 1990.
[12] 中央情報教育研究所編, 高度情報処理技術者育成指針プロダクション・エンジニア編, 日本情報処理開発協会, 1988.
[13] CIP Language Group, The Munich Project CIP Vol.I:The Wide Spectrum Language CIP-L, LNCS 183, Springer-Verlag, 1985.
[14] CIP System Group, The Munich Project CIP Vol.II:The Program Transformation System CIP-S, LNCS 292, Springer-Verlag, 1987.
[15] R. Constable, et al., Implementing Mathematics with the Nuprl Proof Development System, Prentice Hall, 1986.
[16] N. Delisle and D. Garlan, A formal specification of an oscilloscope, IEEE Software, 1990 Sept., 29-36.
[17] T. de Marco, Structured Analysis and System Specification, Yourdon Press, 1978, 黒田純一郎他訳, 構造化分析とシステム仕様, 日経マグロウヒル, 1986.
[18] E. W. Dijkstra, Discipline of Programming, Prentice Hall, 1976. 浦昭二, 土居範久, 原田賢一訳, プログラミング原論, サイエンス社, 1983.
[19] E. W. Dijkstra and W. H. J. Feijen, A Method of Programming, Addison-Wesley, 1988.
[20] A. Diller, Z—An Introduction to Formal Method, John Wiley, 1990.
[21] G. Dromey, Program Derivation—The Development of Programs from Specifications. Addison-Wesley, 1989.
[22] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification, 1 : Equations and Initial Semantics, Springer-Verlag, 1985.
[23] ibid., Fundamentals of Algebraic Specification, 2 : Module Specifications and Constraints, Springer-Verlag, 1990.
[24] 淵一博監修, 古川康一, 溝口文雄共編, プログラム変換, 共立出版, 1987.
[25] K. Futatsugi, et al., Principles of OBJ2, Proc. Symp. Prin. Prog. Lang. 1985.
[26] J. B. Goodenough, et al., Toward a theory of test data selection, Proc. Int. Conf. Reliable Soft. 1975, 493-510.
[27] D. Gries, Science of Programming, Springer-Verlag, 1981. 筧捷彦訳, プログラミングの科学, 培風館, 1991.
[28] K. Havelund, An RSL Tutorial, CRI, 1990.
[29] K. Havelund and A. E. Haxthausen, RSL Reference Manual, CRI, 1990.
[30] A. E. Haxthausen, A Tutorial on RAISE, CRI, 1990.
[31] S. Hayashi and H. Nakano, PX : A Computational Logic. MIT Press, 1988.
[32] I. Hayes(ed.), Specification Case Studies, Prentice Hall, 1986.
[33] S. Hekmatpour and D. C. Ince, Software Prototyping, Formal Methods and VDM, Addison-Wesley, 1988.
[34] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[35] 井田哲雄編, 新しいプログラミンン・パラダイム, 共立出版, 1989.
[36] 井田哲雄, 田中二郎編, 続新しいプログラミング・パラダイム, 共立出版, 1990.
[37] D. C. Ince, An Introduction to Discrete Mathematics and Formal System Specification, Oxford, 1988.
[38] M. A. Jackson, System Development, Prentice Hall, 1983. 大野侑郎・山崎利治訳.

システム開発，共立出版，1990.
[39] C. B. Jones, Software Development : A Rigorous Approach, Prentice Hall, 1980.
[40] ibid., Systematic Software Development using VDM, 2nd ed. Prentice Hall, 1990.
[41] C. B. Jones and R. Shaw, Case Studies in Systematic Software Development, Prentice Hall., 1990.
[42] J. T. Latham, et al., The Programming Process. An Introduction using VDM and Pascal, Addison-Wesley, 1990.
[43] K. Levitt, et al., The HDM Handbook, 3 Vols. SRI, 1979.
[44] R. Milne, Semantic Foundations of RSL, CRI, 1990.
[45] ibid., RSL Proof Rules, CRI, 1990.
[46] A. R. J. Milner, Communication and Concurrency, Prentice Hall, 1989.
[47] C. Morgan, Programming from Specifications, Prentice Hall, 1990.
[48] R. Nakajima and T. Yuasa (eds.), The IOTA Programming System, LNCS 160, Springer-Verlag, 1983.
[49] B. Nordström, et al., Programming in Martin-Löf's Type Theory. An Introduction, Oxford, 1990.
[50] B. H. Partee, Mathematical Methods in Linguistics, Kluwer Academic, 1990.
[51] H. A. Partsch, Specification and Transformation of Programs——A Formal Approach to Software Development, Springer-Verlag, 1990.
[52] C. L. N. Ruggles(ed.), Formal Methods in Standards ——A Report from the BCS Working Group, Springer-Verlag, 1990.
[53] J. M. Spivey, The Z Notation : A Reference Manual,Prentice Hall, 1988.
[54] ibid., Understanding Z, Cambridge, 1989.
[55] ibid., Specifying a real-time kernel, IEEE Software 1990(Sept.), 21-28.
[56] J. M. Wing, A study of twelve specifications of the library problem, 1987. CMU-CS-87-142.

執筆者紹介　山　崎　利　治 (Toshiharu Yamasaki)

1957年名古屋大学理学部数学科卒業．同年吉沢会計機(株)入社．翌年日本レミントン・ユニバック(株)へ移籍．著書に，「プログラム言語」(昭晃堂，1989年)，「プログラムの設計」(共立出版，1990年)，共訳書に，M. T. ベルティーニ，Y. タリノー共著「構造的コボル教則本」(TBS出版会，1977年)，M. ジャクソン著「システム開発」(共立出版，1990年)などがある．現在日本ユニシス(株)システム技術本部 研究開発部 主席研究員．



分散システム機能仕様図

A Functional Schema for the Distributed Processing System

佐　藤　　博

要　約　システムの分析，設計作業を有効に行うためには，設計段階でエンドユーザと標的システムについての同一の理解を確立することが不可欠であるが，現在のところ次の条件を満たす設計仕様記述法がない．すなわち，

1)　エンドユーザにも容易に理解できること，できれば図形表現を用いていること．

2)　メインフレームとマイクロ・プロダクトよりなる分散システムの機能分担，タイミングが表現できること．

T. デマルコのデータフロー・ダイアグラム法は上記の2)を満足せず，P. ワードの変換図法は1)を満足しない．

本稿では，変換図法に基礎を置き，上記の条件を満たすように改善した設計仕様記述法を提案している．すなわち，エンドユーザがシステムに対して発行するコマンドや選択肢等の制御情報を表現できる，分散システムにおいて処理が行われるコンピュータ・システムの区別ができる，またそれらの間の起動・受動の関係を表記できる等である．

Abstract　Very effective efforts to analyze and design a system inevitably require both a systems developer and an end-user to mutually have an identical understanding of a targeted system at the stage of its design. At present, however, there are no methodologies or tools available by which to describe systems design specifications which satisfy the following requirements:

1)　High understandability for end-users preferably by representation in charts and graphics

2)　Capability of representing shared functions and timing factors for the distributed processing system consisting of a mainframe computer and micro-products.

The data flow diagram created by T. DeMarco does not satisfy item 1) above, nor the tranformation schema by P. Ward item 2) above.

This paper proposes a new method of describing systems design specifications, which has been so refined on the basis of the transformation schema as to meet both of the two different requirements. The features provided by this method have enabled users to represent such control data as commands and options to be issued to the system, to identify a computer system which operates in the distributed processing system and to describe active/passive relationships in between.

1. はじめに

マイクロ・プロダクトの驚異的なコスト・パフォーマンスの向上は，今後10年は続くと予想されている．安価で省スペースのマイクロ・プロダクトは事務所，店頭，生産現場を問わず，データ処理の「現場」へ今後より以上に導入されていくと予想される．このような「現場」の情報システム化は，その目的から次のように二分される．すなわち，

1)　基幹・定型業務のシステム化

2)　非定型業務のシステム化

である．後者の非定型業務のシステム化についてはエンドユーザ自らが，非手続き言語等を用いてパーソナル・コンピュータ（PC）やワークステーション（WS）をベースに進めていくことになる．

マイクロ・プロダクトをシステム要素とする基幹・定型業務のシステム化は，機能の高度化・複合化をシステム構築の主眼とするので，そのシステム構築には設計の早期の段階からエンドユーザを巻き込む必要がある．また，コンピュータ・ハードやソフトの調達のためコンピュータ・メーカを，またプログラム作成のために外部のソフトハウス等を，開発プロジェクトに参加させる必要がある場合が多い．しかし，これらの「関与者」は目的とするシステムに関して「統一的な解釈（モデル）」が与えられない限り，それぞれ異なる視点から独自の理解（意味，または解釈）を持つことは想像に難くない．合理的にシステムを構築するためには，関与者がそれぞれ抱く解釈を事前に分析，評価，選択しこれらを統合することが必要となる[1]．

そのためには，システム設計の初期の段階で関与者間で構築しようとするシステムについて「統一的な解釈（モデル）」を確立する必要がある．従来のいわゆるライフサイクル・モデルでは，要求定義，概要設計，機能設計，論理設計，プログラミング，機能テスト，システムテスト等の各工程が，あたかも水が滝を落ちるように（ウォータフォール）順序よく実行されるべきである，としていた．そこでは，エンドユーザは，要求定義とシステムテストの工程で参加するのが一般的であった．

その主な理由は，機能設計書・論理設計書等の記述法が一般の人には馴染みがなくわかりにくいことの他に，HOW（実現方法）の定義とWHAT（目標機能）の定義を工程の上でも峻別すべきであって，HOWはエンドユーザの関知するところではない，との考えが底流にあることである．しかし，現実には納期あるいは技術動向等はいずれもHOWと切っても切れない関係にあり，これらを考慮しないでよいシステムはありえない．一般に技術与件は明示的にか暗黙的にかにかかわらずウォータフォールの上部で考慮されている*．

したがって，そのシステムがエンドユーザの業務の中枢に係われば係わるほど，またそのシステムが先端的なものであればあるほどエンドユーザもHOWをも視野に入れてのシステム設計に参画することが，システム構築の成功のためには欠かせなくなる．

たとえば金融トレーディング・システムや新聞原稿入力システム等の高度のマンマシン・インタフェースを要求されるシステムでは，マイクロ・プロダクトをサブシステムとして組み込み，応答性・視認性等の向上を狙うが，ここでは基幹系ホスト・コンピュータとマイクロ・プロダクトの機能分担が設計のポイントの一つになる．この機能分担の設計はHOWに属する事柄ではあるが，応答性，視認性等に大きく影響するからエンドユーザを巻き込んでの十分な事前評価が必要である．また，エンドユーザがHOWについての概略の知識を持つことは，後々のシステムの改良・手直しを計画する際に効率的に議論を進める助けになるだろう．

---

* D.T. ロスらは次のように主張している[6]．すなわち，要求定義は状況分析，機能仕様，設計条件についての課題から構成され，これらはWHAT，HOWおよびWHYという一連の相互に重なり合った応答の一部である．つまり，機能仕様，設計条件のそれぞれについても，WHAT，HOW，WHYが重なっている，としている．

本稿では，エンドユーザにも容易に理解できる分散システム機能仕様図を提示する．これは T. デマルコの構造化分析技法[2]を分散システム向きに拡張したもので，拡張に当たっては P. ワードの変換図法(Transformation-Schema)[3][4]の思想を基礎としている．

2. DFD と統一モデル

2.1 構造化分析の概要

システム設計技法として構造化分析 (Structured Analysis) は重要な位置を占めるが，このためのツールとしてデータフロー・ダイアグラム (DFD) がある．これは T. デマルコによって 1970 年代末に完成された．DFD は表 1 に示す記法を用いてシステムモデルを記述する．

表1 DFD で使用する記号

Table 1 Symbols for DFD

記　号	名　称	説　明
データフロー名 ⟶	データフロー	一つ以上のデータ項目からなる情報の流れを示す．
(処理名)	処　理	入力データフローから出力データフローへの加工または変換を示す．
データストア名	データストア	データの蓄積を示す．
データの発生源名／行先名	データの発生源／行先	分析の対象となる業務の範囲外にあり，データの発生源または行先となる対象

この記法を用いて，従来のウォータフォール型の開発ライフサイクル・モデルと新しい開発ライフサイクル・モデルとをそれぞれ図 1，2 に示す．図 1 のウォータフォール型開発ライフサイクル・モデルの「1.要求定義」と「2.概要設計」は，図 2 の新しい開発ライフサイクル・モデルでは「1.構造化分析」にまとめられている．

すなわち，ユーザの要求は，前者では要求仕様書にそのすべてが封じ込められることが期待されているが，後者の新しい開発ライフサイクル・モデルでは構造化設計の直前までにそれが猶予されている．その意味するところは，エンドユーザを含めたシステム分析チームが WHAT と HOW を一体化して分析を進めることにある．

T. デマルコの提唱した構造化分析の手順を DFD で表し図 3 に示す．すなわち，分析の手順は，「現在の仕事のやり方のモデル化 (現行物理データフロー)」，「それを論理化したモデル (現行論理データフロー)」，「それの改良後のモデル (将来論理データフロー)」，「将来の仕事のやり方のモデル (将来物理データフロー)」等で，そのポイントは「概要から詳細へ」という段階的詳細化にあるのではなく，現行物理モデルから将来物理モデルへの段階的な写像にある．そして，将来物理モデルへの「写像」に当たってはエンドユーザを巻き込んでの代替案の選択に，そのポイントがある．

2.2 DFD の概要

構造化分析には，エンドユーザとシステム開発者が共通に理解し合えるモデルの表記法が必要となる．DFD は次のような特徴により，エンドユーザにも容易に理解でき

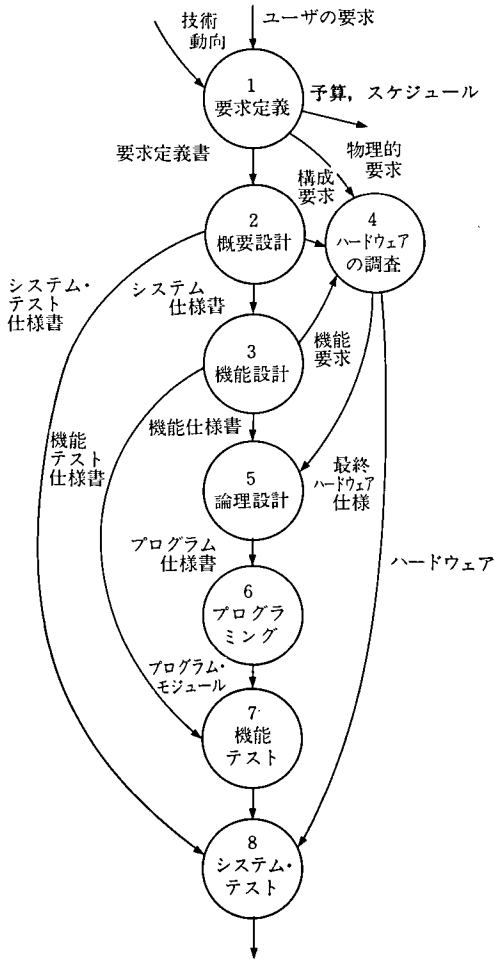


図1　ウォータフォール型開発ライフサイクル

Fig. 1　Waterfall type of development life cycle

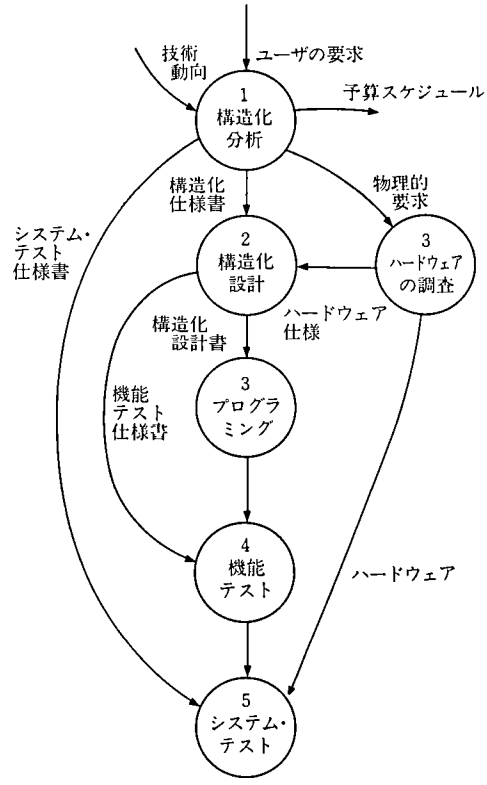


図2　新しい開発ライフサイクル

Fig. 2　New type of development life cycle

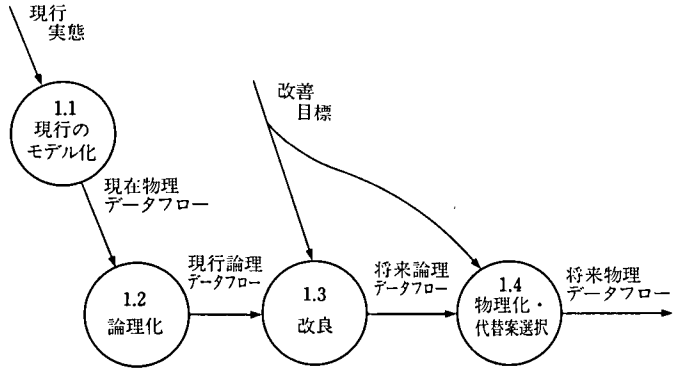


図3　構造化分析の手順

Fig. 3　Procedure of structured analysis

る表記法である[7]．すなわち，

1) 図形表現をする
2) 階層化表現ができる
3) 制御構造は表現しない
4) データの流れを強調している

等である[2]．

図形表現については，表1に示した記号および図1，2の例でその大要が示されている．階層化とは，図4に示したように一つの「処理」が複数の「処理」を纏めて上位のDFDとして表すことをいう．また，図1，2の例で明らかなようにDFDは，矢線に併記するデータ（ファイル，または項目）の流れとその変換ノードとして処理を記述する．その結果，処理の繰り返し，分岐等の「制御（コントロール）」の情報は記述しない．

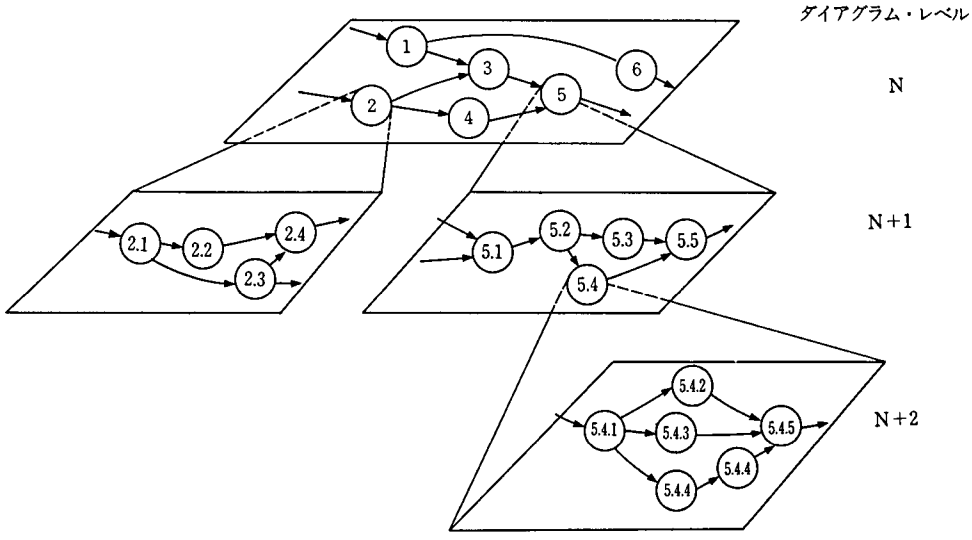


図4 DFDの階層を用いた表現

Fig. 4 Layer structure of DFD

しかし，DFDはこのような特徴のために，分散システム，リアルタイム・システム等のシステム構造の表記には適さない．たとえば，次のようなシステムには適用ができない．すなわち，エンドユーザとコンピュータとの間の制御情報やデータの入出力の順序・分岐・タイミング等，また複数のコンピュータ間の仕事の分担やデータ授受の態様，すなわち一括かトランザクション単位か，等がエンドユーザの使い勝手に重要な影響を与えるシステムである．これらの機能特性は，システム概要設計の段階でエンドユーザを巻き込んで十分に検討されているべきである．

以下にこれらの限界の詳細（3章），そのための改良法（4章）を述べる．

3. DFD の限界

3.1 マンマシン・インタフェースの表記の問題

まず，マンマシン・インタフェース（MMI）とマンマシン境界の相違を明らかにしておこう．DFD 上には手作業処理とコンピュータ化される処理を含んだトータル・システムが記述される．マンマシン境界とは，手作業処理とコンピュータ化される処理との境界をいう（図5）．

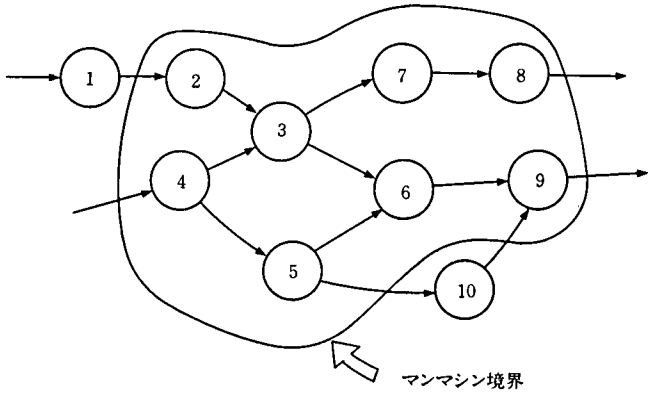


図5　マンマシン境界

Fig. 5　Man-machine boundary

これに対して，MMI はエンドユーザとコンピュータ・システムとの操作上の接触面を指す．

バッチ・システムの場合は，入力伝票をキーパンチ部門に渡す時点がマンマシン境界であり，同時に MMI であった．両者は一致していた．しかし，エンドユーザの作業そのものがコンピュータ・システムとの対話を通して行われるようになってくると，マンマシン境界の内側に MMI が存在するようになった（図6）．

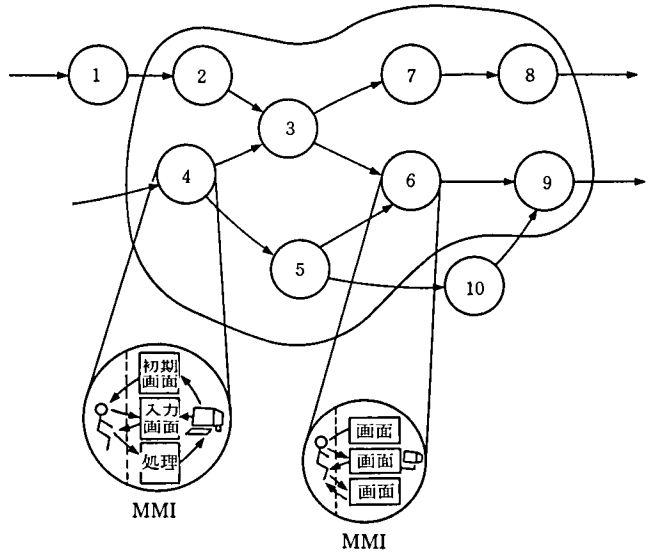


図6　マンマシン・インタフェース（MMI）

Fig. 6　Man-machine interface

そして，データ入出力の容易性，コマンド入力の容易性，応答性等のMMI特性がシステムの有効性を決定する最大の要因の一つになってきた．したがって，MMI特性についてエンドユーザからテスト実施者まで全関与者が，統一的な解釈を共有することが必要になってきている．ここで，統一的な解釈にはHOWも含む．なぜなら，MMIを構成する特性要因のうち重要なものについては，それらに対する重み付け，改善の方策の創出とその評価等に全関与者の参画が望まれるからである．しかし，DFDは機能については表すことができるが，処理の頻度，タイミング，応答性等を実現するHOWについては表現できない．これは，前出図2の表記法からも明らかである．

実務では，一般に「画面遷移図」と呼ばれる表示画面の流れを描いたものをMMIの設計書としているが，これもHOWとの関連が表現できない．

3.2 分散処理システムの表記の問題

分散処理システムは，一般に疎結合と密結合に大別される．疎結合は，一連の作業（たとえば，図7の「処理」2，3，4，5）が一つのサブシステムで閉じて処理され，その終了後にファイル転送で他のサブ・システムで後続の作業（たとえば，「処理」6，7，8，9）が閉じて処理されるようなタイプの分散処理システムである．図7から明らかなようにDFDは，この分散の態様を表現できている．

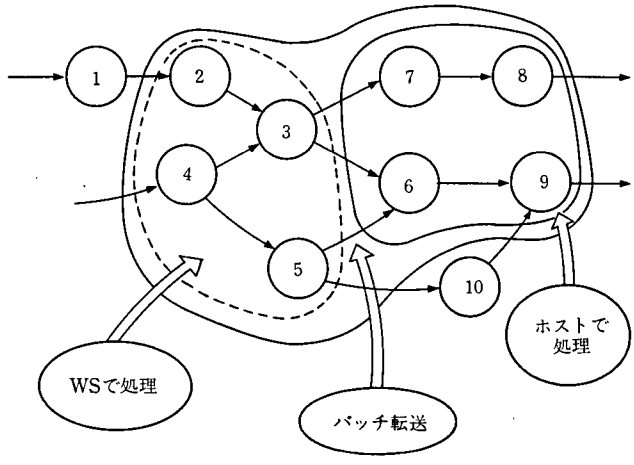


図7 疎結合型分散処理システム

Fig. 7 Loosely coupled distributed processing system

しかし，一つの「処理」がホストとWSで協調しながらなされるような密結合型の分散処理システムの場合には，ホストとWS間の仕事の分担の態様が，DFDでは表現できない．たとえば図8において，「処理4」はWS上でデータ加工を行い，その過程でホストのデータベースを更新するような密結合型の分散システムであるとする．この場合，「処理4」から出る矢線（データフロー）は，ホストとWSの間での「条件分岐」を含むダイナミックな連携処理の結果出力されるだろう．しかしながら，DFDはこのようなダイナミックな連携を表現できない．

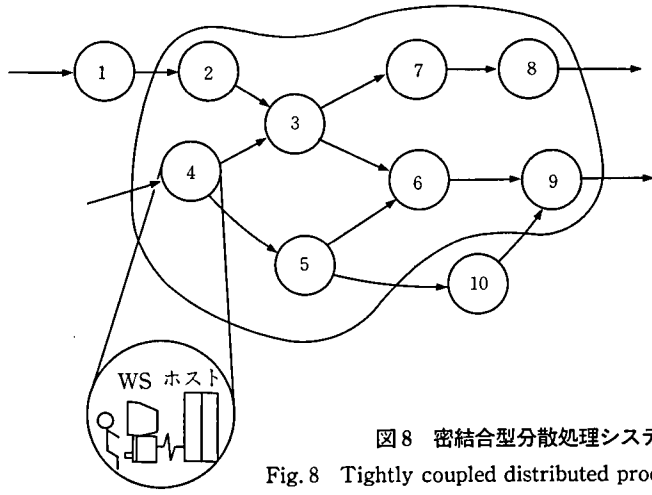


図8　密結合型分散処理システム

Fig. 8　Tightly coupled distributed processing system

表2　変換図で使用される記号

Table 2　Symbols for transformation-schema

記　号	名　称	説　明
——→	不連続なデータフロー	システムがある単位で送受信するトランザクションやデータの集合の流れ
——→	連続なデータフロー	ある時間内で連続で可変なデータやその集合の流れ
(データ変換)	データ変換	DFDの(処理名 プロセス)と同じ
(制御変換)	制御変換	(データ変換)を活性化したり非活性化したりすることで,「データ変換」の働きを制御する.
·····→	シグナル	送信側が(データ変換)あるいは(制御変換)に伝える意志表示
·····→	活性化	送信側が(データ変換)に何らかの出力を生成させようとする送信側の意志表示
·····→	非活性化	送信側が(データ変換)に何らかの出力の生成を中止させようとする送信側の意志表示
データストア	データストア	保存遅延用のデータの保管庫でファイルを抽象化したもの
バッファ	バッファ	(データ変換)により作られた流れを保存するために特殊な保存場所で,変換部分によって消費されるまでの保存遅延の役目を果たす.スタッフやキューを抽象化したもの

4. DFD表記法の拡張

4.1 ワードの変換図

P. ワードは，DFDに拡張をしてシステムの制御の態様，あるいはタイミングを表現できるようにした．このように表現したものを「変換図 (Transformation-Schema)」と呼んでいる．変換図で使用される記号を表2に示す．

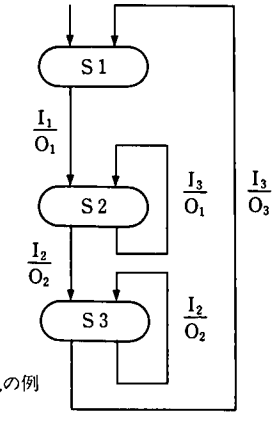
DFDがデータの流れにより機能構造を表現しようとするのに対し，変換図は制御動作により機能構造を表現しようとする．このため変換図は，表2の記号に加えミーリー型の有限オートマン出力表記を導入している．すなわち，制御変換部の制御動作をステートマシンに対応させている．

図9にステートマシン表記の例を示す．すなわち，(a)で［ステートS1］と［入力I_1］のセルは［O_1：S2］であるが，これは「制御変換部はステートがS1である時に，I_1の入力があればO_1を出力してステートをS2にする」ということを表している．図9(b)は(a)と等価で，図形表現したものである．以下に述べるように本稿では図9(b)の図形表現を用いる．

変換図は，システムの制御とタイミングに関する一般的な表記法であるため，構成規則および実行規則が上記に挙げた以外に詳細に規定されている．したがって，これをもってエンドユーザを初めとするすべての関与者に理解できる統一モデルの表記法とするには無理がある．統一モデルの表記法としてここに掲げた表2および図9(b)のステートマシン表記を基本として分散処理システム向きの表記法を定めるのが適当であろう．その内容を次節に述べる．

ステート＼入力	I_1	I_2	I_3
S1（初期値）	O_1：S2	—	—
S2	—	O_2：S3	O_1：S2
S3	—	O_2：S3	O_3：S1

(a) ステート/入力の表形式表現の例



(b) 入力/出力の図形表現の例

図9 ステートマシン表記の例

Fig. 9 An example of the state machine presentation

4.2 変換図を加味したDFDの拡張

3章で明らかになったように，DFDでは次の二つの表現が困難である．

1) マンマシン・インタフェース (MMI) の表記

2) 密結合型分散処理システムの表記

MMIは，人間が自己の側から見たところの，人間と機械系との間の制御情報とデー

タの授受の態様である．また，広義のコンピュータ・システムの構成要素にエンドユーザも含めることができるから，表記上エンドユーザを一つの「制御変換部」と「データ変換部」を備えたサブシステムとして扱うことができる．こうしたとき，MMI の表記は，エンドユーザを制御変換部とした場合の「ステートマシン」に相当する．

エンドユーザがコマンドあるいはデータを入力することにより，WS 上の画面は遷移していくが，これは機械系からの指示ととらえるよりエンドユーザの「ステート」ととらえたほうが設計上は考えやすい．機械系が表示してきた画面は，機械系側の「ステート」ではなく，機械系側が認識している「エンドユーザ側のステート」を表している．(通常のシステムではエンドユーザと機械系側のステートと合致している．)

また，密結合型分散処理システムは，サブシステム相互間で行われる制御情報とデータの授受の態様である．したがって，この場合も各サブシステムは変換図の「制御変換」と「データ変換」を兼ね備えたものとし，かつホストと WS の区別がつくようにすることによりそれらのダイナミックな機能構造が表現できる．

以上のような背景のもとに，DFD に対して次の拡張をする．

1)　DFD の表記法を拡張して表 3 のようにする．(拡張 DFD と呼ぶ)

表 3　拡張 DFD で用いる記号
Table 3　Symbols for extended DFD

記　号	名　称	説　明
データ名 →	データフロー	DFD のデータフローと同じ
1.0 データ変換	WS データ変換	WS 上で実行される「処理(プロセス))」「変換」を示す．
オペレータ	オペレータ操作	WS 側のオペレータの「操作」による「制御」または「処理」「変換」を示す．
ホスト データ変換	ホストデータ変換	ホスト上で実行される「処理」「変換」を示す．
⋯⋯→	データ変換起動	「WS データ変換」あるいは「ホストデータ変換」の起動をさせようとする送信側の意志
WS データストア名	WS データストア	WS 側ファイルへのデータの蓄積を示す．
ホスト データストア名	ホストデータストア	ホスト側ファイルへのデータの蓄積を示す．

2)　オペレータ操作の状態を表現するために変換図法のステート図を拡張導入する．(拡張ステート図と呼ぶ)

この拡張 DFD と拡張ステート図を併用することにより，分散処理システムの統一モデルを表記するツールとする．表記した仕様を「分散処理システム仕様図」と呼ぶ．

以下に表記法の若干の注意を述べる．

①　拡張 DFD は，T. デマルコの DFD を完全に含む．したがってシステムの上位

レベルは，通常の DFD を用いて表し，拡張 DFD を用いなければならない下位レベルの「処理」のみにそれを適用する，ことができる．（例：図 12(a)は図 11 の「処理 2」の詳細図になっている．）

② 拡張 DFD 自体も階層表現できる．この場合，オペレータのステートに影響するなら対応するステート図もまた，階層表現すべきである．

③ 拡張ステート図の各ステートは，WS の「処理」番号に対応させるのが実際的には有効である．そして，下半分には入力画面名（あるいは画面番号）を記入する（図 10）．

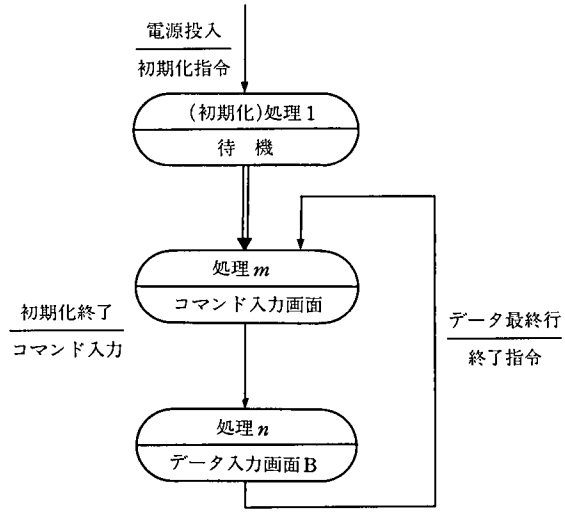


図 10 拡張ステート図で用いる記号

Fig. 10 Symbols for extended state diagram

④ 機械系側の初期化処理あるいはバッチファイル転送が間に入る等のような，オペレータ側の「待機」の状態を表す場合は，ステートとして「待機」と記入する．この場合，次のステートへの移動は機械系側の指示待ちになる．すなわち，ステートマシンに対する「入力」は必要としない．このことを明示するために⇒の追加をする（図 10）．

⑤ 拡張ステート図の入力/出力の「入力」はオペレータに対する操作の「きっかけ」（イベント）であり，「出力」は機械系に対する指示ないしはコマンドの入力である．前者は拡張 DFD 上では，通常オペレータに対する「データフロー」で表される．後者は，「WS データ変換」あるいは「ホストデータ変換」に対する「データ変換起動」に対応する．

5. 例　　題

A 印刷会社はホストと WS からなる「新聞記事原稿入力・校正システム」を運用している．その統一モデルを拡張 DFD と拡張ステート図を用いて以下に表記してみる．

運用の条件：

1)　原稿入力部門は漢字6段シフト鍵盤を装備したWSを複数台保有している．
2)　それぞれのWSに専任のオペレータが配置される．オペレータは渡された原稿に従ってWSから記事の入力を行う．
3)　コード化された記事はホストに蓄積される．
4)　ホストは蓄積されたコード化記事をディスケット媒体に落とし，印刷部門に渡す．
5)　印刷部門は記事ごとにゲラ刷りし，校正部門へ渡す．
6)　校正部門はゲラ上の誤りに「朱」をいれ原稿入力部門へ渡す．
7)　原稿入力部門は朱入りゲラをオペレータに割り当てる．朱のないゲラについては，対応するホスト上のコード化された記事に「校正済み」のマークを付ける．
8)　オペレータはホスト上の対応するコード化記事を読み出して修正をする．後は，上記3)と同じ．
9)　上記8)でホストからのコード化記事の読み出し時間は，記事の量に関係するが90％の記事は15分から30分要するので，その間，次の新規記事入力をする．
10)　校正済みの記事はホストの自動組版システムに渡される．(本システムの対象外)

図11に最上位のDFDを示す．破線の内部がシステム化対象領域であるが，これは処理2と3からなる．そのうち処理2はホスト，WSそれにオペレータからなる分散処理システムである．また，「コード化記事」ファイルも，それがホスト上かあるいはWS上かの区分を明示しないことに注意を要する．

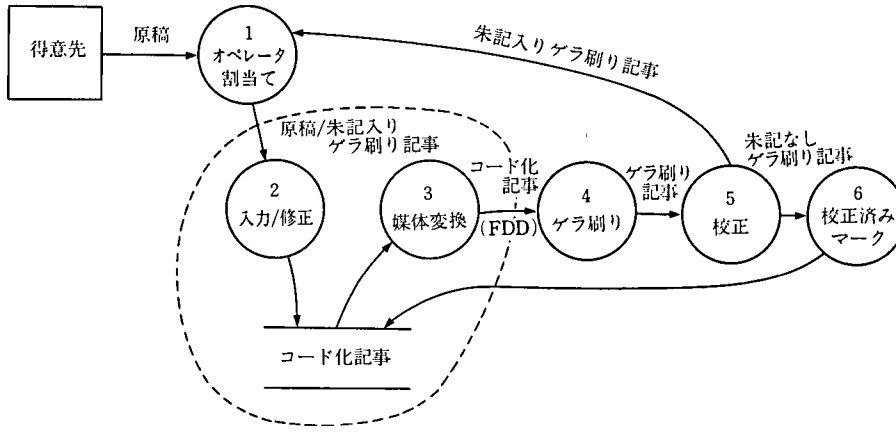


図11　新聞記事原稿入力・校正システム（レベル0＝最上位）

Fig. 11　Newspaper article input and verify system (Level 0)

図12に処理2の詳細を(a)拡張DFDと(b)拡張ステート図を用いて示す．オペレータは，初期選択画面入力によって新規記事入力(処理2.1)かコード化記事読み込み(処理2.2)かの選択をする．処理2.1および2.2はそれぞれ処理終了後，処理2.3を起動する．((a)拡張DFDは2.1および2.2から「プロセス起動フロー」が出ている．)ただし，(b)拡張ステート図は，2.2は記事番号をオペレータが受け取ると，すぐに「停

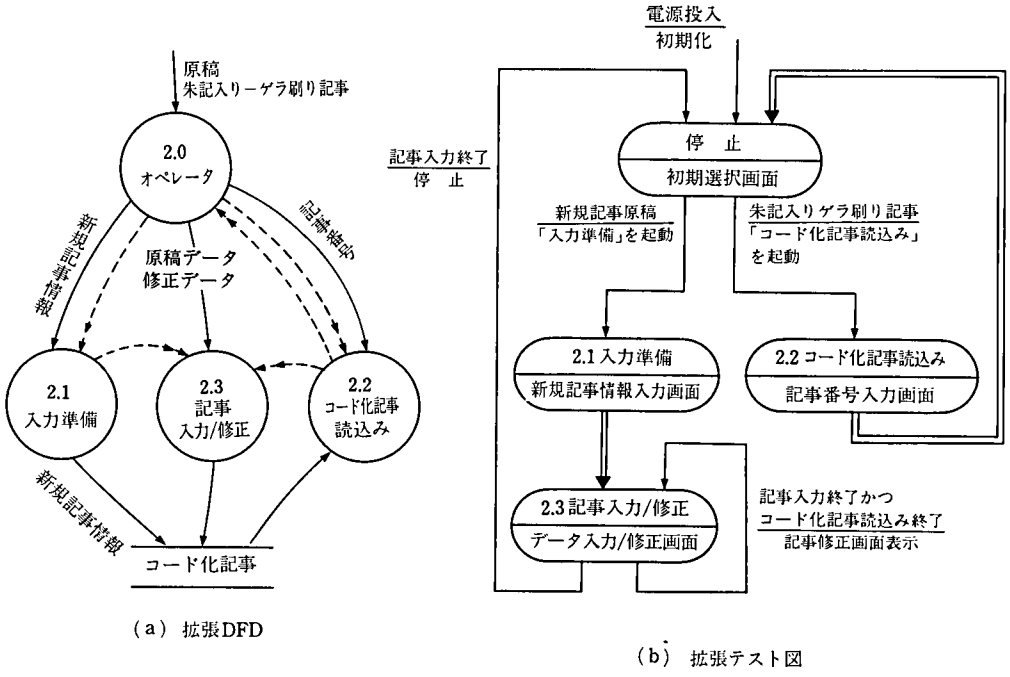


図12 「処理2：入力/修正」の詳細図（レベル1）

Fig. 12 Details of process 2 of Fig. 11

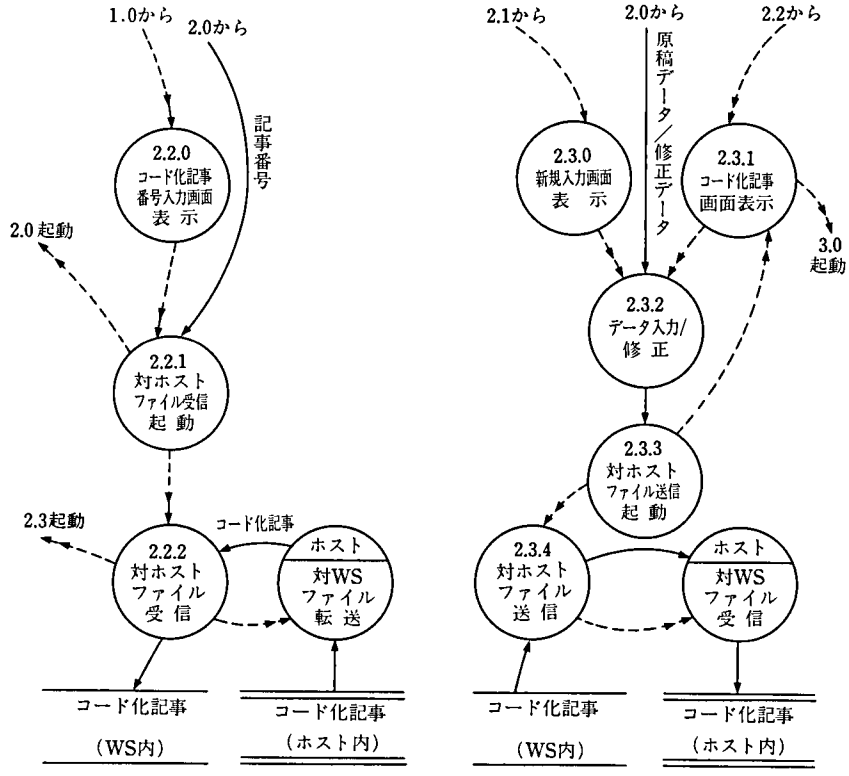


図13 「処理2.2および2.3」の詳細図（レベル2）

Fig. 13 Details of process 2.2 and 2.3 of Fig. 12

止」ステートになり，2.2からの起動シグナルは2.3の記事入力/修正の終了後に有効になることを示している．記事原稿の入力は，(a)では処理2.0から2.3へのデータフローで，(b)では処理2.3のステートで示されている．

図12においてもファイルがホストかWSかの区別はついていない．これに関係する処理は2.2と2.3である．2.2の「コード化記事読み込み」は，対ホストファイル転送を起動する仕事と実際にファイル転送する仕事に分かれ，後者はホスト側のプログラムの起動を伴う．これの詳細を図13に示す．WSの内部ではファイル転送を起動させると2.0を起動させる．また，ファイル転送が終了すると2.3を起動させる．(2.2.2からは対ホストファイル転送の起動と2.3の起動と2本のシグナルが出ている．この順序性を明示するには，2.2.2のステート図を作ればよいが，それをするまでもないだろう．)

2.3の対ホストファイル送信についても，送信の起動をすると2.3.1を起動する．2.3.1は2.2(詳しくは2.2.2から)の起動がかかっていれば2.3.2へ，そうでなければ2.0を起動する．(これも明示するにはステート図を作ればよい．)

6. 考　　察

1) 統一モデル記述ツールとしてDFDを用いて，その中の一つの処理がホストとWSで協調的に行われることを表記するには，DFDの下位レベルとして拡張DFDおよび拡張ステート図が有効である．(DFDとミックスする)

2) エンドユーザにとっては，DFDはほとんど抵抗のない，わかりやすいものである．また，ステート図も画面遷移に対応して理解すれば抵抗は少ないと思われる．

3) システムの事前評価として，ステート図に画面遷移の予測時間（＝ステート移動の予測時間）を記入すれば，より有効な評価になる．

4) ステート図はオペレータから見たモデルである．したがって，システムの内部動作は表現しない．システム内部のタスクの連携のタイミングを明示するには，そのタスクのステート図を作ればよい．しかし，これはあまりに複雑になり勧められない．今後の課題である．

5) ホストが複数である場合には，「ホストデータ変換」の円の上部にホスト名を記入する等の方法が考えられる．

6) E. ヨードンは，システムの複雑さは，機能，データおよび時間に依存した動作という三つの重要な複雑さの次元に帰せられる．そしてシステムのモデル化ツールはこれらの三つの次元のどれかに主軸を置いている，と述べている[5]．

本稿は，機能と時間に依存した動作に照準を当てたツールの提案であり，データ構造が複雑でない，分散処理システムに有利である．

7. お　わ　り　に

統一モデルの表記法が，とくに分散処理システムの分野で重要であることを論じた．次に，分散処理システムの統一モデルの表記法としてDFDは不十分であることを述べ，その対策としてDFDの記法に制御信号を加え，ステート図を拡張したものを併用する方式を提案した．

E. ヨードンも述べているように，今までの設計法では，一つのタスク，一つのマシンを仮定しており，これが大きな問題になっている[5]．とくにマイクロ・プロダクトの基幹業務への本格的な活用時代を迎え，より良いツールの開発が望まれる．

参考文献 ［1］ 松本吉弘,「オブジェクト指向操作的仕様に関する一考察」,情報処理学会論文誌,1988年3月，Vol. 29, No. 3.
［2］ T. de Marco(黒田他監訳),「構造化分析とシステム仕様」，日経マグロウヒル，1981.
［3］ P. Ward, "The Transformation Schema : An Extention of the Data Flow Diagram to Represent Control and Timing", IEEE Trans. 1986 Feb. Vol. SE-12, No. 2.
［4］ P. Ward（立田訳),「ワード氏のリアルタイムSA手法：変換図」，bit 1988 5月，Vol. 20, No. 5.
［5］ E. ヨードン,「見直しの時期にきたシステム構造化技法」，日経コンピュータ，1986. 7. 21.
［6］ D. Ross, K. SchomanJr.（前川訳),「要求定義のための構造的分析」，bit 1978, Vol. 10, No. 10（臨時増刊号―ソフトウェア工学―要求定義)
［7］ 津村泰弘，岡野泰夫,「データフローを中心とした初等SE教育の実践」，情報処理学会，情報システム研究会報告 No. 21-5，1988年11月15日.

執筆者紹介 佐 藤 博（Hiroshi Sato）

1967年武蔵工業大学卒業．同年日本ユニシス(株)入社．9000シリーズ，S 80のスプーリング・システム，アセンブラ・プロセッサ，通信制御ハンドラ等の基本ソフトの開発を経て，1976年よりワークステーション・システムの開発に従事．現在，ワークステーション・ソフトウェア1部長．武蔵工業大学非常勤講師，技術士（情報処理)，中小企業診断士，情報処理学会会員．

自然言語仕様からのデータフロー図構成法

A Method by which to Create Dataflow Diagrams out of Natural-language Specifications

大 野 浩 史

要 約 本稿では，自然語の要求記述からデータフロー図作成のための枠組みと手順を提案する．本研究の目的は，各種のソフトウェア設計法の初期ステップで行われている問題理解の過程を支援することにある．

提案する枠組では，関係表現モデルに従って単文に書き換えられた自然語仕様からデータ間の関係やプロセスを抽出した後，二つのモデル（データ構造のモデル，データの状態変換モデル）に基づいて構造化しながらさらに設計に必要な関係とプロセスを追加，抽出する．抽出したプロセスとデータ間の関係からは，容易にデータフロー図に変換することができる．

本稿では，最後に簡単な例題を用いて適用結果を示し，この方法の有効性を確認する．

Abstract This report is intended to propose the framework and procedures which help to create dataflow diagrams out of specifications written in a natural language. The purpose of this study consists in supporting the process of problem understanding which is required at the initial stage for various software design methods.

The proposed framework begins with extracting both data-to-data relations and processes out of the natural-language specifications rewritten into simple sentences according to the relation representation model, and then goes to the further extraction of more relations and processes required for software designing, while structuring them on the basis of two models——data structure model and data status transform model. The extracted processes and data-to-data relations are easily transformable into dataflows.

By showing the result of an application for which a simple problem was used, this paper tries to make sure of the effectiveness of this method.

1. はじめに

近年，ソフトウェア工学の分野において，上流工程の重要性に対する認識が深まってきている[1]．というのは，ソフトウェア開発過程において，コード化や設計のみならず仕様定義や要求分析が信頼性の高いソフトウェアを作成するために，より決定的な要因となっているからである．

従来より，さまざまなプログラミング方法論や設計法が提案されてきたが，そこでの設計作業とは与えられた課題を，たとえば，HIPOチャート*，N/Sチャート**，データフロー図[2]，ジャクソン木等といった設計文書に変換することである，ということができる．

設計にとりかかる前に要求仕様は明確でなければならないが，多くの場合には与え

* HIPOチャート：1970年頃IBM社により開発されたプログラム図式の一つである．

** N/Sチャート：1973年頃にI. Nassi と B. Shneiderman により提案されたプログラム図式の一つである．

られた課題の記述があいまいであったり，整合性がなかったり，本質的な情報が抜け落ちていたりする．このようなことから，これらの不十分さを取り除くことが可能な問題理解の過程は，設計やコード化よりももっと重要であるといえる．

プログラミングの課題は，通常顧客から自然語で与えられ，作成者はそれに基づいて制限された言語で，要求をより明確に記述する．

Booch[3] の提案したオブジェクト指向設計法においては，与えられた自然語の中からオブジェクト，メッセージ，および属性に関連する名詞や動詞，形容詞を抜き出すというステップを設けている．またジャクソン法[4] においても，その最初のステップで類似した語の抜き出しが勧められている．しかしながら，これらの方法論においては，抜き出すべき語が単にオブジェクトや属性の候補として羅列されるに過ぎず，どれを実際に選択するかについての明確な手順は示されていない．そのため，結局は方法論の専門家でなければこれらの語を後のステップでうまく利用できないという問題点を持っている．

一方，自然言語処理の分野では，文章の意味理解にその研究の重点が移っている．そのため，意味を表現しやすい対象によるモデル化に基づいた機械翻訳[5] や自然言語インタフェースの研究[6],[7] が盛んに行われ，同モデルに基づく自然語仕様によるハードウェアの仕様設計への適用[8] も始まっている．筆者らは，以前よりソフトウェア設計法の初期ステップで行う問題理解についての研究[9]~[11] を進めてきた．

本稿では，最初に自然語の要求記述からデータフロー図の構成に必要な知識を抽出するための要求理解の枠組を提案し，さらに枠組に基づいた手順とその適用例を示す．

提案する枠組では，最初に与えられた要求仕様を関係表現モデルに従った単文に書き換えた後，自然語解析を行うことにより，語句間の関係を抽出する．次に状態モデルに基づくデータの構造化と状態変換モデルに基づくプロセスの構造化を行い，必要な語句間の関係をもれなく抽出後，データフロー図とデータ辞書に変換する．

2章で提案するデータフロー図作成の枠組みを説明する．その後，枠組みで利用するモデルを3章（文の構造モデル），および5章（データの状態モデル），6章（データの状態変換モデル）で説明する．また4章，および7章では各構造化の手順を示す．最後に8章では電報解析問題への適用を示し，その有効性を確認する．

2. 提案する枠組

最初に，データフロー図の構成に必要な知識が次の三つの語句間の関係であると仮定する．

① データ（名詞句）間の関係
　データフローの定義に必要な知識である．

② データ（名詞句）一操作（動詞）間の関係
　プロセス（動詞句）として定義される．

③ プロセス（動詞句）間の関係
　プロセスの構造化に必要な知識である．

要求仕様を読んで理解した内容は，関係表現モデル（後述）に基づいた複数の単文で記述されるものとする．図1にデータフロー図構成の枠組みを示す．まず単文で記

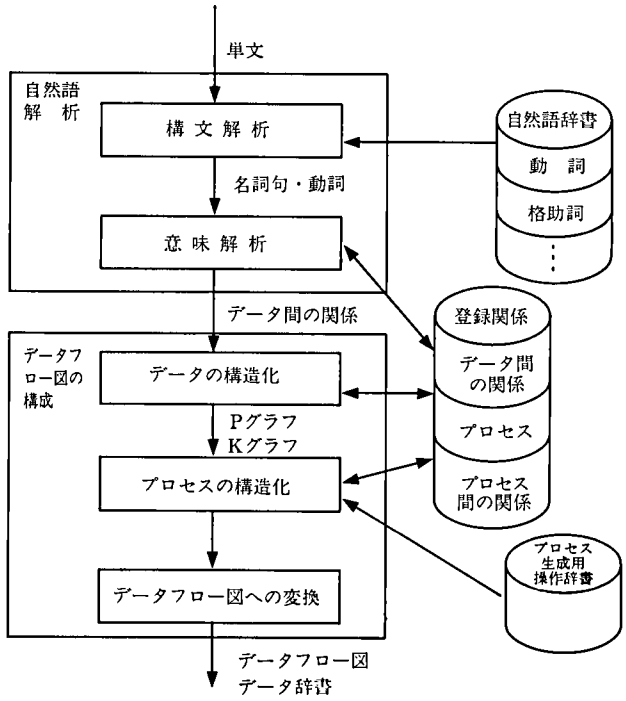


図 1　データフロー図作成の枠組

Fig. 1　Framework of constructing dataflow diagram

述された要求仕様から自然語解析を行うことにより，前記①②の関係を抽出する．与えられた単文を構文解析して名詞句や動詞を抽出し，意味解析によってデータ間の関係やプロセス（動詞句）を抽出する．次に抽出した関係を用いてデータフロー図を構成する．構成はデータの構造化，プロセスの構造化，データフロー図への変換の順に行う．

前記③の関係はプロセスの構造化によって得る．ただし，与えられた要求記述がデータフロー図の作成に必要十分な知識を含んでいるとは限らないので，必要に応じてデータ間の関係やプロセスを追加する必要がある．

また，プロセス追加に際してはデータ構造を無視したプロセスにならないような手順を設定し，そのために必要なプロセス生成用の操作辞書を用意した．

3.　単文の自然語解析

本章では，最初に要求記述に含まれる意味（関係）を単文で表現するための関係表現モデルについて述べた後，単文から抽出される関係の定義について記述する．

3.1　関係表現モデル

関係表現モデルとは，要求仕様の意味を単文で表現するためのモデルで，①単文の表現形式，②単文の構文規則，③単文の最小構成要素(語)の辞書から構成される．

単文の表現形式には次の 2 種類が存在する．

・操作文形式

データ(名詞句)と操作(動詞)の間の関係を記述する形式

・関係文形式

データ(名詞句)間の関係を記述する形式

例文1「電報内の語数を数える」は操作文の，また例文2「電報語は語と空白の記号列から構成される」は関係文の例である．

次に，BNF記法で記述した単文の構文規則を示す．

〈単文〉：：＝〈名詞句A〉〈格助詞〉{〈名詞句A〉〈格助詞1〉}〈動詞〉

〈名詞句A〉：：＝{〈名詞句B〉〈並列助詞1〉}〈名詞句B〉

〈名詞句B〉：：＝{〈名詞句C〉〈格助詞2〉}〈名詞句C〉

〈名詞句C〉：：＝{〈修飾語〉}{〈動詞〉}{〈名詞〉}〈名詞〉

〈修飾語〉：：＝{〈形容詞〉|〈接頭語〉}

この規則に従って，単文「電報語は語と空白の記号列から構成される」を構文解析した例が図2である．

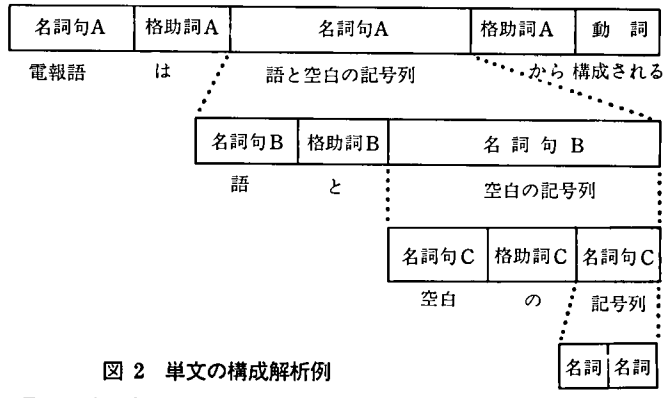


図2 単文の構成解析例

Fig. 2 Example of syntax analysis of simple sentence

また，単文の最小構成要素(語)の辞書（電報解析問題[12]に必要な語のみ）を表1に示す．

表1 単文の最小構成要素(語)の辞書

Table 1 Dictionary of words

語の種類		語の例(電報解析問題)
動詞	操作動詞	取り出す，作成する
	関係動詞	構成される，ある
格助詞	格助詞1	は，から，より，へ，を
	格助詞2	の
並列助詞	並列助詞1	および，と，や
	並列助詞2	または，あるいは
名詞		電報，語，記号列
修飾語	形容詞	長すぎる
	接頭語	各，ある
接尾語		以上，内

3.2 関係の定義

2章で仮定した3種類の関係についてのそれぞれの定義と，その表現形式について述べる．

1) データ(名詞句)間の関係 R_{nn}……データのモジュール化に対応してデータの構成関係を定義し，抽象化に対応して分類関係を定義する．あらかじめ用意した関係の定義を表2に示す．また定義の表現形式を次に示す．

形式：R_{nn}（関係名，(名詞句 X) *，(名詞句 Y) *)

(名詞句 X) *，(名詞句 Y) * は，2個以上の名詞句を要素に持つリストを表現し，要素が一つしかない場合は括弧を省略する．先にあげた関係文の例の場合，関係 R_{nn} は，R_{nn}(has-parts，電報，(語，空白記号列)) となる．

2) プロセス O……データとそれに対する操作の組（動詞句）からプロセス O を定義する．プロセス O を次の形式で表現する．名詞句 Y はプロセスの入力，名詞句 Z は出力を表す．

形式：O(動詞句，(名詞句 Y) *，(名詞句 Z) *)

例として，文「電報内の語数を数える」をもとにプロセス O（電報内の語数を数える，(語)，電報内の語数）を決定する．

3) プロセス間の関係 R_{∞}……プロセス間の包含関係を R_{∞} として定義する．R_{∞} を次の形式で定義する．

形式：R_{∞}(動詞句1，(動詞句2) *)

動詞句1は(動詞句2)* で表現される動詞句のリストの要素を内部に包含する動詞句である．動詞句1と動詞句2がまったく同じプロセスを指す場合は動詞句2の括弧を省略する．例として，動詞句 O_1 が，動詞句 O_2，O_3 を包含する場合，関係 R_{∞} は，$R_{\infty}(O_1, (O_2, O_3))$ となる．

表2　データ間の関係
Table 2　Relation between data

	関　係	定　義	自然語表現
構成関係	Rnn(has-parts, X, (Y_1……Y_n))	X の構成要素の集合＝{Y_1, ……, Y_n}	X はリスト(Y)* の構成要素 Y_1……Y_n から構成される．
	Rnn(has-iter, X, Y)	X の構成要素の集合＝Y の集合	X は Y の集合の構成要素から構成される．
等価関係	Rnn(is-equal, X, Y)	X＝Y	X は Y に等しい．
分類関係	Rnn(is-kind-of, X, Y)	X の集合⊂ Y の集合	Y は X と notX に分類される．
	Rnn(are-kinds-of, (X_1, ……, X_n), Y)	Y の集合＝X_1 の集合∪…∪X_n の集合，かつ $X_i \cap X_j = \Phi$	Y は X_1…… X_n に分類される．

4. 自然語解析手順

4.1 構文解析手順

単文を3章で述べた構文規則に従って解析する.

4.2 意味解析手順

構文解析結果を用いて意味解析を行う手順を図3に示す. 最初に名詞句B, 名詞句Cの構文解析結果を用いてデータ間の関係 R_{nn} を抽出する. 次に動詞の種類を調べ, 動詞が関係動詞である場合は名詞句Aを用いてデータ間の関係 R_{nn}, 操作動詞の場合は名詞句Aと動詞を用いてプロセスを抽出する. 以下それぞれの抽出方法について述べる.

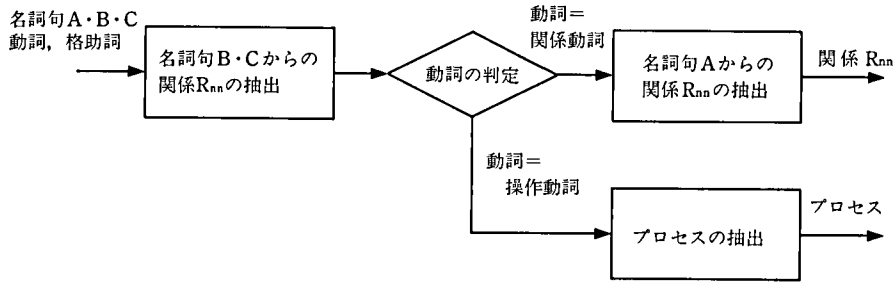


図 3 意味解析手順

Fig. 3 Process of semantic analysis

4.2.1 データ間の関係 R_{nn} の抽出

1) 名詞句B, Cからの関係 R_{nn} の抽出……名詞句からの関係抽出は, 名詞句の構文解析の結果に経験的に得た関係抽出規則を適用することにより行う.

関係抽出規則の一例を次に示す.

関係抽出規則1：名詞句Bの構文解析結果が〈名詞句C〉=〈名詞n1〉〈名詞n2〉であり, かつn2がデータの並ぶ状態を意味する名詞であるとき, 関係 R_{nn}(has-iter, 名詞句B, 名詞句C)を抽出する. たとえば名詞句「記号系列」について説明すると, 構文解析の結果 (n1："記号", n2："系列") に抽出規則を適用することで関係 R_{nn}(has-iter, 記号系列, 記号) を抽出する.

2) 名詞句Aからの関係 R_{nn} の抽出……関係動詞を持つ単文 (関係文) から名詞句Aの2項関係を抽出する. 関係文の持つ関係動詞に応じて抽出する関係を決定し, 関係文内の格助詞を用いて2項を抽出する. 2項抽出のための格助詞は関係動詞ごとにあらかじめ定めておく.

例文「処理の結果は電文, メッセージ, 課金語数から構成される」の場合, 関係動詞「構成される」に対応する抽出関係としてhas-partsを決定する. また抽出用の格助詞「は」と「から」を用いて2項を抽出し, 関係 R_{nn}(has-parts, 処理の結果, (電文, メッセージ, 課金語数)) を得る.

4.2.2 プロセスの抽出

操作動詞を持つ文からプロセス, 出力データを抽出する. 抽出には関係文の場合と

同様に，操作文の持つ操作動詞ごとに抽出用の格助詞をあらかじめ定めておく．例文「電報内の語数を数える」の場合，操作動詞「数える」と出力データ抽出用の格助詞「を」を用いてプロセスO（電報内の語数を数える，()，(電報内の語数)）を抽出する．なお，入力データについてはプロセスの構造化の中で決定する．

5. データの構造化

5.1 データ構造のモデル化

データのモジュール化および抽象化を用いてデータ構造をモデル化する．データをモジュール化するためにデータ間の構成関係に注目し，データを抽象化するためにデータの分類関係に注目する．注目したそれぞれの関係をグラフで表現したもの(Pグラフ，Kグラフ）をデータ構造のモデルとする．以下それぞれのグラフについて説明をする．

5.2 Pグラフ

Pグラフはデータの構成関係を表現するラベル付きの木である．関係R_{nn}(has-parts，(X)*，Y) が成り立つとき，データ (名詞句) YをP集合体とする．また関係R_{nn}(has-iter，X，Y) が成り立つとき，XをI集合体とする．構成関係を持たないデータ (名詞句) はすべて個体とする．データ構造をP集合体，I集合体，個体を用いてグラフで表現したものをPグラフと呼ぶ．

図4にPグラフの例を示す．ある親ノードに接続されている子ノードは，その親ノードの構成要素であることを示す．構成の種類によって，記号P，Iで表現されるラベルを用意する．P集合体とその構成要素間のアークにはラベルPを付記し，I集合体とその構成要素(以後，繰返要素と呼ぶ)間のアークにはラベルIを付記する．ある親ノードとその間にある子ノードとの関係は，必ず複数のP，または一つのIのどちらかで，PとIとの混在を許さない．

5.3 Kグラフ

Kグラフは，I集合体の構成要素である繰返要素の分類関係を表現するラベル付きの木である．図5にKグラフの例を示す．ある親ノードに接続されている子ノードは，その親ノードの分類結果であることを示す．Kグラフの各アークにはKを付記する．

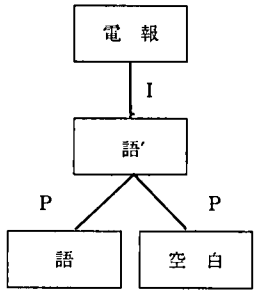


図4　Pグラフ
Fig. 4　P graph

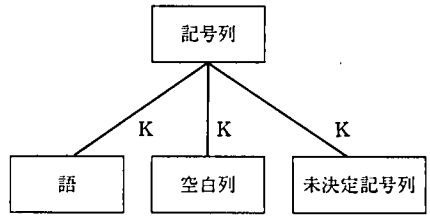


図5　Kグラフ
Fig. 5　K graph

6. プロセスの構造化

最初にプロセスを構造化するための枠組みとしてデータの状態変換モデルを提案後，モデルに従ってプロセス構造を表現するグラフ（Fグラフ）について説明する．

6.1 データの状態変換モデル

プロセス設計の思考過程をモデル化したものをデータの状態変換モデル（図6）と呼ぶ．あるプロセスの設計過程を，入力集合の状態が出力集合の状態に変換される過程であるとみなす．状態の変化する過程を分類過程と変換過程とに分割し，それぞれの過程の操作を分類操作と変換操作と呼ぶ．また，あるレベルの変換過程をその下位レベルの状態変換過程とみなし，モデルを階層的に構成する．

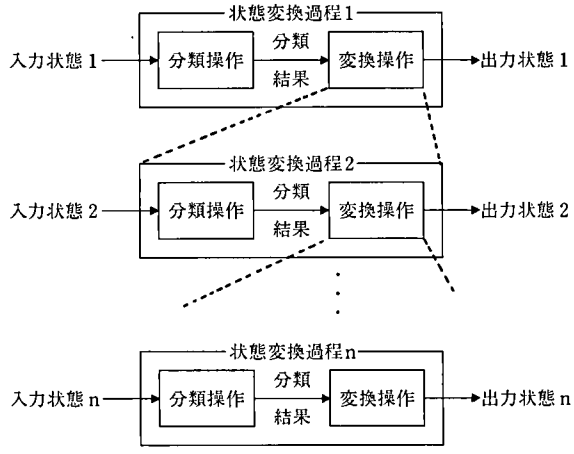


図 6 状態変換モデル
Fig. 6 Data status transform model

6.2 Fグラフ

データの状態変換モデルに基づいて構成した表を変換表と呼ぶ．変換表は，①変換前のデータの状態を表現する状態1，②繰返要素の分類，③変換，および④変換後のデータ状態を表現する状態2の各欄から構成される．状態1，2の欄にはデータがI集合体，P集合体，個体の各種類別に配置される．また繰返要素の分類欄は分類欄と分類結果欄に分けられる．この表を用いて，状態の変換過程を有向グラフで表現したものがFグラフである．図7にFグラフの例を示す．

状態の欄には，状態を表現するための構成要素に対応するデータが配置される．状態1には初期状態および変換途中の状態にあるデータが並び，状態2には変換途中の状態および目標状態にあるデータが並ぶ．また状態分類の欄には分類操作が対応し，変換操作が対応する．分類操作は，あるデータ(繰返要素)を分類するために行う操作であり，変換操作はデータを変換するための操作である．

Fグラフでは，状態の構成要素，分類結果を表す名詞句と状態操作を表す動詞句が節となり，名詞句と動詞句を左から右への向きを持つ線で交互に接続する．またある変換操作は，さらに下位レベルのFグラフに展開される．

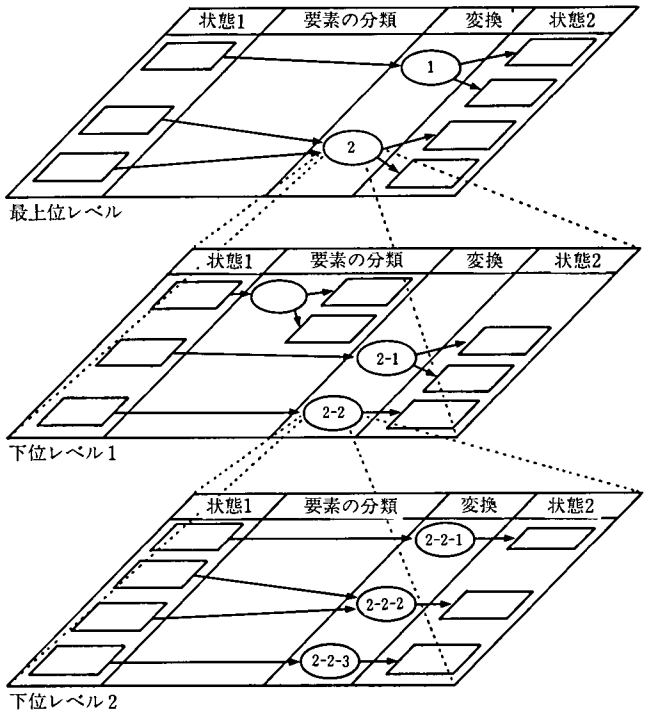


図 7 Fグラフ
Fig. 7 F graph

7. データフロー図の構成手順

7.1 データの構造化手順

7.1.1 Pグラフの構成

単文から抽出した関係のうち，構成関係を用いてPグラフを作成する．あるデータを親ノードとし，その構成要素を子ノードとするアーク群を名前の同じノードを重ね合わせることにより，Pグラフを構成する．アークには，親ノードの種類に従ってPまたはIのラベルを付記する．

7.1.2 Kグラフの構成

単文から抽出した関係のうち，分類関係を用いてKグラフを作成する．関係 R_{nn}(is-kind-of, X, Y) が成り立つとき，XをYとnot Yの二つに分類する．

また関係 R_{nn}(are-kinds-of, $(X_1, \cdots X_n)$, Y)が成り立つとき，Yを $X_1, \cdots, X_n$ に分類する．あるデータを親ノードとした分類結果を子ノードとするアーク群を，名前の同じノードを重ね合わせることにより，Kグラフを構成する．各ノードにはラベルKを付記する．

7.2 プロセスの構造化手順

7.2.1 最上位レベルのFグラフの構成

最初に各Pグラフの根に当たる要素をシステム全体の入出力とみなし，入力に当たるものか出力に当たるものかを決定し，入力要素をFグラフの状態1へ，出力要素を状態2へ位置づける．次に状態1の入力要素に対して，その出力となる要素を状態2

から選び，両者を結ぶ変換操作の操作名を任意に設定する．これをすべての要素に対して行い，システム全体の概要を示すFグラフを決定する．

図8に作成した最上位レベルのFグラフの例を示す．

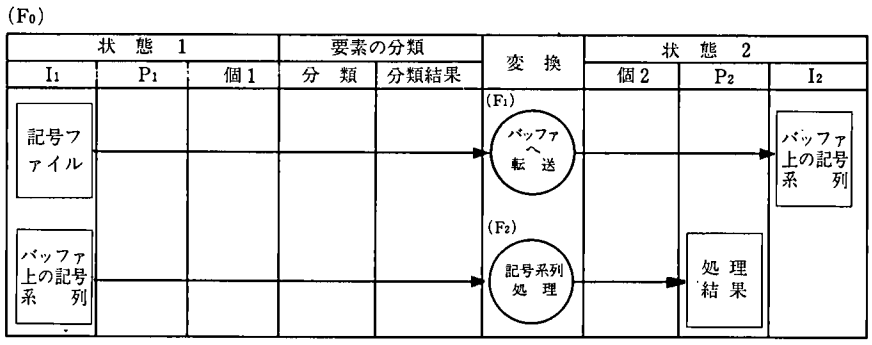


図 8 最上位レベルのFグラフ

Fig. 8 Highest level of F graph

7.2.2 下位レベルのFグラフの構成

構成手順を図9に示す．入出力データの配置を行った後，繰返要素の取り出しを行う．各繰返要素についてプロセスを決定しながらプロセス間の接続を行う．決定に際してはプロセス生成用の操作辞書を用いる．プロセス設定後にFグラフを検証する．以下例を用いて各手順を説明する．

(手順1) 入出力データの配置

レベル1の入出力を見て，対応する入出力データを決定する．「バッファ上の記号系列」から「電報」を作成する場合，状態1に「バッファ上の記号系列」を置き，

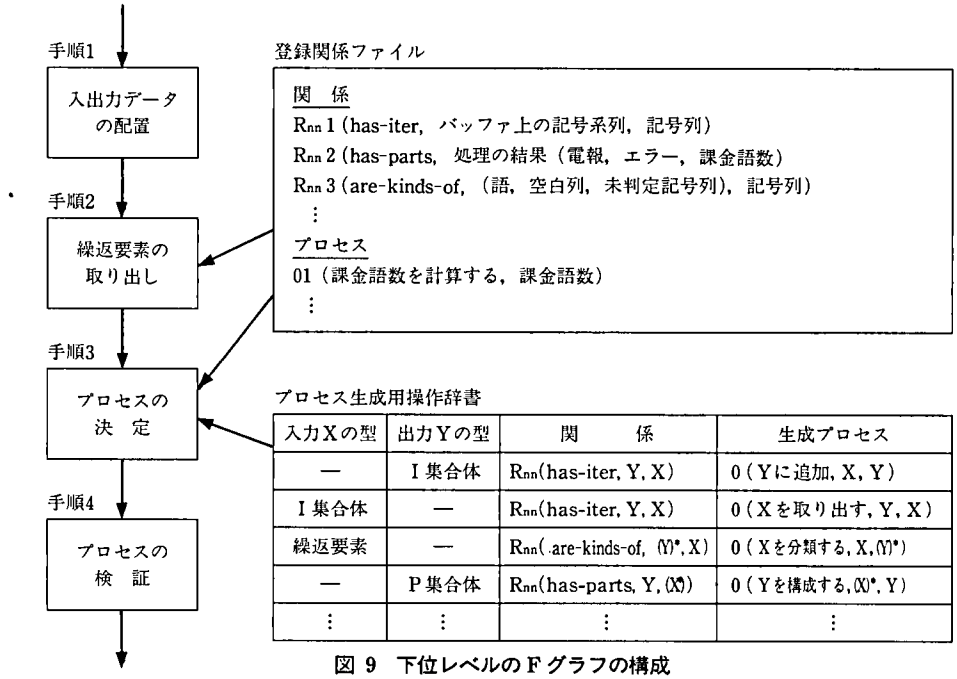


図 9 下位レベルのFグラフの構成

Fig. 9 Constructing lower level of F graph

状態2に「電報」を置く．

(手順2) 繰返要素の取り出し

登録関係より「記号列」は「バッファ上の記号系列」の繰返要素であることがわかっている．したがって，「記号列」を取り扱う繰返要素と決める．

(手順3) プロセスの決定

入力要素から出力要素に向かってプロセスを決定する．入力要素として選ばれた繰返要素に対する分類操作の有無を調べ，あれば分類の欄に位置づける．次に分類結果をもとに，変換操作の入出力を選択する．それに対応する変換操作があるかどうかを，関係対応操作表と計算機操作表から候補を見つけプロセスを決定する．その中にない場合は，適当な名前を命名した上でFグラフのレベルを一つ下げて同様の手順を繰り返す．図10にその例を示す．

(手順4) プロセスの検証

設定された変換操作の検証を行う．検証は，出力要素から矢印をたどりながら入力要素へ向かい，必要十分な入力要素が揃っているかどうかを調べる．例の場合，「電報」を作成するのに必要な入力要素として「空白」が欠落していることが判明したのでFグラフに追加する．また最初に設定したプロセスOと対応する操作との対応をチェックする．

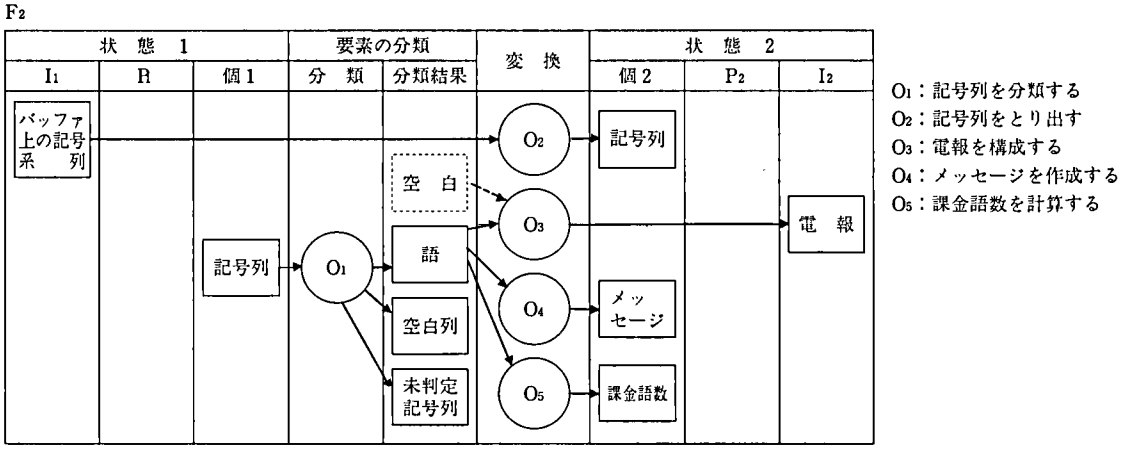


図 10　操作の決定過程

Fig. 10　Process of deciding operations

7.3 データフロー図への変換

できあがったFグラフをもとに，データフロー図を作成する．各レベルのFグラフ内で状態1と状態2に同じデータ名が出てきた場合は，あるプロセスの出力が別のプロセスの入力であると考え，プロセス間を接続することによりデータフロー図を得ることができる．Fグラフの階層をそのままデータフロー図の階層として利用するか，または入出力を合わせながら単階層にして利用する．

データ辞書はPグラフ，Kグラフをもとにデータ辞書の記法に変換することで容易に得ることができる．

8. 適　用　例

本方法の比較的小さな例題への適用結果を示す．

8.1 適 用 問 題

適用問題としてここでは電報解析問題を選んだ．この問題の仕様から与えられる単文を以下に示す．

① 電報処理のプログラムを作成する．
② 電報の流れは記号の系列である．
③ 記号は文字，数字，空白記号に分類される．
④ 定長の記号系列をバッファ領域に転送する．
⑤ 電報は電報語の繰り返しから構成される．
⑥ 電報語は語と空白記号の系列から構成される．
⑦ ZZZZ は電報の区切語である．
⑧ 電報の流れを終了する．
⑨ 空電報は語の存在しない電報に等しい．

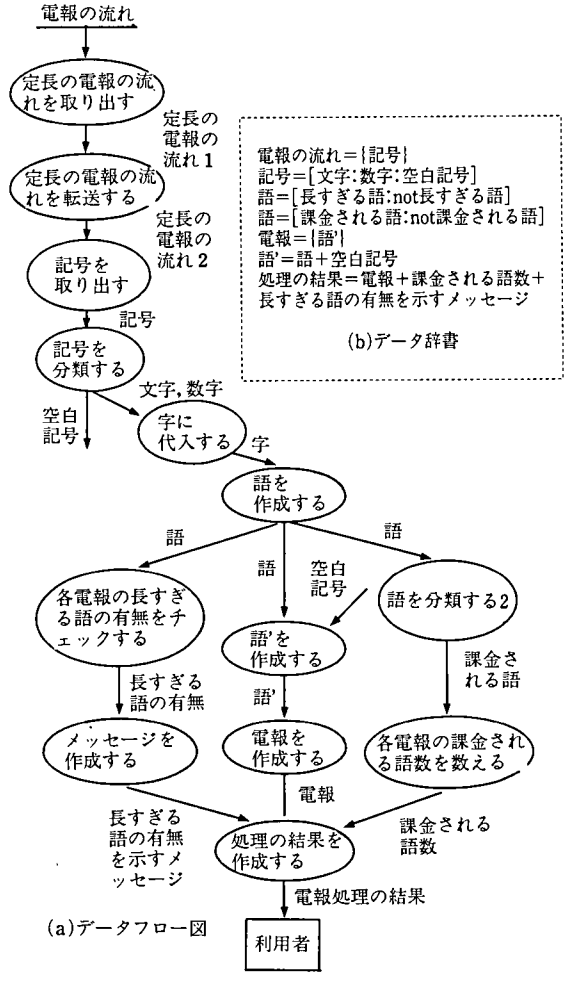


図 11　抽出単文（電報解析問題）
Fig. 11　Extracted simple sentences

⑩ 各電報の長すぎる語の有無を調べる．
⑪ 各電報の課金される語の数を数える．
⑫ ZZZZとSTOPは課金されない語である．
⑬ 13字以上の語は長すぎる語に等しい．
⑭ 処理の結果は電報と課金される語数とメッセージから構成される．

8.2 適 用 結 果

最終的に得られたデータフロー図とデータ辞書を図11に示す．

9. お わ り に

自然語と要求記述に対して，要求定義に必要な情報を抽出するための枠組みと作成過程について述べた．これは設計法の前段階で行っている人間の問題理解，整理に対する一つの仮説であり，客観的な検証が必要である．筆者は，すでに本稿で述べた手順に従って学生実験[11]や上記の課題を処理するシステムの試作を行い，次のような有効性や問題点を確認した．

1) 有 効 性
   ① 人間になじみやすい自然語から，方法論の記法に従った要求定義記述を半自動的に作成できる．
   ② 作業者の方法論の適用能力に係わらず，同程度の品質の要求定義記述（今回の場合，具体的にはデータフロー図，データ辞書を指す）を作成できる．
   ③ 関係情報の他の設計法（ジャクソン法，オブジェクト指向設計法等）への適用が有効であると思われる．
2) 問 題 点
   ① 複文に対する関係表現モデルがないため，複文の場合の自然語解析ができない．
   ② 制御に関する記述を省いてるため，リアルタイムSA(Structured Analysis)には対応できない．

今後は，問題点を解決するためにモデルを拡張する予定である．

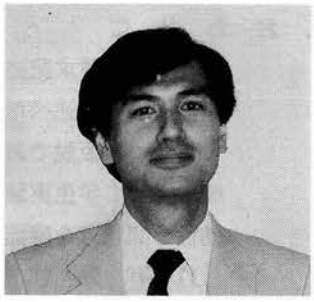
本稿の作成に当たり，多くの示唆を与えて下さった大阪大学基礎工学部情報工学科の鳥居宏次教授，菊野享教授に深く感謝の意を表したい．

参考文献 [1] 中村正弘，“ソフト開発の上流工程を支援する――CASEツールいよいよ日本に上陸”，日経コンピュータ，1989年2月13日．
[2] T. DeMarco, Structured Analysis and System Specification, Prentice-Hall, 1978 (高梨智弘，黒田純一郎 監訳，構造化分析とシステム仕様，日経マグロウヒル社，1986)．
[3] G. Booch, Software Engineering with Ada, Benamin-Cummings, 1983.
[4] M. A. Jackson, Principle of Program Design, Academic Press, 1975 (鳥居宏次翻訳，“構造化プログラム設計の原理”，日本コンピュータ協会，1980)．
[5] 石崎，井佐原，徳永，田中，“文脈と対象世界モデルを利用した機械翻訳へ向けて”，人工知能学会誌，Vol. 4, No. 6, 1989, pp. 660～669.
[6] 加藤，中川，“自然言語インタフェースシステムにおける意図の把握と話題の管理”，情報処理学会論文誌，Vol. 29, No. 9, 1988, pp. 815～823.
[7] 大澤一郎，“オブジェクト指向による自然言語処理”，情報処理，Vol. 29, No. 4, 1988, pp. 390～395.

[8] 中村, 雪下, 打橋, 小栗, "仕様設計エキスパートシステムのアーキテクチャー", 情報処理学会論文誌, Vol. 30, No. 5, 1989, pp. 564～577.
[9] 大野浩史, 菊野亨, 鳥居宏次, "自然語によるプログラム仕様に対する理解モデルの提案", 情報処理学会知識工学と人工知能研究会資料, 88-AI-60, 1988.
[10] 大野浩史, 菊野亨, 鳥居宏次, "抽象化に基づく文章理解モデルのソフトウェア設計法への適用", 情報処理学会ソフトウェア工学研究会資料, 88-SE-62, 1988.
[11] 大野浩史, 菊野亨, 鳥居宏次, "プログラム設計のための構造表現モデルの提案と複合設計への適用", 情報処理学会ソフトウェア工学研究会資料, 89-SE-65, 1989.
[12] K. Torii, Y. Morisawa, Y. Sugiyama and T. Kasami, "Functional Programming and logical programming for telegram analysis", Proc. of 7th ICSE, 1984, pp. 57～64.

執筆者紹介 大 野 浩 史 (Koji Ohno)

昭和53年大阪大学経済学部 経営学科卒業. 同年日本ユニシス(株)入社. 教育部にて客先教育に従事. 62年4月大阪大学基礎工学部 情報工学科受託研究員, 平成2年4月同大学同学部同学科知能情報処理システム寄付講座教員を経て, 現在システム企画本部技術監査室所属. 情報処理学会会員.



会話型プログラムの仕様記述に関する提案

A Proposal on the Specifications Description for Conversational Programs

木 下 博 文

要 約 本稿は，会話型プログラムの仕様記述方法の提案である．近年，プロジェクトの大型化，プログラム開発の外注化の傾向が顕著であり，より正確な設計者—プログラム開発者間の意志疎通が求められている．

意志疎通の主たる手段は，プログラム仕様書であり，その果たす役割は重大である．

提案する方法の特徴は，

1) 会話の手順を状態遷移図により規定する，

2) 状態間の遷移ごとに機能仕様を与える，

3) 機能仕様は入力状態から出力状態への関数として与える，

である．

この方法は，某社でのシステム開発に使用され，システム設計者，プログラム開発者間の意志疎通に有効であった．

本稿は，この方法の枠組み，記述例を記している．

Abstract This paper is intended to propose how to describe the specifications of a conversational program. In recent years, current trends are prominently showing that projects are growing larger and larger in scale; more and more program development is going out to external software houses whereby more precise communications between a program designer and its developer have become what is eagerly called for. The major communications medium is written program specifications, whose role is played very importantly.

The method proposed here provides features such as

1) specifying the dialogue procedures (between a program and a user) in the form of a state transition diagram

2) giving functional specifications for every transition from one state to another

3) using algebraic functions from an input state to an output state to give functional specifications

Practically used for a systems development project at a certain customer, the proposed method has turned out helpful for effective communications between the systems designer and program creators.

This paper illustrates the framework of the method and some of specifications description samples.

1. はじめに

本稿では，会話型処理プログラムの仕様記述の方法について提案する．プログラム仕様書（以下，仕様書と記す）はシステム設計者とプログラム開発者のコミュニケーションの中心的な手段である．

プロジェクトの大型化，あるいはプログラム開発段階の作業の外注化傾向により，仕様書の果たす役割はますます重要となってきている．

仕様記述の方法として，当社には「RSDM (Reliable System/Software Development Method) プログラム仕様記述技法（以下，RSDM と記す）[1],[2]」があるが，次

の理由により会話型処理プログラムの仕様記述には不便さがある．

1) 端末利用者とプログラム間の会話の手順は，プログラム機能仕様の一部と考えられるが，それについて記述する方法がない．

2) 一組の入出力表明によりプログラムの機能仕様を与えている．しかしながら，会話型処理プログラムの場合，プログラムの停止（終了）状態が多岐にわたるため，一組の入出力表明で機能仕様を記述することは困難である．

本提案は「RSDM」をもとにした，会話型プログラムの仕様記述法について述べる．提案する方法の特徴は，次の通りである．

1) 状態遷移図を用いて会話の手順を記述する．

2) 状態間の遷移ごとに機能仕様を与える．

3) 機能仕様は入力状態から出力状態への関数として定義する．

この方法の原型は，小規模ではあるが某社での実際のシステム開発で利用され，その開発はおおむね成功を収めた．提案内容は，実際に適用した方法に手直しを加えたものである．

また，この方法については既存の手法を利用したのみで技術的な独創性はないが，実用的である，適用できる類似システムが多い，ということが本稿による提案の動機である．

2. 基本的なアイデア

以下に，この方法の基本的なアイデアを述べる．

2.1 会話の手順の記述

プログラムが作り出す応答情報と利用者の入力情報のやりとりにより，仕事を進めていく処理形態を会話型処理と呼ぶ．

入力情報は，次の2種類に分類できる．

① 利用者が入力した情報．これを入力コマンドと呼ぶ．

② ①以外の入力画面に含まれる情報．たとえば，画面の見出しテキスト，罫線情報等がこれに当たる．

一方，応答情報も同様に分類できる．

① プログラムが加工・計算して作り出す情報．これを内部変数と呼ぶことにする．

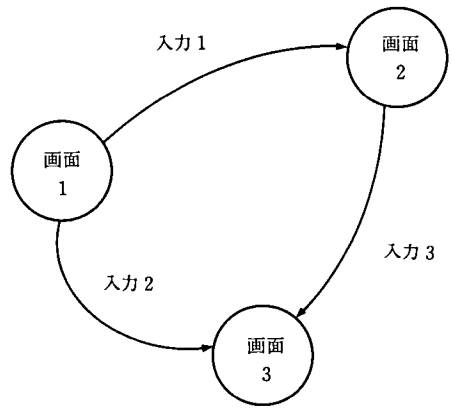
② 入力情報の②と同じであり，画面の見出し情報等である．

入力コマンドの並び（入力順）を形式言語と見たとき，それは正規言語と見做して実用上さしつかえない．正規言語は，一般に有限オートマトンによって受容(accept)される言語であり，状態遷移図により表現することができる．

以上のことから，状態遷移図を用いて会話型処理プログラムが取り扱うべき入力コマンドの並び（すなわち会話の手順）を記述する．

図1に状態遷移の図を示す．

状態は「円」で表され，入力コマンドの入力待ち状態，すなわちプログラムの停止状態を示している．会話型処理プログラムの場合，何らかの応答画面を出力してからプログラムは停止する．したがって，状態はその時点で表示されている画面形式で表



- 〝画面1〟の状態で入力コマンド〝入力1〟が入力されたら〝画面2〟を表示して停止する．
- 〝画面2〟の状態で入力コマンド〝入力3〟が入力されたら〝画面3〟を表示して停止する．
- 〝画面1〟の状態で入力コマンド〝入力2〟が入力されたら〝画面3〟を表示して停止する．

図1　状態遷移図の例

Fig. 1　Example of state transition diagram

すことができる．

状態間の遷移は「矢印」で表され，会話の手順と入力コマンドの種類を示す．

矢印の起点を「入力状態」，終点を「出力状態」と呼ぶことにする．

2.2　機能仕様の与え方

状態間の遷移ごとに機能仕様を与える．

個々の機能は，ある入力状態から特定の出力状態を作り出す一種の関数と見做すことができる．したがって，

$$F：入力状態 \rightarrow 出力状態$$

なる関数で機能仕様を与える．

会話型処理プログラムでは，

- どのような条件が成り立っているとき，どのような入力コマンドを受け付けるか，
- どのような応答情報を作り出すか，
- プログラム停止時に外部記憶（たとえば，データベース）に保存されているべき情報は何か，

が主要な関心事である．

以上のことから入力状態は，

{〈入力コマンド〉，〈画面形式〉，〈入力条件〉}

の組で表す．また，出力状態は，

{〈画面形式〉，〈内部変数〉，〈出力ファイル〉}

の組で表す．ここで〈画面形式〉，〈入力条件〉，〈内部変数〉，および〈出力ファイル〉の内容は次の通りである．

・〈画面形式〉：入出力画面を構成する見出し，項目，項目のデータ型名等の情報（形式情報）を持つものとする．
・〈入力条件〉：条件式を与える．入力条件を満たす時のみ出力状態に遷移するものとする．
・〈内部変数〉：プログラムにより加工・編集された情報を指し，画面形式との組で実際の入力画面・応答画面となる．
・〈出力ファイル〉：入力状態から出力状態への遷移の過程で外部記憶に保存されるべき情報を指す．たとえば，データベースの更新結果がこの情報に相当する．

3. 仕様記述の枠組み

構文と記述内容について述べる．図2にBNF (Backus Naul Form) による仕様記述の構文を記す．仕様書は，背景部・抽象部・詳細部より構成される．

〈program 仕様〉＝〈見出し〉〈背景〉〈抽象部〉〈詳細部〉
〈見出し〉＝**program**〈program 名〉;
〈背景〉＝**背景**〈業務内容〉〈man/machine interface〉〈program の type〉
〈業務内容〉＝**業務内容**〈業務上の用途記述〉;
〈man/machine interface〉＝man/machine_interface〈会話の状態遷移記述〉;
〈program の type〉＝program type〈program の型宣言〉;
〈抽象部〉＝〈型宣言部〉〈入力宣言部〉〈出力宣言部〉〈関数宣言部〉〈機能定義部〉
〈型宣言部〉＝type〈型定義〉|;〈型定義〉| ;
〈型定義〉＝〈レコード型定義〉|〈ファイル型定義〉
〈ファイル型定義〉＝〈ファイル型名〉＝file of〈レコード型〉;
〈入力宣言部〉＝input file〈ファイル宣言〉|;〈ファイル宣言〉| ;
〈出力宣言部〉＝output file〈ファイル宣言〉|;〈ファイル宣言〉| ;
〈ファイル宣言〉＝〈ファイル名〉|, 〈ファイル名〉| :〈ファイル型〉
〈関数宣言部〉＝function〈関数宣言〉|;〈関数宣言〉| ;
〈関数宣言〉＝〈関数名〉:→〈関数型〉|
　　〈関数名〉:(〈引数型〉|, 〈引数型〉)→〈関数型〉
〈関数型〉, 〈引数型〉＝〈データ型〉
〈機能定義部〉＝**機能記述**〈機能記述〉|, 〈機能記述〉| ;
〈機能記述〉＝〈状態 id〉～〈状態 id〉:〈入力状態〉→〈出力状態〉;
〈入力状態〉＝〈入力コマンド〉, 〈display__format__id〉, 〈入力条件〉
〈入力コマンド〉＝〈データ型〉
〈入力条件〉＝〈条件式〉
〈出力状態〉＝〈display__format-id〉, 〈内部変数〉|, 〈内部変数〉|,
　　〈出力ファイル〉|, 〈出力ファイル〉|
〈出力ファイル〉＝〈ファイル型〉 〈内部変数〉＝〈データ型〉
〈詳細部〉＝〈データ詳細〉〈関数詳細〉
〈データ詳細〉＝data detailes〈データの詳細記述〉
〈関数詳細〉＝function detailes〈関数の詳細記述〉

図 2　仕様記述の構文

Fig. 2　Syntax of program specification

背景部は主に利用者のためのものである．会話の手順および業務上の用途の提示により，プログラムの機能について利用者に直感的な理解を与えるように設けられている．

記述は，プログラムの機能仕様を明示するために抽象部と詳細部に分ける．抽象部

が機能仕様に当たり，詳細部はプログラムを開発する上で必要な入出力ファイル等の物理情報を記述する．また，抽象部では入出力ファイルをレコードの集合とみなす．したがって，抽象部での機能記述は集合演算・関係演算を用いて記述してよい．一般のファイル処理であれば集合・関係演算を用いて必要十分にその機能記述ができ，かつ簡潔にそれを表すことができよう．

3.1 抽　象　部

抽象部は，型宣言部・入力宣言部・出力宣言部・関数宣言部・機能定義部から構成されている．型および入出力宣言部では取り扱うデータの仕様を記述し，関数宣言部と機能定義部ではプログラムの機能仕様を記述する．

1) 型宣言部……型宣言部では，プログラムで取り扱う変数，レコード，ファイルの種類を宣言する．

変数は，

変数型　is_型種類

で宣言する．

レコードは，

レコード型　is_record
　変数名：変数型
end_record；

で宣言する．レコード型は変数名を要素に持つ順序対を定義したものとする．

ファイルは，

ファイル型　file_of　レコード型；

で宣言する．file_of はレコード型を要素に持つ部分集合を定義したものとする．

変数の値の参照，代入は，

変数名＝，レコード型．変数名＝，ファイル名．レコード型．変数名＝

でできるものとする．

2) 入出力宣言部……プログラム全体の入出力ファイルを宣言する．入出力ファイルの物理的名称とそのファイル型の対応を明らかにする．

3) 関数宣言部……機能定義で用いる関数を提示する．「関数」は，あるデータ型(型宣言された）から他のデータ型への変換を行う機構である．「編集」・「計算」・「データベースの入出力」等も一種のデータ型変換として捉えることができる．とくにデータベースの入出力については，その物理的な構造に左右されることなく簡潔に表現できる．

関数は，

関数名：引数型 → 関数型
　where　対応関係式

で表す．引数型，関数型は型宣言されているデータ型である．

where 以降に，引数型と関数型の対応関係を記述する．

4) 機能定義部……機能を説明する方法には，手順的に説明する方法と表明による方法がある．ここでは，以下の理由により表明による方法を採用する．

保守フェーズでは，手順すなわち処理ロジックからそのプログラムの機能を読

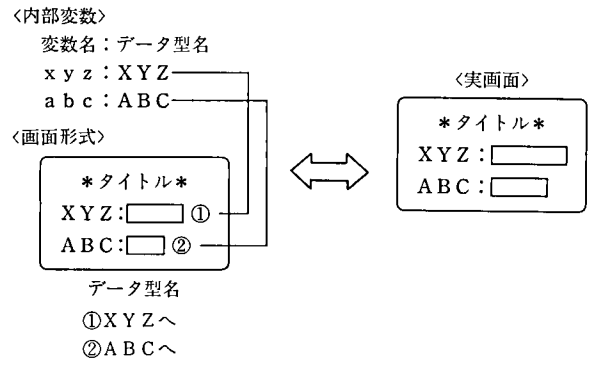


図 3　実画面と画面形式，内部変数間の対応

Fig. 3　Mapping rule between screen, screen information and variables

み取るのは，一般的に困難である．一方，表明による記述では，終了状態が入力との関係において記述されるため簡明にその機能がわかる．また，テスト時にも上記理由によりテストケースの設定・テスト結果の検証が容易となる．

〈入力状態〉は入力表明に対応し，〈出力状態〉は出力表明に対応する．

入力状態は〈入力コマンド〉〈画面形式〉〈入力条件〉の組で，出力状態は〈画面形式〉〈内部変数〉〈出力ファイル〉の組で表すことはすでに述べた．〈内部変数〉の性質を陽に表すために以下の記法を用意する．

変数名：入力状態から出力状態への移行の過程で値の変化しない変数であることを示す．

変数名′：入力状態から出力状態への移行の過程で，計算・加工され値が変化した変数であることを示す．

入力画面・応答画面，画面形式，入力コマンド・内部変数の三者の対応関係を図3に示す．入力時は，入力画面と画面形式内のデータ型情報をもとに同一のデータ型を持つ入力コマンドに入力情報が与えられるものとする．出力時は，画面形式内のデータ型と同一のデータ型を持つ内部変数の値により応答画面が生成されるものとする．

3.2 詳　細　部

詳細部は，実現方式（使用ソフトウェア，ハードウェア）により記述内容が異なってくる．実現環境に合致した情報を記述することとする．

詳細部は，データ詳細と関数詳細から構成される．

1)　データ詳細

たとえば，以下の情報が記述される．

①　画面について：画面図，入出力項目のデータ型名

DPS 1100（Display Processing System 1100；画面制御用ソフトウェア）を使用する時は画面番号，画面名等を提示

②　ファイルについて：使用するファイルハンドラ，編成方式，キー項目の変数名

レコード型のフォーマットおよび登録集名等

2) 関数の詳細

以下のような情報が記述される．

・引数と関数型（関数の出力）の演算

・必要であれば，使用するデータベース・インタフェースルーチンの指定

4. 仕様記述例

本章では，住所録管理のプログラムを例題として仕様記述の例を示す．

4.1 要　　件

要件として，以下の項目があげられた．

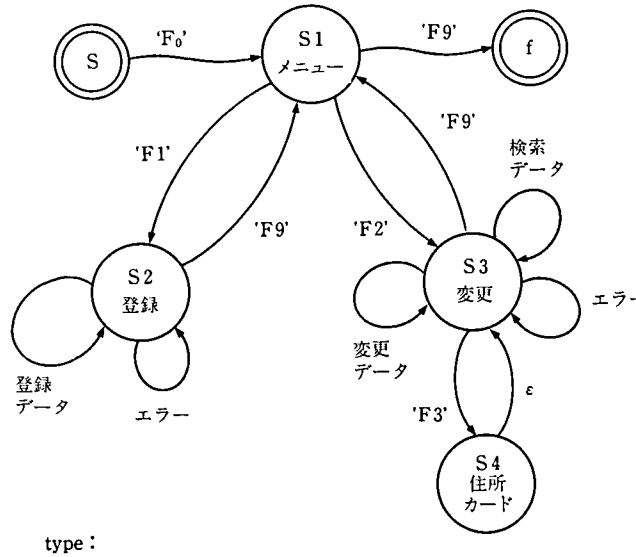
VDT 画面を通して知人の氏名，住所，電話番号（以下，知人情報と略）を記録・変更・検索し，必要であれば宛先カードを印書するプログラムを作成したい．検索は，氏名の読みで行いたいので，ふりがなをキーにして知人情報を蓄積してほしい．

変更は，事前に検索機能により旧内容を表示しておき，それに対し修正できるようにしてほしい．変更済および検索結果画面から，宛先カードを印書したい．

4.2 記　述　例

以下に，住所管理プログラムの仕様記述例を示す．

```
program：住所録；
背景：
    知人の名前，住所，電話番号を記録する．
    記録内容から，郵便に使える宛先カードを印書することができる．
man/machine_interface；
```




```
type：
    氏名かな is character string；
    氏名，住所，電話番号，処理結果 is character string；
    住所録データ is record
      j_氏名：氏名
      j_住所：住所
      j_電話：電話番号
```

```
    end_record;
    登録データ is record
      r_氏名かな：氏名かな
      r_住所録データ：住所録データ
    end_record;
    変更データ is record
      c_氏名かな：氏名かな
      c_住所録データ：住所録データ
    end_record;
    住所録マスタ is record
      m_氏名かな：氏名かな
      m_住所録データ：住所録データ
    end_record;
    住所録ファイル is file_of 住所録マスタ;
input_file：JF：住所録ファイル;
output_file：JF：住所録ファイル;
function：
    who：氏名かな，住所録ファイル→住所録マスタ
      where 住所録マスタ＝氏名かな◁住所録ファイル
    is_exist：氏名かな，住所録ファイル→ boolean
      where #{氏名かな◁住所録ファイル}＝0 → false
            #{氏名かな◁住所録ファイル}＞0 → true
    str_住所録：登録データ，住所録ファイル→住所録ァイル'
      where 住所録ファイル'＝住所録ファイル U 登録データ
    mod_住所録：変更データ，住所録ファイル→住所録ファイル'
      where 住所録ファイル'＝住所録ファイル⊕
                          {変更データ．c_氏名→変更データ．c_住所録}
機能記述：
    *メニュー表示*
    S～S1   :{"F0", ε, ε}→{S1, ε, ε}
    *登録画面表示*
    S1～S2  :{"F1", S1, ε}→{S2, ε, ε}
    *知人登録*
    S2～S2  :{登録データ，S2，is_exist(r_氏名かな，JF)＝false}
             →
             {S2，(登録データ，処理結果＝'complete'),
                  str_住所録(登録データ，JF)}
    *登録エラー*
    S2～S2  :{登録データ，S2，is_exist(r_氏名かな，JF)＝true}
             →
             {S2，処理結果＝'already exist'，ε}
    *変更画面表示*
    S1～S3   :{"F2", S1, ε}→{S3, ε, ε}
    *変更*
    S3～S3  :{変更データ，S3，(is_exist(c_氏名かな，JF)＝true
                              ∧c_住所録データベース≠スペース)}
```

→

{S3, (変更データ, 処理結果='complete'),
mod_住所録 (変更データ, JF)}

＊検索＊

S3～S3 ：{変更データ, S3, (is_exist(c_氏名かな, JF)＝true
∧c_住所録データ＝スペース)}

→

{S3, who(c_氏名かな), ε}

＊変更検索エラー＊

S3～S3 ：{変更データ, S3, is_exist(c_氏名かな, JF)＝false}

→{S3, 処理結果='not known', ε}

＊カード印書＊

S3～S4 ：{"F3", c_氏名かな, S3, ε}→{S4, who(c 氏名かな), ε}

S4～S3 ：{ε, S4, ε}→{S3, ε, ε}

＊メニュー表示＊

S2～S1 ：{"F9", S2, ε}→{S1, ε, ε}

S4～S3 ：{"F9", S3, ε}→{S1, ε, ε}

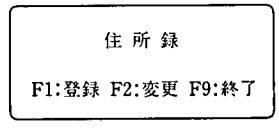
＊終了＊

S3～f ：{"F9", S1, ε}→ε

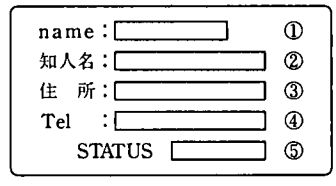
data_detailes：

JF ：編成方式は ISAM.
key は氏名かな.
以下省略.

S1 ：画面形式は以下の通り.

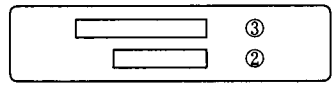


S2, S3 ：画面形式は以下の通り.



型 名
① 氏名かな
② 氏名
③ 住所
④ 電話番号
⑤ 処理結果

S4：画面形式は以下の通り. 出力媒体はプリンタ.



③ 住所
④ 氏名

function_detailes：

省略；

例題中に説明を行っていない記号が現れているが, これらは仕様記述言語 Z[3] で用いられているもので, 意味は以下の通りである.

◁：domain restriction

たとえば,

1 ◁{(1, a), (2, b), (3, c)} のとき, 結果は {(1, a)}

となる．

#：cardinality

#Sは，集合S内の要素の数である．

⊕：overriding

たとえば，

{(1, a), (2, b)}⊕{(1, d),}は{(1, d), (2, b)}

となる．

x → y：(x, y)の順序対を表す．

なお，詳細な定義は参考文献[3]を参照されたい．

5. おわりに

一般に複雑な課題は，より小さな課題に分解したほうが考えやすい．状態遷移図の導入により，プログラムの機能が状態間の遷移で表せる小さなプロセスに分解される．その結果，プロセスごとの機能仕様が書きやすく，また読みやすいものとなる．

また，会話の手順はプログラムの使い勝手の優劣を決める重要な要素である．システム設計者はその決定に当たり利用者の了解が必要となる．状態遷移図のような単純な図式は，利用者にとってもプログラム機能の理解の助けとなる．

この仕様記述法は，いわばプロジェクト内でのみ通用するローカルなものである．一方，近年仕様記述言語につき国際標準化の動きがある．たとえばVDM，Zがそれであり，VDMはすでにBSI標準となっている．われわれシステム開発を担当するものとして，プログラム言語と同様にこれらの言語を使えるようになることは，開発者間のコミュニケーション向上にも役立ち大いに歓迎するものである．これらについて，勉強の必要性を痛感する．

後に，本稿がプログラムの仕様記述を担当するシステム設計者の方々に何らかの参考になれば幸いである．また，本方法の適用の初期の段階から報告書のまとめに至るまで，生産技術部の染谷氏に貴重な意見をいただいた．また，研究開発部の山崎氏には，多くの改善点を指摘していただいた．この場をかりて感謝の意を表したい．

参考文献
［1］ RSDMプログラム仕様記述技法，日本ユニシス(株)，昭和58年9月．
［2］ プログラム仕様記述技法，日本ユニシス(株)教育部，1984.
［3］ D.C. Ince, An Introduction to Discrete Mathematics and Formal System Specification, Oxford Applid Mathematics and Computer Science Series.

執筆者紹介 木 下 博 文 (Hirofumi Kinoshita)

昭和46年 東京工業専門学校 電気工学科卒業．同年日本ユニシス(株)入社．製造工業，航空，旅行業のアプリケーション開発に従事．現在，日本ユニシスソフトウェア(株)社会公共統括部に所属．

通信ソフトウェア開発効率化の手法

An Approach to Improved Productivity in Communications Software Development

宮坂順之

要　約　1980年代末以来，ISOによるOSI(Open Systems Interconnection)，CCITTによるISDN(Integrated Services Digital Network)の標準化が精力的に行われてきた．標準化は下位のレイヤより順次進められており，現在は上位レイヤの標準化が行われている．これまでに，これらの機関が作成した標準ドキュメントの量は，10万ページを越えている．

これらの標準は最終的にホスト，通信制御装置，ワークステーション，パーソナルコンピュータ等の通信ソフトウェアとして実装される必要がある．現用の方法を用いて実装した場合，多大のマンパワが必要である．マンパワを少なくし，保守を容易にするためには，通信ソフトウェアの効率的開発手法の確立が必須である．

本稿では，プロトコル開発言語，テストシステムの自動化等，通信ソフトウェアの開発効率化のためのツールの実現方法について記述した．これを用いることにより，開発要員の削減が可能であり，かつ機能拡張等の変更に容易に対応できるであろう．

Abstract　Since the second half of the 1980s energetic efforts have been made to standardize the Integrated Services Digital Network (ISDN) by CCITT and the Open Systems Interconnection (OSI) by ISO. Those standardization efforts started with lower-layer protocols and now are under way for upper-layer standards. The standard documents assembled by those organizations up to now have reached no fewer than 100 thousand pages in volume.

Such standards need to be finally implemented on host systems, communications control units, workstations, personal computers and the like for use as communications software. Current ways of developing software, if applied for production of the software for those types of eqipment, would very likely require a great deal of man-power. For the sake of less development man-power and easier product maintenance, it is essential to establish a new approach which addresses the efficient development of communications software.

This paper describes how to implement tools for improving productivity in the developing of communications software packages including those for a protocol development language, automated test systems, and so forth. The author believes that the use of what is written here would bring about a reduction in man-power and make it possible to respond quickly to requirements for product modifications like functional expansion.

1. はじめに

本稿では，OSI/ISDNプロトコル等を実現する通信ソフトウェア* 開発の効率化技法について記述する．

---

* ここでの通信ソフトウェアは，OSIのレイヤ1〜7のプロトコル，BSC等のベーシックプロトコルを実現するソフトウェアを意味する．以降これをレイヤプログラムと呼ぶ．
ISO：(International Standard Organization)，OSI：(Open Systems Interconnection)開放型システム間相互接続
ISDN：(Integrated Services Digital Network)統合デジタル通信網

1.1 背　　景

現在，コンピュータ通信を取り巻く環境は急速に変貌しつつある．

今までの通信システムは，メーカが独自に通信のアーキテクチャを開発し，それに基づいてホストおよびターミナルシステムを提供してきた．しかし，現在ではマルチベンダ環境での相互接続性が強く求められており，今後は国際標準に基づいたターミナル，ホストシステムの提供が強く求められるようになってきた．

この要求に応えるため，現在ISOを中心にOSIの開発が進められている．メーカは，今後4～6年かけて独自のアーキテクチャをOSI標準のアーキテクチャに切り替えていくことになろう．また，現在のOSIは機能的にも未熟であり，今後3～4年間は，ネットワーク構成の自動化機能，ネットワーク管理機能，セキュリティ機能等の機能の充実をはかっていくことになろう．したがって，メーカはこれを継続的にフォローしていく必要がある．

これらに対応していくためには，膨大なマンパワが必要である．このような負荷をできるだけ少なくしていくためには，プロトコル開発の効率化，ソフトウェアの変更のしやすさ，および移植を可能とすることが重要である．

1.2 目　　的

通信ソフトウェア開発効率化のためには，以下に示す項目を実現する必要がある．

① レイヤプログラムの部品化/共通化を実現する．

レイヤプログラムとは，OSIレイヤ2～7を実現した通信ソフトウェアであり，プロトコルごとに独立したソフトウェアモジュールとして作成される．部品化は，ハンドラが共通に使用する処理をファクションまたはマクロライブラリとして提供することであり，共通化とは，ハンドラを異なるプロダクト間で共通に使用できるようにすることである．

② レイヤプログラムの記述の標準化を推進する．

③ レイヤプログラムを移植可能にし，プロダクト間で共通に使用できるようにする．

④ プロトコル仕様書に近い形でソフトウェアの記述が可能な言語の提供．

⑤ テストシステムによりレイヤプログラムテストの自動化/省力化を実現し，開発期間の短縮を図る．

⑥ ネットワーク管理インタフェースを仮想化し，今後の機能追加・変更等を容易とする．

⑦ ディレクトリおよび構成テーブル形式記述言語の提供により，データベースアクセスを仮想化し，外部で定義するコンフィグレーション情報とレイヤプログラムが使用するテーブルとの統合化を実現する．

ディレクトリとは，通信のエンティティ(回線，アプリケーション等)の関連および属性を記述したファイルである．レイヤプログラムはディレクトリを利用することにより，トランスポートパス等を設定するとき，どの経路を使用すれば良いか決定できる．

本稿では，これらの実現方法について，具体的に記述する．

2. システムの概要

本開発効率化を具体化したシステムを，仮想プロトコル機械(VPM：Virtual Protocol Machine)システム(以降 VPM システム)と呼ぶ．VPM システムは，プロトコル開発の自動化，効率化を具現化するソフトウェアである．

2.1 システムの構成

本 VPM システムは，次の五つのソフトウェアより構成される(図 1)．

1) VPM……VPM システムの中心のプログラムでレイヤプログラムに対して各種の機能を提供する．これらの機能は標準化されており，プロトコルの種類，ハードウェア，プログラムの置かれる環境には依存しない統一したインタフェースを提供する．したがってプログラムの製作者は，その動作環境等の知識は必要なく，プロトコルの知識があれば容易にレイヤプログラムを作成することが可能となる．

2) プロトコル作成支援プロセッサ……プロトコル作成者を支援するプロセッサで，レイヤプログラムの製作者と会話形式でインタフェースし，プロトコル仕様書に近い形でレイヤプログラムの記述を可能とするプロセッサである．VPM が使用する各種のデータベース・制御テーブル等の生成，コンフィグレーションプロセッサの使用するネットワーク構成のためのテーブル定義書，およびレイヤプログラムの本体であるプロセスコード(後述)・アクションコード(後述)を生成する．

3) テストシステム……テストシステムは，レイヤプログラムのユニットテスト，およびシステムテストを会話形式にて行うためのソフトウェアで，テストの自動化を実現しテスト期間の短縮を可能とする．

4) コンフィグレーション・プロセッサ……コンフィグレーション・プロセッサは，レイヤプログラム等が定義したテーブルの形式，テーブル相互間の関連に基づき，実際の物理的・論理的ネットワークの構成およびディレクトリを会話または一括

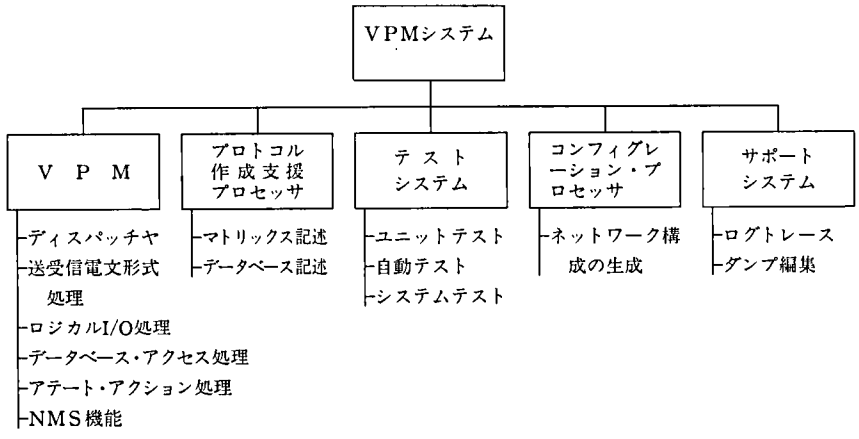


図1　VPM システムの構成

Fig. 1　Configuration of VPM system

形式で作り出すプログラムである．データは，レイヤプログラムとは独立に VPM に登録され，レイヤプログラムよりアクセスすることが可能である．

5) サポートシステム……サポートシステムは，レイヤプログラムより集められた，ログデータ/トレースデータの編集，統計情報の編集，ダンプデータの編集を行うソフトウェアの集合である．

2.2 レイヤプログラム作成の流れ

レイヤプログラム作成の流れを図 2 に示す．

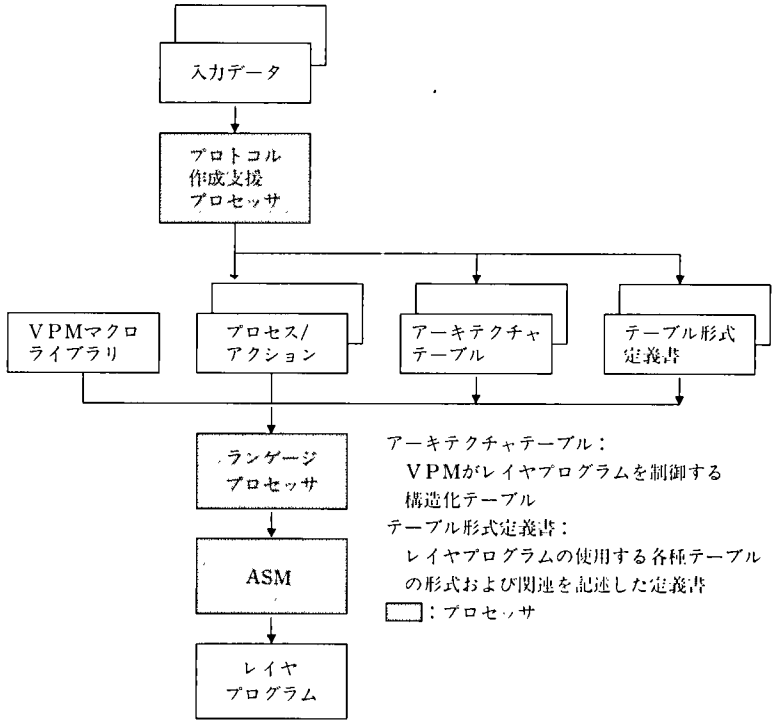


図 2 レイヤプログラム生成の流れ

Fig. 2 Procedure of program development

プログラマより対話形式により入力されたデータに基づき，プロトコル作成支援プロセッサは，VPM が使用する各種アーキテクチャテーブル，プロセスコード，アクションコード等(後述)を作成する．プロトコル作成支援プロセッサの出力は，C 等のランゲージプロセッサの入力となる．また，テーブル形式定義書はコンフィグレーションプロセッサの入力にもなり，ネットワーク生成のベースとしても共通に使用される．

作成されたレイヤプログラムは，VPM に登録することにより動作可能となる．

2.3 レイヤプログラムの記述の標準化

レイヤプログラムの作成は標準化されており(図 3)，プログラマはプロトコル作成支援プロセッサが提供する標準の手順にそって，①～⑫ の順番にコード入力，変更，テーブルの定義等を行っていけばよい．他のプログラムとの関連，プログラムの置かれる環境，等は意識する必要なく，プロトコル仕様書に近い形で入力可能である．

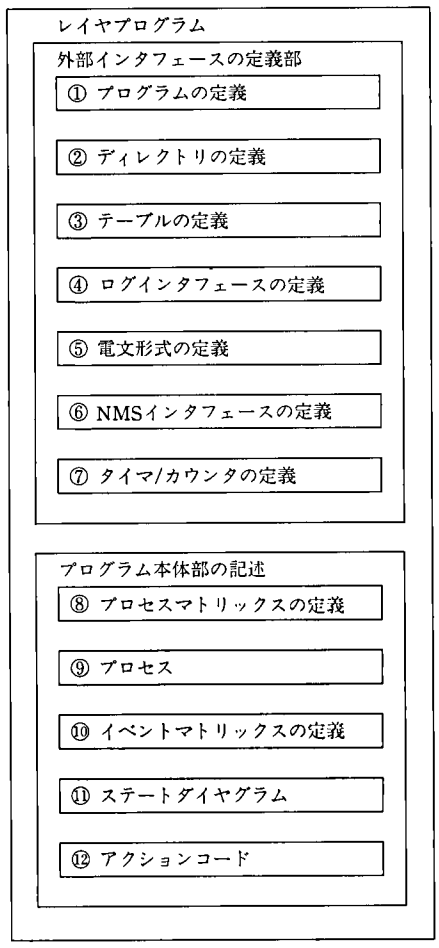


図3　プログラムの記述
Fig. 3　Standardization of program

3. 仕　　様

3.1 VPM

VPMとレイヤプログラムとの関連を図4に示す．図に示すようにレイヤプログラム相互が直接インタフェースすることはなく，必ずVPMを介してデータおよび制御情報の送受信を行う．また，送受信は基本的にパケットの形式で行われる．

VPMとレイヤプログラムとのインタフェースは標準化されており，統一した形式をしている．プログラムの作者はインタフェースの仕様を知る必要はなく，プロトコル仕様書に記述されているプリミティブ，ステートダイヤグラム，電文形式，等の仕様の知識があれば相互に通信できるように配慮されている．

VPMは以下の機能をレイヤプログラムに提供する．以降，これらの機能について説明する．

- ディスパッチャ
- 送受信電文処理
- ロジカルI/O処理

・データベースアクセス処理
・ステートアクション処理
・NMS インタフェース機能
・タイマ処理
・ロギング処理

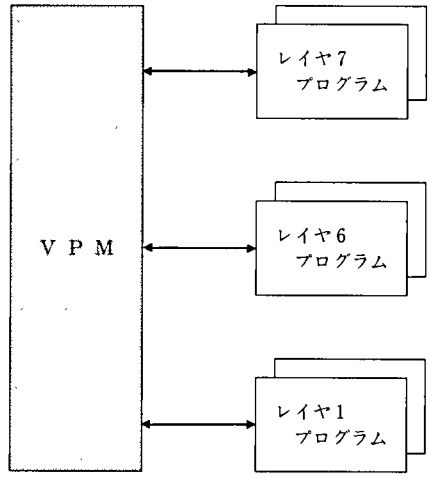


図4 VPM とレイヤプログラムとの関連
Fig. 4 Relationship between layer program and VPM

3.1.1 ディスパッチャ

ディスパッチャは，レイヤプログラム等のアクティビティ制御を行う．アクティビティには二つのタイプがある．一つはパケットの送受信に係わるもので，パケットをベースにレイヤプログラムをスケジュールする．該タイプのスケジューリングのプライオリティには四つのクラスがあり，優先データ等の送信に使用する．ただし，クラス1はレイヤプログラムでは使用できない．

二つ目は，タイムアウト，回線ダウン，等のイベント* に係わるもので，これはパケットを持たず，エンティティベースのスケジューリングである．これらのイベントは障害時一斉に発生する可能性があり，パケットレベルのインタフェースを使用すると，バッファを使い果たす恐れがあるので，それを防止するためのものである．

これらのいずれを使用するかはレイヤプログラムに任される．レイヤプログラムは回線ダウン等の処理時，上位または下位プログラムの I/O アクセス時に指定すればよい．指定しない場合，パケットインタフェースが仮定される．また，いずれを使用した場合でも，レイヤプログラム間のインタフェースの整合性は維持される．

3.1.2 送受信電文処理

送受信電文処理は，VPM 内の MAPPER プログラムにより，プロトコル作成支援プロセッサにより作成された電文形式定義に基づき，翻訳および生成処理される(図5)．

これより，レイヤプログラムは，電文形式定義時の電文のフィールド名により直接

* イベント：プロトコル仕様書の中で一般に使用される用語で，内部的，外部的に発生する事象の総称である．プロトコルは該事象をベースに記述される．

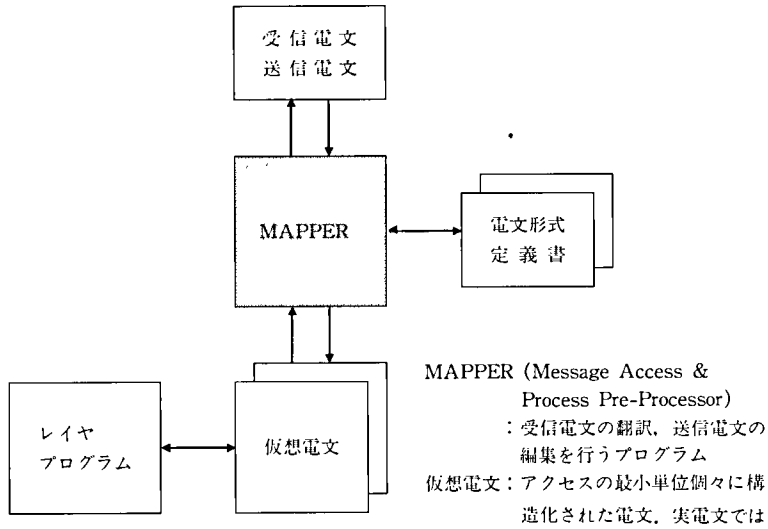


図5　電文形式翻訳プログラム

Fig. 5　Message format interpreter

その値の参照が可能である．したがって，煩雑な受信または送信電文の解析，作成が不要で容易に電文の処理が可能で，プロトコル仕様書の電文形式と同等な形での処理が可能となる．

仮想電文領域は，レイヤプログラム各々に4個設定可能であり，電文のコピー/セグメンテーション等が，これら仮想電文相互間にて可能である．また，このための処理ルーチンもMAPPERが提供する．

3.1.3　ロジカルI/O処理

本処理は，レイヤ相互間，またはレイヤ/NMS相互間，等にてデータおよび制御情報の送受信を行うものである．これにより，レイヤプログラムは任意の相手と自由にデータおよび制御情報を送受信することができる．

3.1.4　データベースアクセス処理

本処理は，レイヤプログラムが参照するテーブルのアクセス処理を行う．レイヤプログラムは，プロトコルを実行する上で，ディレクトリ，セッションテーブル・端末テーブル等，各種のテーブルをアクセスする必要がある．これらのテーブルの形式はプロトコルの属性により異なる．したがって，自由に形式を定義できなくてはならない．

レイヤプログラムは，テーブルの種類およびテーブル名，アドレスによりテーブルの参照が可能である，また，テーブル内のフィールドの読み書きは，プログラム作成時に定義されたテーブル形式定義書のフィールド定義名により，直接行うことができる(図6)．

これにより，プログラマはテーブルの実際の存在場所，物理的形式，ビットの位置等を知る必要はなく，容易にプログラムを作成できる．また，新たにフィールドを追加した場合も，レイヤプログラムは何の影響も受けない．

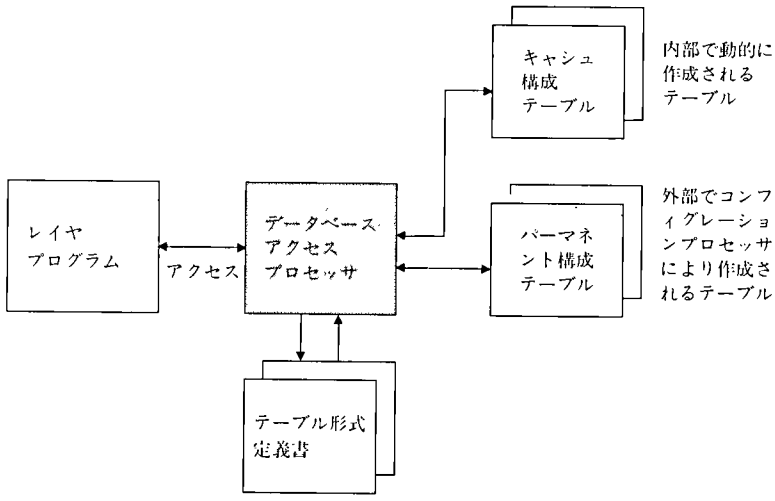


図6　テーブルのアクセス

Fig. 6　Configuration table access process

また，該定義テーブルはコンフィグレーションプロセッサ(後述)により共通に使用される．したがってレイヤプログラムは，コンフィグレーションプロセッサへの構成変更等の指示は必要ない．

3.1.5　ステートアクション処理

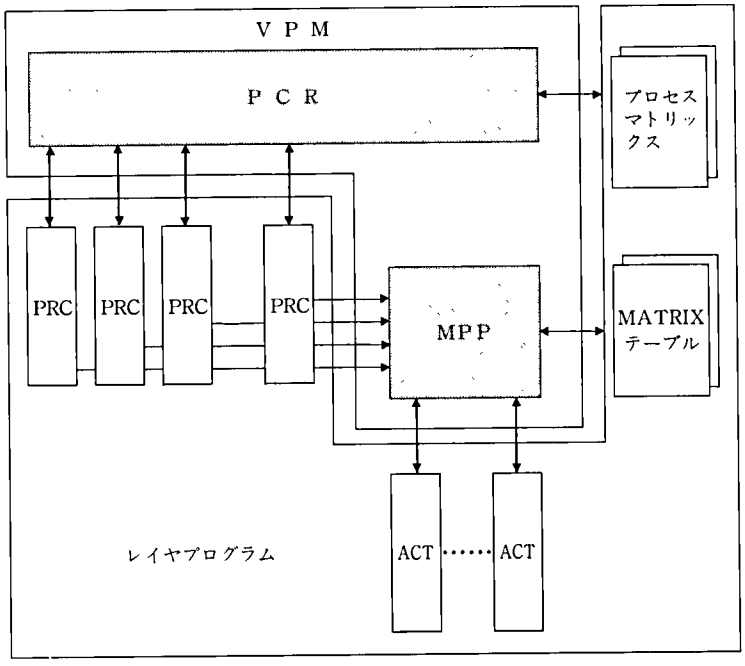
ステートアクション処理の流れ図を図7に示す．VPM はステートアクション処理を，PCR(プロセス制御ルーチン)，MPP(MATRIX 処理ルーチン)，の二つの階層に分けて処理する．

図の PCR および MPP が使用するプロセスマトリックス，MATRIX テーブルはプリプロセッサが作成する．また，レイヤプログラムも二つの階層に対応し，プロセス処理(PRC)プログラムおよびアクション(ACT)プログラムに分けられる．

PCR は，レイヤプログラムの定義時に指定されるプロセスマトリックスに基づき，受信したパケットのプリミティブのタイプ，タイムアウト，NMS よりの入力，内部的に生成されるプロセス，等の種類に応じ指定されたプロセスに制御を渡す．PRC は指定されたプリミティブの処理，等を実行し MATRIX 入力のイベントを発生する．また，PRC はその処理の中で他のプロセスを発生させることができる．したがって，データ受信時の処理でアウトプットのウインドゥがオープンし，データを送信可能となったとき，またはビジー状態が解除されたときには，アウトプット要求のプロセスを発生させることにより送信処理も同時に行うことが可能である．したがって，個々の PRC は自身の処理に専念すれば良く，他のことを考慮する必要はない．

MPP は，イベントの種類および MATRIX に基づき ACT に制御を渡す．ACT は個々のイベントに対応した処理の実行を行う．また，ACT はそのアクション処理の中で，PRC と同様に，さらにイベントおよびプロセスを発生させることができる．したがって，複数のアクションおよびプロセスを同時に処理できる．

MATRIX には階層構造を持たせることも可能で，下位のステートダイヤグラムは，



PCR	プロセス制御ルーチン
MPP	MATRIX 処理ルーチン
プロセスマトリックス	プロセス処理マトリックステーブル
PRC	プロセス処理プログラム
MATRIX	ステートアクションテーブル
ACT	アクションプログラム

図7　ステートアクション処理流れ図

Fig. 7　Flow of state action processing

上位のステートが指定されたステートにないとき，イベントの処理は行わず，リセット状態に保持される．

以上の機能により，ACT および PRC は非常に簡単な構造にすることができる．実際のコーディング例を付録 A に示す．

3.1.6　NMS インタフェース機能

NMS インタフェース機能は，オペレータまたは上位ネットワーク制御プログラム(アプリケーションプログラムまたはネットワーク管理プログラム)からの各種制御コマンドの一部の処理，またコマンドのレイヤプログラムとの受け渡し，受信コマンドの翻訳，送信コマンドの編集を行う．コマンドはパラメタ形式をしており，オペレータからの文字形式のコマンドは，本プログラム内の翻訳プログラムによりパラメタ形式に変換される．レイヤプログラムはパラメタ化された形式にて処理すればよい．また，プロセスマトリックスの定義でコマンドごとにプロセスを指定できるので，プロセスは該当コマンドの処理を実施すればよい．

3.1.7　タイマ処理

レイヤプログラムは，プロトコル制御のため各種のタイマが必要である．タイマ処理では，エンティティ(セション，端末等)ごとに複数のタイマを準備している．した

がって，レイヤプログラムはタイマのセット/クリア時エンティティとの対応付けは不要で，容易にタイマの制御が可能である．

3.1.8 ロギング処理

本処理は，エラー，統計情報の収集等の処理を行うルーチンで，ログインタフェース定義テーブルの規定に従って，メッセージの発生，カウンタのカウントアップ，ログエリアの設定を行う(後述)．

3.2 プロトコル作成支援プロセッサ

本プロセッサは，レイヤプログラムの製作者と会話または一括入力にてハンドラの作成を支援するプログラムで，VPM が使用する以下のアーキテクチャテーブルおよびコードの生成を行う．本章ではこれらの役割について解説する．

- プロセスコード
- アクションコード
- レイヤプログラム制御テーブル
- プロセスマトリックステーブル
- ステートアクションテーブル
- イベントマトリックステーブル
- 電文形式制御テーブル
- テーブルアクセス制御テーブル
- ログインタフェーステーブル

3.2.1 プロセスコード

該コードは，ハンドラの本体でプリミティブの種類ごと，内部的に発生するアクティビティ個々に作成する．VPM はプロセスごとにアクティビティの割り振りを行う．この割り振りはプロセスマトリックスをベースに行われる．

3.2.2 アクションコード

該コードは，イベントダイヤグラム定義時のアクション処理コードで，ステートアクション処理プログラムにより使用される．本コードは該当するイベントの処理を実施する．

3.2.3 レイヤプログラム制御テーブル

本テーブルは，図 8 に示す構造をしており，VPM がレイヤプログラムを制御するために必要な以下の情報が含まれる．

- プログラム名，プログラム属性情報
- プロセスマトリックステーブル，ステートアクションテーブル等のインデックス
- レイヤ番号，サブレイヤ番号

3.2.4 プロセスマトリックステーブル

本テーブルは，受信電文の処理，タイムアウトの処理，送信電文の処理，NMS コマンド等レイヤプログラムのプロセスルーチンのインデックスが格納され，VPM がこれらのプロセスを呼び出す時に使用する(図 9)．プロセスの指定のないものについてはデフォルトのプロセスが仮定され，該当の処理が提供されていないことがログされる．また，プロトコルに共通的に使用されるものについては，標準的なプロセスが提

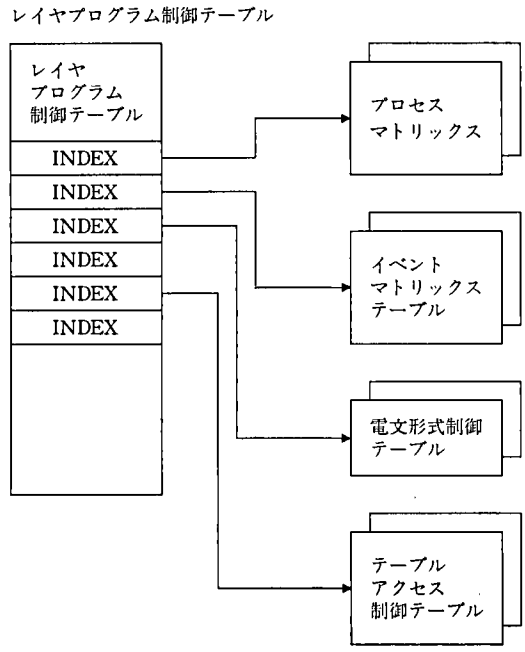


図8　レイヤプログラム制御テーブルの構成

Fig. 8　Configuration of layer program control table

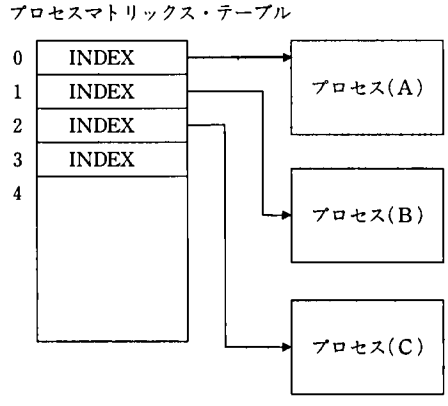


図9　プロセスマトリックステーブルの構成

Fig. 9　Configuration of process matrix control table

供されるので，レイヤプログラムの製作者はそれを修正使用することができる．

3.2.5　ステートアクションテーブル

本テーブルは，イベントごとに当該イベントを参照するマトリックス番号とその中で参照しているステート番号(複数可)およびその場合のアクションプログラムのインデックスが入っている(図10)．実際の記述例については，付録A参照のこと．

VPMは該テーブルに基づきイベントの処理のためのアクションプログラムを呼び出す．

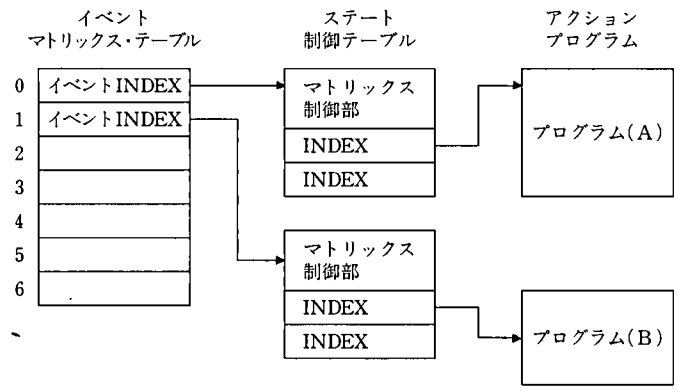


図10　ステートアクション・テーブルの構成

Fig. 10　Configuration of state action control table

3.2.6　電文形式定義テーブル

本テーブルは，MAPPERにより，電文の翻訳•作成を行うのに使用される．一般的に電文の形式は図11の構成をしており，電文形式の記述もこれに対応して以下の三つのレベルで記述される．実際のコーディング例を付録Bに示す．

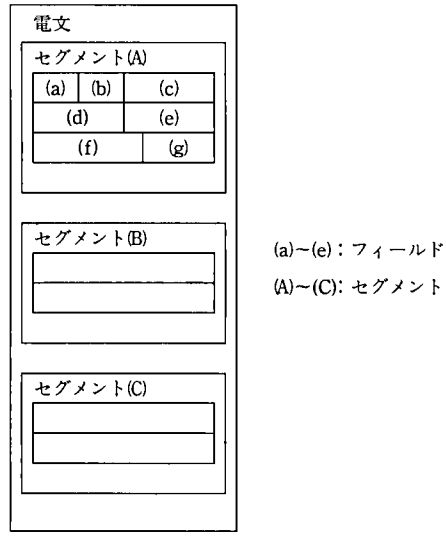


図11　電文構成の定義

Fig. 11　Definition of message format

フィールドの定義では，BIT/BITの集合またはバイト/バイトの集合等アクセスする最小単位のエリアの定義を行う．レイヤプログラムは該フィールドの名前にて読み書きが可能である．セグメントはフィールドの集合を示し，メッセージは送受信する電文の単位である．たとえば，X.25のGFI/LCNフィールドは，全メッセージ共通に使用する．これをセグメントとしてまとめて定義しておけば，全体のメッセージが共通に参照できる．

プロトコル作成支援プロセッサは，ユーザの定義に基づき，図12に示すテーブルを生成する．VPMのMAPPERルーチンは，該テーブルをベースに受信電文・送信電文の翻訳，編集処理を実施する．ただし，個々のフィールドの分析処理およびフィールドの値の準備は受信および送信プロセスの責任である．

図のセグメント(a)～(c)はメッセージ(A)等より標準に使用できる．フィールドは各セグメントの中で一意に規定する．

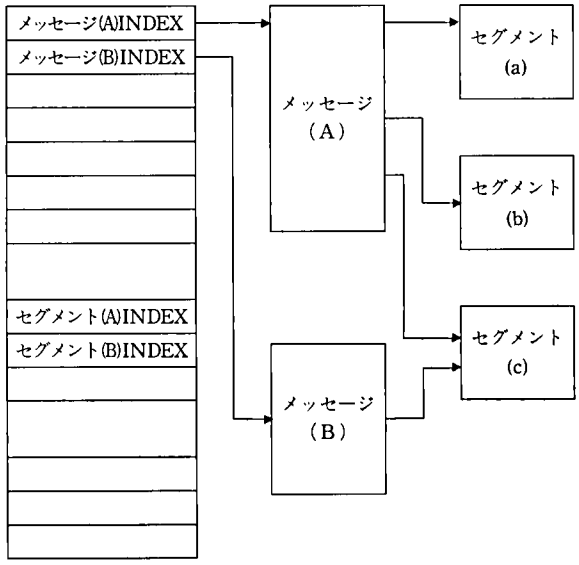


図12　電文形式制御テーブル

Fig. 12　Configuration of message format control table

3.2.7　テーブルアクセス制御テーブル

本テーブルは，レイヤプログラムが使用する各種のテーブル(ディレクトリ，端末テーブル等)の読み出し・書き込み制御に使用する．これにより，レイヤプログラムは物理的テーブルの形式には関係なく，本定義の名前にてアクセスが可能となる(図13)．

ただし，ここではテーブルの形式とテーブル相互間の関連のみを定義する．実際のテーブルはコンフィグレーションプロセッサ(後述)，または動的に，たとえばロジカルチャネルが確立した時に，レイヤプログラムよりの指示によりVPMが作成する．ただしフィールドの値はレイヤプログラムが指定しなくてはならない．指定のないものについては，電文形式の定義時に指示されたデフォルト値が仮定される．前者をパーマネントテーブル，後者をキャッシュテーブルと呼ぶ．

3.2.8　ログインタフェーステーブル

本テーブルはレイヤプログラムがエラー，統計情報等をオペレータまたはネットワーク管理システムに知らせるためのインタフェースの規約を規定するものである．ログには二つのタイプがある．一つはメッセージタイプで，イベントが発生したとき，メッセージをコンソールまたはネットワーク管理システムに通知するもので，重大な

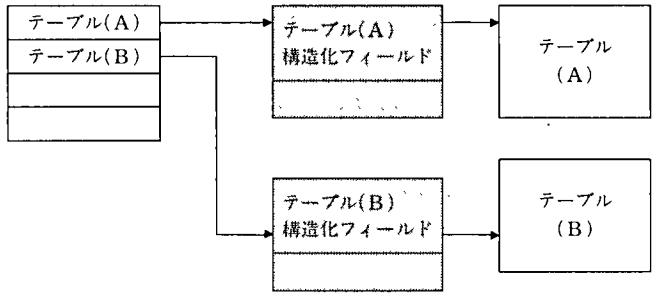


図13　テーブルアクセス制御テーブル

Fig. 13　Configuration of table access control table

エラーが発生した場合等に使用される．

二つ目はカウントタイプのログで，インターミッテント(修復可能なエラー)なエラーの場合に使用されるもので，イベントが発生時カウントのみが取られる．カウンタがスレショールドになった時にエラーメッセージを発生する．本タイプでは，ログエリアの確保の方法としてパーマネントアサインとダイナミックアサインの2種を提供する．前者は固定的にエンティティにエリアを割り付けするもので，後者はNMSコマンドの指定により動的にエリアが取られるものである．これは定義時に指定できる．

3.3　テストシステム

テストシステムは，レイヤプログラムのユニットテストおよびシステムテストの2種類のソフトウェアから成る．

3.3.1　ユニットテスト

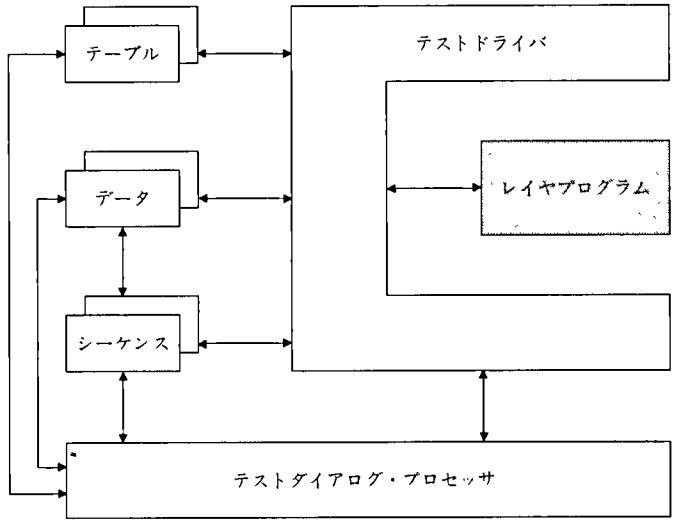
該ソフトウェアは，レイヤプログラムを対話形式でテストするソフトウェアで，開発の効率化のため，できるだけ自動的にテストが可能なように配慮されている．

テストスクリプトのテスト手順はテストシーケンス制御テーブルに記述される．本システムは，この指定された手順に従って，データ，イベント等を発生する．レイヤプログラムの動作はシーケンスチャートの形でログされ，後刻の解析に使用される．また，複数のテストケースを連続して行えるように，データおよびテーブルの情報のセットアップはテストケースごとに指定でき，各種のエラー状態を任意に発生させることができる(図14)．これによりテストを自動的に行うことが可能である．

図14のデータおよびテーブルの形式は，テーブルおよび電文形式定義書にあらかじめ規定されたものを使用する．また，シーケンス，データおよびテーブルのフィールドの値は，テストダイアログプロセッサを介して会話形式で入力する．これら入力データはファイルとして保存され，修正・追加・再使用が可能である．

3.3.2　システムテスト

本テストシステムは，テストされるレイヤを含めて通常の動作環境にて作動する．テストのために上位テストレイヤ(上位テストドライバ)が提供される．該テストレイヤは，ユニットテストと同様，指定されたシーケンスにてデータを発生する．また，入力データの内容，タイプ，イベントに応じたデータの発生，タイマのセット等の制御も可能である(図15)．



テストドライバ：該ソフトウェアはVPMと同等の働きをするのと同時に上位および下位から指定されたデータを，シーケンス制御の指示に従って順番にレイヤプログラムに渡す．同時にシーケンスごとにテーブルに指定された値をセットする．同時にレイヤプログラムの実際のシーケンスをログする．

テストダイアログ・プロセッサ：該ソフトウェアは，オペレータとインタフェースし，シーケンス制御テーブルのセット，送信または受信するデータのタイプ内容のセット，テストのスタート指示等の制御を行う．セットされたデータ，テーブル，シーケンス制御はデータベースとして保存される．

図14　ユニットテスト・システム構成図

Fig. 14　Configuration of unit test system

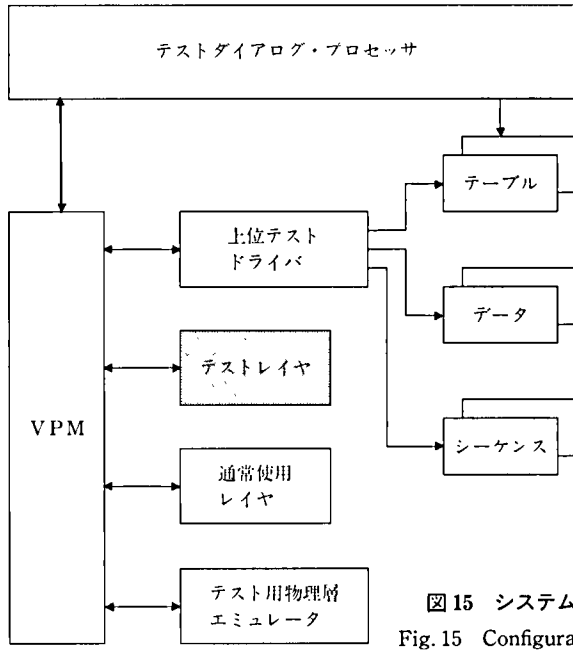


図15　システムテスト・システム構成図

Fig. 15　Configuration of system test system

テスト用のレイヤ1はデータの折り返し等を行う．これは通常のレイヤを使用することも可能である．

3.4 コンフィグレーション・プロセッサ

コンフィグレーション・プロセッサは，レイヤプログラムが作成した，構成テーブル形式に基づき，実際のネットワーク構成データを生成するプロセッサで，会話形式または一括形式にて作動可能である(図16)．

また，オンライン中に一部の構成を変更することも可能である．ただし，変更/追加された構成を実行させるためには，当該構成部分のみの初期化が必要である．このためのオペレータコマンドが提供される．レイヤプログラムもこの初期化プロセスを支援しなくてはならない．

ここで作成したコンフィグレーション・データはVPMに登録され，レイヤプログラムから使用できる．

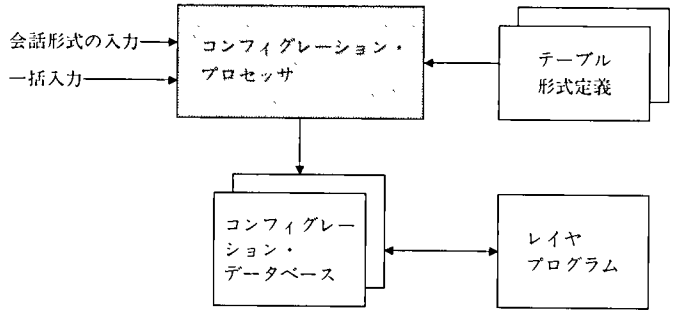


図16 コンフィグレーション・プロセッサの構成
Fig. 16 Configuration of configuration processor

3.5 サポートシステム

本システムは，通信システムを維持・保守する上で必要な各種プロセッサの集合であり，以下のプロセッサからなる．

- 会話型ダンプ編集プロセッサ
- ログ編集プロセッサ
- 統計データ編集プロセッサ

4. おわりに

本稿は，長年通信に係わってきた経験に基づき作成したものである．

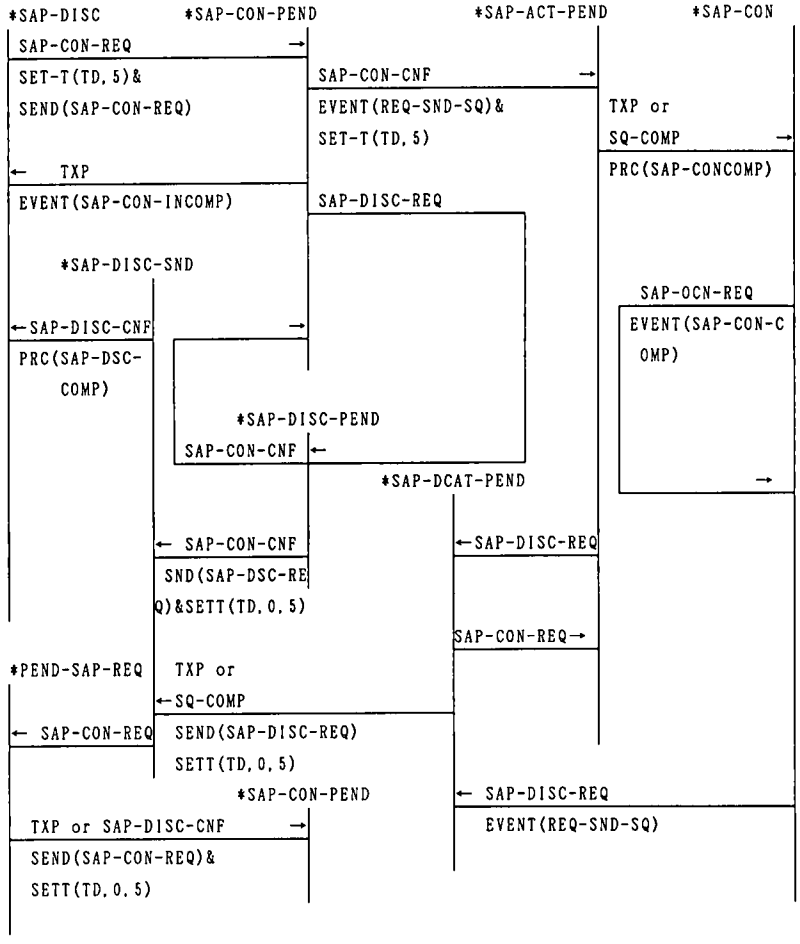
過去，当社は通信システムの開発(ハンドラ)に膨大なマンパワを消費してきた．これは，少なくともOSIプロトコルの仕様が最終的に固まり，一般的に世の中に受け入れられ普及する，20世紀末までは継続するであろう．

このような状況に鑑み，プロトコルの開発の効率化ひいては自動化の手法についてこれまで『プロトコル開発効率化設計仕様書』としてまとめてきた．本稿はこれを要約したものである．

[付録 A]

ステートアクション・ダイヤグラム記述例

以下に上位レイヤとの接続の確率/解除および X.25 のロジカルチャネルの確立/解除についてのステートダイヤグラムおよび,それを本マトリックス記述言語にて作成した場合の記述例を示す.




```
MATRIX  TBL=D, INT=SAP-DISC;
 STATE=SAP-DISC;
    SAP-CON-REQ     NEXT=1, SETT(TD, 0, 5), SEND(SAP-CON-REQ)
 STATE=SAP-CON-PEND;
    SAP-CON-CNF     NEXT=2, SETT(TD, 0, 5), EVENT(REQ-SND-SQ)
    SAP-DISC-REQ    NEXT=4
    TXP             NEXT=0, EVENT(SAP-CON-INCOMP)
 STATE=SAP-ACT-PEND;
    SQ-COMP         NEXT=3, EVENT(SAP-CON-COMP), SETM(0)
    SAP-DISC-REQ    NEXT=6
    TXP             NEXT=3, EVENT(SAP-CON-COMP), SETM(0)
 STATE=SAP-CON;
    SAP-CON-REQ     NEXT=3, EVENT(SAP-CON=COMP), RSETM(0)
    SAP-DISC-REQ    NEXT=6, EVENT(REQ-SEND-SQ), RSETM(0)
 STATE=SAP-DISC-SND;
```

```
    SAP-CON-REQ     NEXT=1
    SAP-CON-CNF     NEXT=5, SETT(TD, 0, 5), SEND(SAP-DISC-REQ)
    TXP             NEXT=5, SETT(TD, 0, 5), SEND(SAP-DISC-REQ)
STATE=SAP-DISC-PEND;
    SAP-DISC-CNF    NEXT=0, EVENT(SAP-DISC-COMP)
    SAP-CON-REQ     NEXT=7
    TXP             NEXT=0, EVENT(SAP-DISC-COMP)
STATE=SAP-DACT-PEND;
    SQ-COMP         NEXT=5, SETT(TD, 0, 5), SEND(SAP-DISC-REQ)
    SAP-CON-REQ     NEXT=2
    TXP             NEXT=5, SETT(TD, 0, 5), SEND(SAP-DISC-REQ)
STATE=PEND-SAP-REQ;
    SAP-DISC-REQ    NEXT=1, SETT(TD, 0, 5), SEND(SAP-CON-REQ)
    TXP             NEXT=1, SETT(TD, 0, 5), SEND(SAP-CON-REQ)
END
```

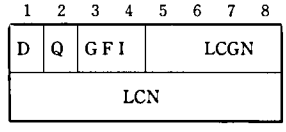
ステートダイヤグラムの記述は，図にそってステート(STATE)とそのステートで意味のあるイベント(SAP-CON-REQ,,,)およびそれに対応した処理(アクション)，次のステートを順番に記述していけばよい．また，これらのステートダイヤグラムは階層構造を持たせることも可能で，下位のステートダイヤグラムは上位のステートが指定されたステートにないとき，ステートアクションの処理は行われず，リセット状態に保持される．

[付録 B]

電文形式の定義例

ここでは，本 VPM を使用し実際の電文形式を定義した例について示す．例は，X 25 のヘッダの定義例である．プロトコルの開発者は，ここで定義したフィールド名(D-BIT, Q-BIT etc)にて直接電文のフィールドをアクセス可能である．

・GFI/LCN　フィールドの定義



上記の電文形式を本記述言語にて記述した例を以下に示す．

```
* SEGMENT NAME=GFI;
* BITS;
    D-BIT=1;
    Q-BIT=1;
    GFI=2;
    LCGN=4;
* BYT;
    LCN=1;
* END
```

注)BITS の意味は，1オクテットの上位の BIT より，FIELD に対応する BIT の数を順番に記述する記述手法の意味である．BYT は1オクテットを単位とした数を指定し，複数のオクテットが集まってフィールドを形成するとき使用する．これ以外に各種のフィールド記述言語を支援する．

執筆者紹介　宮　坂　順　之 (Yorihisa Miyasaka)
昭和43年東北大学理学部物理学科卒業．44年日本ユニシス(株)入社．シリーズ2200/1100の通信システムの開発・設計に従事．現在，ソフトウェア計画部プログラム・マネジメント課に所属．

ソフトウェア生産における工程検査の方法と進め方

A Methodology of Process Inspection in Software Production

西島政信

要　約　開発したソフトウェアがユーザの要求を満足しているかどうかを確認するために，出荷前に製品検査を行い，品質を確認した上でユーザに提供するという方法は，ソフトウェアの品質を保証するための重要な活動である．

しかし，生産の各工程で不良の入り込む可能性が高く，かつ後工程になればなるほど不良の発見と修正に多大な労力と時間を要するソフトウェア生産においては，出荷前の製品検査による品質の確認は，不良の発見が生産の最終段階になるため，不良による手戻りが納期の遅延や開発コストの増大等の問題を生じさせかねない．とくに，ミッション・クリティカルな大規模ソフトウェアの開発においては，製品不良は，納期や開発コストの問題ばかりでなく，製品責任* の問題さえも問われかねない状況にある．

このような問題を回避するには，生産物に入り込んだ不良がそのまま後工程に送り込まれるのを未然に防止し，手戻りによる損失を最小限に抑える活動が必要になる．

本稿では，第三者が工程ごとに生産物の品質を確認し，不良が後工程に送り込まれるのを未然に防止する方策の一つとして工程検査を取り上げる．

まず最初に，検査の位置付けを明確にするために，日本ユニシス(株)システム・プロダクト本部におけるソフトウェアのリリースプロセスを紹介し，その中でソフトウェア検査がどのように位置付けられているかについて述べる．その後，ソフトウェア検査の一つの形態としての工程検査の方法と進め方，その有効性等について事例をもとに概説する．

Abstract　It is an important activity in terms of software quality assurance to make produced software available to customers only after validating its quality through the procedure of product inspection, prior to its release, on whether or not it is in agreement with user requirements.

In software production, however, where the possibility is very great of errors invading in each production process, thus bringing about the problem of "the rearer the process is, the more difficult it becomes to detect errors, "and then error correction takes huge quantities of time and labor, it is not all rare that error-caused reworking may lead to a delay in delivery and an increase in development costs because error detection comes at the final production stage in a quality validation effort for pre-release product inspection. Especially, in large-scale software development which is mission-critical, circumstances around a defective product hardly allows itself to escape not only problems related to a delivery date and costs but blso a product liability issue.

Preventing such problems from arising requires activities by which to stop errors which have invaded the product from being transferred on to the following process, thus minimizing the losses out of reworking.

This paper takes up process inspection as one of the measures which prevent errors from being transferred on to the next process by allowing a third party to verify the quality of the output resulting

* 製品責任(Product Liability)：設計・製造もしくは表示に欠陥がある製品を使用した者，または第三者がその欠陥のために受けた損害に対して，製造業者や販売業者が負うべき賠償責任(JIS Z 8101)

from each development step.

First off, the author writes about the Ninon Unisys software release process and also where software inspection is positioned in the whole process. Then, by picking out a sample case, he gives a brief description of a methodology and its benefits of process inspection which exists as one of the forms of seftware quality verification.

1. はじめに

ソフトウェア製品はハードウェア製品と違って，製品自体が無形なために開発の過程が見えにくく，開発の後工程になればなるほど不良の発見と修正に多大な労力と時間を要するため，不良による設計工程あるいは製造工程への手戻りが，納期の大幅な遅延や開発コストの肥大化等の問題を生じさせている[1]~[3]．

また，近年の大規模ソフトウェアの開発要求は，開発作業を多数の人間で分割しなければならないため，品質管理・原価管理・工程管理等の面で小規模ソフトウェアの開発とは異質な問題[4]を発生させ，高品質なソフトウェアの納期通りの開発をますます困難にしている．とくに，情報化社会の進展に伴うミッション・クリティカルな大規模ソフトウェアの提供では，製品不良は納期遅延や開発コスト増大の問題だけでなく，製品責任(Product Liability)の問題さえも問われかねない状況になってきている．

一方，ソフトウェア生産を取り巻く環境は，高度で複雑な大規模ソフトウェアの開発要求に応じられる技術者の慢性的な不足による未熟練者の増大，生産技術や管理技術の未確立によるプロジェクト管理の困難さ，教育体系や生産環境の未整備による生産効率の悪さ等，依然として厳しい状況にあり，ユーザのニーズやクレームを的確に製品設計に反映し，機能性・信頼性・効率性・使用性・保守性・移植性[5]~[7]の高いソフトウェアをタイムリにユーザに提供できなければ，自社の製品市場さえ維持できなくなりつつある．

このような状況の中で，ユーザに長期にわたって満足して使ってもらえるソフトウェアを継続的に提供していくためには，生産の初期の段階から製品を正しく作り出せるように，生産の“仕組み”と“仕掛け”そのものの質を高めなければならない．

出荷前の製品検査は，品質保証の重要な活動の一つであるが，不良の発見が生産の最終段階になるため，不良の件数や程度によっては納期遅延や開発コストの増大等の問題を発生させる危険性があり，開発の各工程で品質を確認し，不良が後工程に送り込まれるのを未然に防止するための活動も必要になる．

当社のシステム・プロダクト本部(以下当社SP本部)では，そのための一つの方策として提供するすべての基本ソフトウェアをそれぞれの品質水準に応じて，書類審査・製品検査・工程検査の三つの検査形態に分類し，それぞれの形態に従ってソフトウェアの品質を確認し，リリースするという方法をとっている．

本稿では，まず最初に検査の位置付けを明確にするため，当社SP本部におけるソフトウェアのリリースプロセスを紹介し，その中でソフトウェア検査がどのように位置付けられているかについて述べる．その後，ソフトウェア検査の一つの形態としての工程検査の方法と進め方，有効性等について，具体的な事例をもとに概説する．

生産の各工程で不良の入り込む可能性が高い上に，不良箇所の発見と修正に多大な

コストを必要とし，しかも運用段階での不良の発生がユーザ業務に致命的な影響を与えるようなソフトウェアの開発では，生産の各工程で品質を確認しながら開発作業を進めていくことが，現在の状況では，最も経済的にソフトウェアの品質を保証する手段になろう．

なぜなら，工程での不良による手戻りをできる限り少なくすることが，要求品質，開発計画，計画工数との差異を最小化し，かつ納期遅延の発生を極小化させることにつながるからである．

2. ソフトウェア検査とリリースプロセス

2.1 ソフトウェア検査の位置付け

検査とは，被検査品の特性を何らかの方法で計測し，その結果を品質評価基準と比較して合否の判定を行うことである．ソフトウェアの生産活動は，「品質を作り込む活動」と「品質を確認する活動」の二つに大別され，ソフトウェア検査は後者に位置付けられる[8],[9]．

検査とテストの違いは，テスト(以降，試験と同義)は対象物の特性データを得ることが目的であるのに対し，検査の目的は生産の各工程あるいは最終工程で不良品の選別を行い，後工程に対して不良による損失を未然に防止するところにある．

当社SP本部では，品質評価基準に基づくユーザの立場からの公正な合否の判定と不良に対する厳正な是正処置を可能にするため，他社メーカと同じように，検査担当部門を基本ソフトウェア開発・保守担当部門から独立した組織にし，以下のような役割を持たせている．

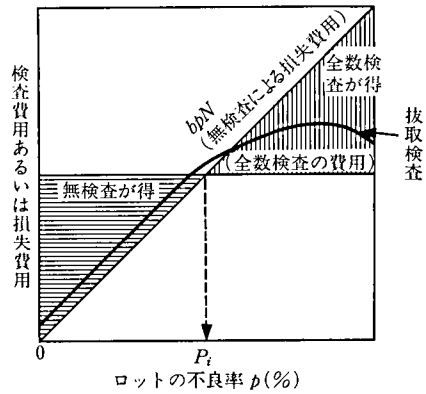
1) 品質評価基準(検査基準)の設定
2) 検査計画の策定
3) 検査ツールの開発および改良
4) 被検査品に対する検査の実施
5) 検査結果の品質評価基準との比較と合否の判定
6) 不良に対する是正指示と処置の確認
7) 検査結果の関連部署への報告
8) 検査技術の調査・研究と技術の向上

2.2 検査コストと検査形態

図1は，検査コストと製品の不良率の関係を示したもので，全数検査に要する検査費用(aN)と無検査による損失費用(bpN)が等しくなる点(損益分岐点)の不良率(Pb)は，$aN=bpN$ より，$Pb=a/b$ で表される[10]．

一般に，検査に要するコストは検査を省くことによって会社が被る損害よりも低く抑える($p<Pb$)のが原則であり，検査に要するコストよりも不良によって会社が受ける損害の方が大きければ($p>Pb$)，不良品がユーザに渡らないように検査活動を強化しなければならない．

しかし，検査活動においても経済的に製品品質を保証する責任があり，検査担当部門を強化する必要があるかどうかは，品質コスト[11],[12]全体のバランスを図りながら進める必要がある．さらに，製品品質がユーザに与える影響の大きさやユーザの品質



	検査費用	不良品による損失費用	計
全数検査	aN	0	aN
無検査	0	bpN	bpN

a：製品1個当たりの検査費用　　p：ロットの不良率
b：不良品1個当たりの損失費用　　N：ロットの中の製品の個数

図1　検査コストと不良率の関係[10]

Fig. 1　Relation of inspection cost vs fraction defective

状況等により，検査の形態や方法を変えることも必要になろう．

製造工業分野で広く使われている「調整型抜取検査」は，被検査品に対して許容される品質水準(AQL：Acceptance Quality Level，許容品質水準)を決め，定常的に品質の良いものに対しては“ゆるい検査”，逆に定常的に品質の悪いものに対しては“きつい検査”，品質に多少のバラツキのあるものに対しては“なみの検査”を適用し，検査を経済的かつ効率的に実施しようというものである[10]．

ソフトウェア生産においても同じように，開発担当部門の設計や製造工程の生産物の品質状況，およびリリース後の品質状況が定常的に良好な場合は，開発担当部門の自主的な品質活動を主体として，検査担当部門による検査を省くことも可能である．逆に，設計や製造工程での工程作業にバラツキがあり，運用段階での品質状況が悪い場合は，検査担当部門の厳しい検査によって品質を確保しなければならなくなる．

検査は，ユーザの要求品質のレベル，製品の重要度，開発・保守段階の作業の安定度，運用段階での品質状況等に応じて，検査形態を変えて運用するのが現実的である．

当社SP本部では，表1の成熟指数あるいは安定指数の許容品質水準をもとに，主管するすべてのソフトウェアを図2の三つの形態のいずれかの検査を受けるプロダクトにあらかじめ分類し，それぞれの形態に従って工程検査・製品検査・書類審査を実施している．

書類審査対象プロダクトは，書類審査のための品質評価報告書(図3)のみによって合否の判定を行うもので，過去の実績から品質状況が安定(工程作業が安定)していて，不良がユーザ業務に多大な影響(たとえば，システムストップ)を与えないようなソフトウェアに対して行う“ゆるい検査”の形態である．

製品検査対象プロダクトは，書類審査と製品検査によって合否を判定するもので，

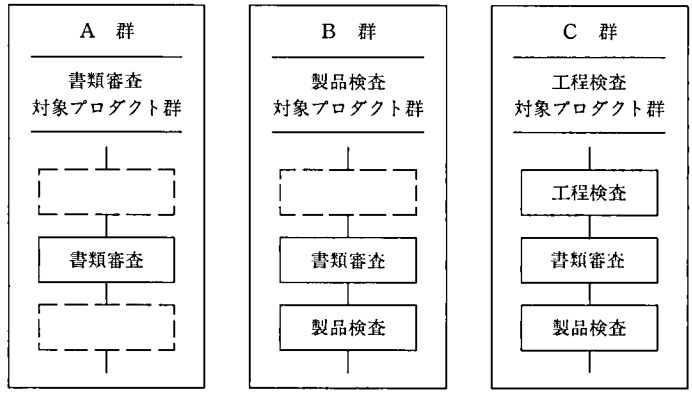
表1　検査種別と許容品質水準の関係
Table 1　Relation of inspection class vs acceptance quality level

検査種別	成熟指数(X)	安定指数(Y)
工程検査	4.0 ≦X	1.5≦Y
製品検査	0.18≦X<4.0	0.3≦Y<1.5
書類審査	X<0.18	Y<0.3

成熟指数(SSUR/Y・S)：年・システム当たりのトラブル(SSUR)の件数
安定指数(EMSS/Y・S)：年・システム当たりのトラブル(EMSS)の件数
SSUR：Software Status User's Report
EMSS：Emergency Maintenance Stop by Software

検査形態	検査対象プロダクト	工程検査	製品検査	書類審査
A　群	書類審査対象プロダクト	−	−	○
B　群	製品検査対象プロダクト	−	○	○
C　群	工程検査対象プロダクト	○	○	○

検査形態と検査対象プロダクトおよび検査の種別



検査形態と検査順序

図2　検査形態と検査対象プロダクトの関係
Fig. 2　Relation of inspection type and selected product group

過去の実績から，品質状況にバラツキ(工程作業が不安定)があり，不良がユーザ業務に多大な影響を与えるようなソフトウェアに対して行う"なみの検査"の形態である．

一方，工程検査対象プロダクトは，開発工程の節目(工程と工程の間)での検査に合格し，さらに書類審査と製品検査にも合格しなければ，該ソフトウェアをリリースできない"きつい検査"の形態である．工程検査は，主として不良の修正に多大な労力と時間を要し，かつ不良がユーザ業務に多大な影響を与えると予想される新規開発プロダクトに対して行われる．

2.3　検査形態の選定方法

当社SP本部が主管するすべてのソフトウェアは，年度初めに以下の手順により，そ

平成　年　月　日　Rev-3

部長印	課長印	担当印

NULソフトウェア品質評価報告書

1. プロダクト名:＿＿＿＿ レベル名:＿＿＿＿ リリース予定日:　年　月　日
検査形態：工程検査 / 製品検査/書類審査

2. リリース計画差異　：年度計画差異:　有　無　四半期計画差異:　有　無
3. 該リリース・レベル　：プロダクト・サイズ＿＿＿＿(LOS/LOC)
：リリース形態　a. イニシャル/メジャー・アップデート(下記 b と c についても記入)
b. フィーチャ・アップデート → 改良コード＿＿件＿＿(LOS/LOC)
C. スタビリティ・アップデート → バグ修正＿＿件＿＿(LOS/LOC)
1件50LOS以上のバグ修正＿＿件
：個別改造コード　＿＿件＿＿(LOS/LOC)

4. リリース可否条件項目 ((2)から(9)について主管部記入、(1)及び(10)について検査担当部記入)

(1) リリース前3 / 2 / 1カ月レビュー実施:　完　未　→　1カ月前レビューでの評価：問題なし / 要注意 / 否
不要 (スタビリティ・アップデート・リリース時)

(2) 新たな非互換 / 非整合:＿＿件　(3) 新たな制限事項:＿＿件　(4) 新たな使用上の留意点:＿＿件

(5) 情報提供(マニュアル)

	マニュアル名	
SRA	:　年　月　日発刊予定　ページ	(　)k:　年　月　日発刊予定 新/改　ページ
SRD導入ガイド	:　年　月　日発刊予定　ページ	(　)k:　年　月　日発刊予定 新/改　ページ
SRDリリース・メモ	:　年　月　日発刊予定　ページ	(　)k:　年　月　日発刊予定 新/改　ページ

(6) ファンクション・テスト　：実施日:　年　月　日～　年　月　日　テスト報告書:　有　無

(7) 不具合及びSPRの対策
：外部仕様レビュー　＿＿件(内 Fix　件)　内部仕様レビュー　＿＿件(内 Fix　件)
：コード・レビュー　＿＿件(内 Fix　件)　ユニット・テスト　＿＿件(内 Fix　件)
：ファンクション・テスト　＿＿件(内 Fix　件)　DPC(MO)リリース(SPR)　＿＿件(内 Fix　件)
：客先環境　＿＿件(内 Fix　件)
：合計＿＿件 → FIX＿＿件、その他＿＿件 (一覧表、又は特記事項に記述の事)
：TAT90日を越えるSPR　＿＿件　→ SPR未処置一覧添付のこと

(8)性能差異　：　有　無　有りの場合の要因＿＿＿＿

(9)SSURの処置
：総未処理件数　＿＿件 うち前々レベル以降で発生する件数　＿＿件
：TAT90日を越え、NUL解析中のSSUR　＿＿件
→ SSUR未処置一覧添付のこと
：TAT90日を越え、UNISYS解析中のSSUR　＿＿件

(10)検査結果:
a 工程検査　各工程での問題　有　無　工程検査　合　否
b 製品検査　整合性検査時の不具合　＿＿件　(基準検査項目数＿＿/実施検査項目数＿＿)
安定性テスト時の不具合　＿＿件　(基準検査時間数＿＿/実施検査時間数＿＿)
DPC MO環境検査(5日間)　＿＿件　投入総M/P　＿＿人時　製品検査　合　否
c 書類審査　合　否

5. 品質情報 (運守項目)

(1)外部仕様レビュー	:参加人数　＿＿人	所要時間	＿＿HR	投入総M/P	＿＿人時
(2)内部仕様レビュー	:参加人数　＿＿人	所要時間	＿＿HR	投入総M/P	＿＿人時
(3)コード・レビュー	:参加人数　＿＿人	所要時間	＿＿HR	投入総M/P	＿＿人時
(4)ユニット・テスト	:項目数　＿＿	ケース数	＿＿	投入総M/P	＿＿人時
(5)ファンクション・テスト	:項目数　＿＿	新規項目数	＿＿	前回項目数	＿＿ 投入総M/P　＿＿人時
	:ケース数　＿＿	新規ケース数	＿＿	前回ケース数	＿＿
(6)客先環境テスト	:サイト名				

テスト形態　a: NULファンクション・テスト、b: NULシステム・テスト、c: 客先システム・テスト

(7) 書類審査対象プロダクトのDPC環境評価期間(MO): リリース日　年　月　日　評価期間＿＿日間
(8) 工程生成物の有無: <RD(本 / 枝), PP(骨 / 本), PSD(本 / 枝), SDS(本 / 枝), CR(枝), UT(計 / 枝), FT(計 / 枝)>
(9) テスト環境: a. 使用機種、EXECレベル
b. 主要関連プロダクト名及びレベル
(10) インストール・テスト：実施日　月　日　未実施理由 (　)

6. 使用予定サイト
(1) 使用予定サイト/予定時期　:＿＿
(2) C/R済みサイト/実施リリース月:＿＿

7. 特記事項、或いは 項目1～6に関する追加事項(主管部記入及び検査担当部記入)

リリース判定コメント

サポート部署印				PM印	検査印

判定
ジェネラル・リリース: 可、不可
a. リリース延期　＿＿ヶ月
b. C / Rサイト　＿＿

添付資料
a. ファンクション・テスト報告書
b 未処置一覧表(SPR未処置,SSUR未処置)
c. その他:＿＿

図3　品質評価報告書

Fig. 3　Product evaluation report

れぞれの検査形態が確定される．

1) 「検査対象プロダクト選定・変更基準」により，製品検査対象プロダクトを選定する．

2) 製品検査対象プロダクトの中から工程検査対象プロダクトを選定する．

3) 1)，2)以外のソフトウェアを書類審査対象プロダクトに選定する．

2.3.1 製品検査対象プロダクトの選定

製品検査対象プロダクトは，以下の条件により選定される．

1) 当社SP本部の主要業務計画項目として挙げられ，客先への円滑な導入と安定稼働の実現を目指しているソフトウェア

2) 多数の客先システム(100システム以上)で使用され，かつ品質がユーザ業務に多大な影響を与えるソフトウェア

3) 許容品質水準が書類審査対象プロダクトの選定水準に達していないソフトウェア

4) すでに製品検査対象プロダクトであり，過去2年間の製品検査で一度でも不合格になったことのあるソフトウェア

5) 1)～4)の中で，検査実施時期までに必要最低限の製品検査に必要な検査環境を準備できるソフトウェア

2.3.2 工程検査対象プロダクトの選定

製品検査対象プロダクトに選定されたソフトウェアの中から，以下の条件を満たすソフトウェアが工程検査対象プロダクトに選定される．

1) 許容品質水準が工程検査対象プロダクトの選定水準にあるソフトウェア

2) 工程検査の実施体制と実施に必要な工数が，開発および検査担当部門の双方で準備できるソフトウェア

2.3.3 書類審査対象プロダクトの選定

製品検査および工程検査対象プロダクトに選定されなかった，すべてのソフトウェアを書類審査対象プロダクトに選定する．

2.4 検査形態の変更

図4は，検査対象プロダクトの検査形態の変更の方向を示したものである．検査対象プロダクトの選定と変更は，年1回，年度初めに「検査対象プロダクト選定・変更

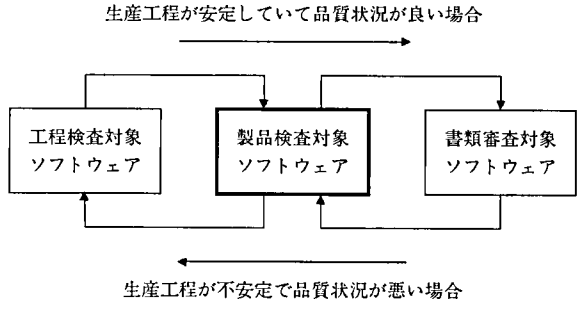


図4 検査形態の変更

Fig. 4 Change of inspection type

基準」に基づいて行われる．

生産工程およびユーザでの稼働品質状況が安定している場合は，左(きつい検査)から右(ゆるい検査)の方向へ，生産工程が不安定でユーザでの品質状況が悪い場合は，右(ゆるい検査)から左(きつい検査)の方向へ検査形態が変更される．

ただし，書類審査対象プロダクトが，製品検査対象プロダクトを飛び越して，工程検査対象プロダクトに選定されることはない．これは，書類審査のみのソフトウェアが，短期間で工程検査の実施環境を整えることが困難なためである．また，工程検査対象プロダクトが製品検査対象プロダクトを飛び越して，書類審査対象プロダクトに選定されることはない．これは，工程検査対象のソフトウェアを書類審査のみのプロダクトに急に移すことにより，工程作業の品質やユーザでの稼働品質状況が以前の状態に戻らないように歯止めをかけるためである．

2.5 書類審査と製品検査の概要

2.5.1 書類審査

書類審査は，品質状況が定常的に安定しているソフトウェアに対し，書類審査用の資料である品質評価報告書により，各工程での所定の作業が十分に行われているかどうかを確認し，合否の判定を行うものである．

品質評価報告書は，検査結果の記載項目(検査担当部門が記入)を除き，ソフトウェア主管部門が記入する．記載項目には必須事項と遵守事項の二つの項目があり，以下の必須項目が検査担当部門により重点的に審査される．

1) 3・2・1か月前のリリース作業の進捗状況
2) ユーザマニュアルの準備状況
3) レビュー(設計書，プログラムコード等)の実施状況とその結果
4) テストの実施状況とその結果
5) 未処置の障害の内容とその対処状況
6) 制限事項や非互換項目の内容とその妥当性
7) 提出すべき資料の添付状況

2.5.2 製品検査

製品検査は，ソフトウェアを実際にコンピュータ・システムで稼働させ，ソフトウェアがユーザに提供されるマニュアル通りに動作し，かつユーザの要求品質を満足する水準にあるかどうかを判定するために行うものである．

製品検査は，できるだけユーザの使用環境に近い検査環境を構築し，以下に示す機能性・信頼性・効率性の三つの基本的な視点(評価項目)から検査対象ソフトウェアの品質が検査される．

1) 機能性……ソフトウェアの機能の正確性，ソフトウェアのレベル間の互換性，関連ソフトウェア間の整合性を確認するための検査
2) 信頼性……所定のソフトウェア環境条件およびハードウェア構成下で過負荷状態を設定し，耐久性を確認するために行う検査
3) 効率性……バッチ，デマンド，リアルタイム処理形態で，ソフトウェアおよびハードウェアの処理能力，処理効率を確認するために行う検査

また，検査担当部門は検査対象ソフトウェアの特性に合わせ，必要に応じて品質評

価基準を設定し，検査結果を品質評価基準と比較して合否の判定を行う．

2.6 ソフトウェア・リリースプロセス

図5は，当社SP本部におけるソフトウェアのリリースプロセスを示したものである．

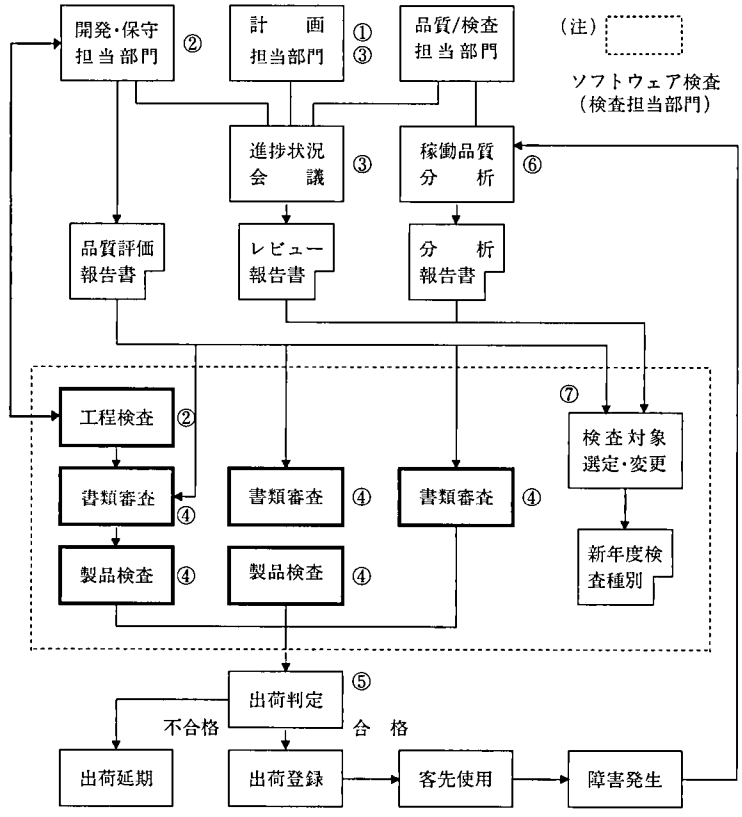


図5　ソフトウェア・リリースプロセス

Fig. 5　Software release process

書類審査と工程検査が従来の製品検査のみのソフトウェア検査に加えられ，三つの検査形態として現行のソフトウェア・リリースプロセスの中に組み入れられたのは1988年4月からであるが，導入までの経緯は以下の通りである．

1)　1984年に，工程検査の必要性から他社メーカの検査技術の調査・研究[13]を行い，工程検査導入のための考え方および実施方法を検討し，開発作業標準の整備，工程検査ツールの作成等，実施に向けての準備作業を完了した．

2)　1986年までに，二つのソフトウェアに対して実験を行い，工程検査の実施手順および実施方法を洗練した．

3)　1987年に，製品検査のみのソフトウェア・リリースプロセスに工程検査を導入すべく検討を行い，最終的に書類審査も含めた三つの検査形態のソフトウェア検査をソフトウェア・リリースプロセスに組み入れるための調整を行った．また，書類審査・製品検査・工程検査のための作業実施要項，それぞれの品質評価基準，検査対象プロダクト選定・変更基準等を整備した．

4)　1988年に，三つの検査形態によるソフトウェア検査をソフトウェア・リリースプロセスの中に組み入れた．また，ソフトウェアの修正保守工程での工程検査についても実験を行った．

以下では，ソフトウェア・リリースプロセスの中で，ソフトウェア検査がどう位置付けられているか，当本部主管のソフトウェアがどのようなプロセスを経てリリースされているかについて，図5に従ってその概略を説明する．

①　当社SP本部からリリースされるすべてのソフトウェアは，年度初めに計画担当部門が作成する「ソフトウェア・リリース計画書」によってリリース予定時期があらかじめ決められる．

②　リリースされるソフトウェアは，ソフトウェア主管部門により，リリース形態(初期出荷版・機能向上版・安定性向上版)に従って開発・改良・保守作業が行われる．ただし，工程検査対象プロダクトの場合は，ソフトウェア主管部門の作業に合わせて，工程ごとに検査担当部門の検査が実施される．

③　リリース予定のソフトウェアは計面通りのリリースを達成するため，計画担当部門により，リリースの3か月前から1か月ごとにリリースに向けての作業の進捗状況がレビューされる．

④　リリース1か月前の進捗状況レビューが終了したソフトウェアは，検査担当部門により，それぞれの検査形態に従って検査が行われる．

・書類審査対象プロダクトは書類審査のみで合否が判定される．

・製品検査対象プロダクトは書類審査と製品検査で合否が判定される．

・工程検査対象プロダクトは開発工程での工程検査に加え，書類審査と製品検査で合否が判定される．

⑤　検査に合格したソフトウェアは，表2で示すリリース判定会議により，最終的なリリースの可否が決定される．

⑥　年度の品質目標が設定されているソフトウェアは，品質管理担当部門により四半期単位にユーザでの稼働品質状況がレビューされる．レビュー結果は，各ソフトウェア主管部門・計画担当部門・検査担当部門に報告され，進捗状況レビューや検査の資料としても活用される．図6は稼働品質管理図の例である．

表2　リース判定会議と可否判定基準

Table 2　Release judgement board and criteria

位置付け	ソフトウェアの出荷可否の最終決定機関
構成メンバ	計画担当部門，サポート担当部門，開発担当部門， 保守担当部門，検査担当部門，品質担当部門
可否判定資料	品質評価報告書および出荷のために必要な所定の資料
可否判定基準	(1)　書類審査基準を満足していること (2)　製品検査基準を満足していること (3)　各種マニュアルが整備されていること (4)　サポート体制が整っていること
判定の種別	(1)　合　　格：出荷登録 (2)　条件付合格：条件を明確にして出荷登録 (3)　不　合　格：出荷延期 (4)　再　検　査：再検査後に再度出荷判定

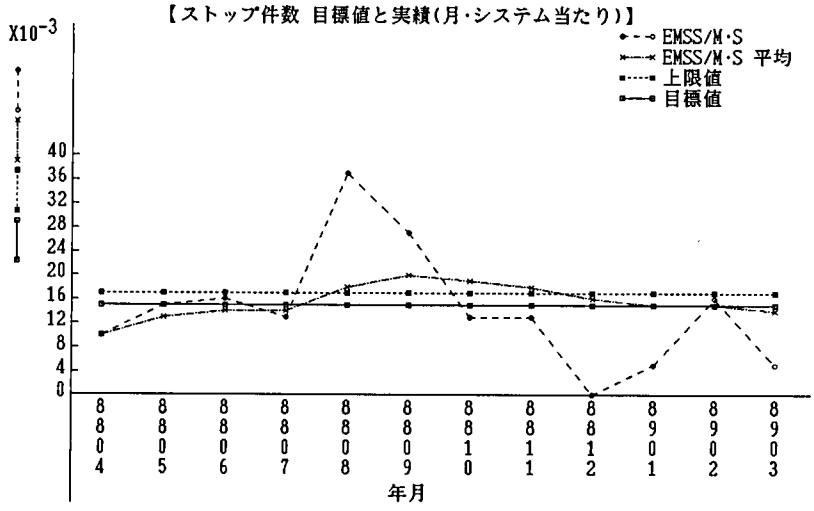


図6 稼働品質管理図(例)

Fig. 6 Chart in use reliability

⑦ 検査対象プロダクトの検査形態の選定およびその変更は，年度初めに検査担当部門が「検査対象プロダクト選定・変更基準」により確定する．

3. ソフトウェアの工程検査の方法と進め方

ソフトウェアの工程検査は，生産の各工程間で行う検査で工程ごとに工程生産物の品質を確認し，工程生産物に入り込んだ不良がそのまま後工程に送り込まれるのを未然に防止し，手戻りによる納期遅延や生産コストの増大を最小限に抑えることが最大の目的である．

ソフトウェア生産においては，工程が進むにつれて不良の発見が困難になり，しかも不良の修正に多大なコストを必要とするため，工程検査は設計や製造工程での不良を早期に発見し，不良の根本原因を各工程作業に確実にフィードバックさせる有効な手段である．

2章では，当社SP本部におけるソフトウェア検査とリリースプロセスを紹介した．本章では，その中の工程検査に焦点を当て，その方法と進め方について概説する．また具体的な事例をもとに，工程検査の有効性，実施上の留意点についても触れる．

3.1 工程検査の考え方

工程検査は,「ソフトウェア生産の各作業が適正に行われているかどうかを確認すると同時に，各工程ごとに作業結果としての工程生産物を検証し，不良が後工程に送り込まれるのを未然に防止するために，検査担当部門が生産の工程間で行う検査」である[8],[10]．

これは，以下の七つの"ソフトウェア生産管理の原則"の中の1)，2)，4)，5)の達成を目指したものである．

1) ソフトウェアの生産工程と各工程での作業の明確化

2) マネジメントの可視性の向上

3)　生産技法や方法論および支援ツールの活用
4)　工程作業の適正な実施の確認と管理
5)　各工程での出力(工程生産物)の検証
6)　生産技法や方法論および支援ツールの定期的な見直しと改善
7)　少数精鋭による開発を可能にするための人材の育成

したがって，工程検査の考え方は以下の4点にまとめられる．

①　第三者による品質の確認
②　各工程での作業のプロセスの確認
③　各工程での工程生産物の確認
④　生産工程の工程間で行われる検査

3.2　工程検査実施手順の概要

図7は，開発工程と工程検査の関係を示したものである．

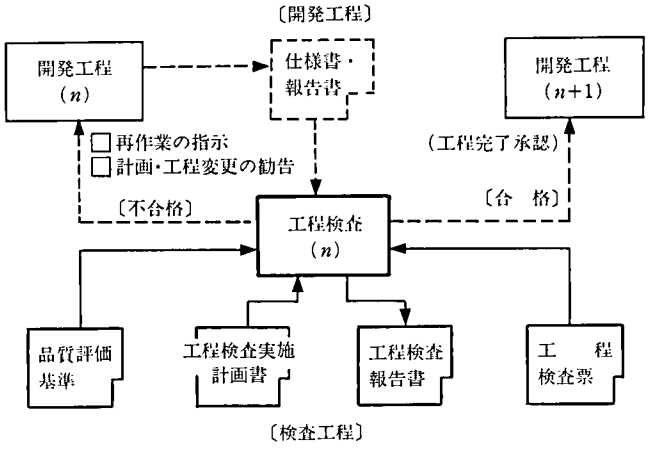


図7　開発工程と工程検査の関係

Fig. 7　Software development step and process inspection

工程検査は，「工程検査実施計画書」に従い，各工程ごとに「工程検査票」を使って工程生産物の確認作業を実施し，「品質評価基準」に基づいて工程完了の判定を行う．検査合格の場合は，「工程検査報告書」で工程完了を承認し，検査の対象になった工程生産物を次工程に引き渡す．検査不合格の場合は，「工程検査報告書」に不合格になった理由と是正処置を明記し，工程生産物を当該工程に差し戻す．

この時点で開発計画の変更が必要であると判定される場合は，「工程検査報告書」でその処置を勧告する．

3.2.1　工程検査計画書の作成

表3は，工程検査の対象となる標準的な生産工程，各工程で工程検査を受ける生産物(工程生産物)および検査で使用する工程検査票の対応を示したものである．

工程検査の対象となるソフトウェアは，開発規模・期間・要員・開発方法・開発環境等の特性により，工程検査の標準的な検査工程・工程生産物・実施手順・品質評価基準を適用できない場合がある．

表3　工程検査の対象工程・工程生産物・工程検査票の関連
Table 3　Relation of development step/output which is inspected and inspection tool (check list)

検査対象工程	工程生産物	工程検査票
要求検討	開発可否検討書 製品開発計画書	製品開発計画書検査票
要求定義	外部仕様書 DR報告書	外部仕様書検査票 デザインレビュー報告書検査票
設計	内部仕様書 DR報告書	内部仕様書検査票 デザインレビュー報告書検査票
製造 (コード作成)	CR報告書	コードレビュー報告書検査票
単体試験 (単体テスト)	単体試験計画書 単体試験報告書	単体試験計画書検査票 単体試験報告書検査票
機能試験 (機能テスト)	機能試験計画書 機能試験報告書	機能試験計画書検査票 機能試験報告書検査票
マニュアル作成	使用解説書 操作解説書	マニュアル検査票

DR：Design Review　　CR：Code Review

検査担当部門は，開発担当部門が作成する開発計画書をベースに当該担当部門と調整を行い，以下の基本的な項目について，その合意事項を工程検査計画書としてまとめる．図8は，工程検査計画書の中の工程検査実施計画/実績表の部分を抜粋したもの

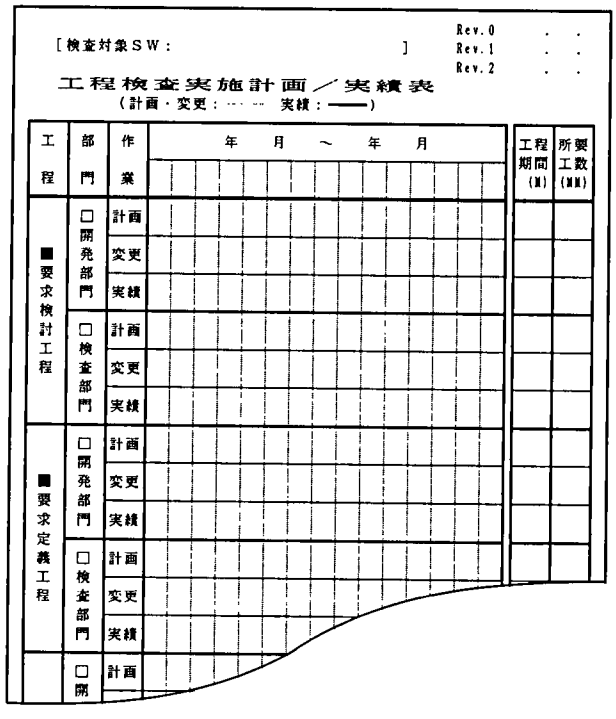


図8　工程検査実施計画/実績表
Fig. 8　Summary sheet of process inspection plan and actual

である．

① 工程検査の対象となる工程と工程生産物
② 工程検査の実施回数(工程期間が長い場合は一つの工程で複数回実施)
③ 工程検査の実施時期
④ 工程検査の実施方法(文書類の査読とレビュー，検査票による検査等)
⑤ 品質判定基準と判定方法および判定結果の処置方法
⑥ 開発作業で遵守すべき作業標準
⑦ 工程検査票の検査内容と例外事項
⑧ 各工程での品質およびコストの実績データの把握方法
⑨ 工程検査担当者の役割

開発担当部門は，検査担当部門と合意した基本的な事項と工程検査に要する付加工数を開発計画書に反映し，必要であれば開発計画書を改訂する．

3.2.2 開発作業標準と工程検査票の整備

工程検査票は，開発担当部門の工程作業が作業標準に従って実施されているか，作業結果としての工程生産物の品質が所定の水準を維持しているかを検査するために使われるツール(チェックリスト)で，当社 SP 本部主管ソフトウェアの開発業務遂行に関する作業標準をもとにして作られている．

開発担当部門は，この作業標準に従って工程作業を実施することになるが，検査担当部門との合意により，作業手順や作業方法に変更がある場合は，その変更に合わせて作業標準と工程検査票の変更を行う．

また，作業標準を新規に作成する必要がある場合は，検査担当部門が作業標準と工程検査票を作成し，開発担当部門の合意を得ることになる．

図 9 は，要求定義工程の工程生産物である外部仕様書の工程検査票を抜粋したものである．

3.2.3 工程検査の実施

作業標準および工程検査票の整備が終われば，工程検査実施計画/実績表に従って工程検査が始められる．

各工程での検査は，再検査を除いて 1 回を原則としているが，一つの工程で工程期間が 1 か月以上にわたる場合は，必要に応じ開発担当部門との合意で複数回の工程検査を実施することがある．この時は，以下に示す二つの方法で工程検査を実施する．

1) 工程中間検査……レビューボードによる検査の方法で，必要に応じて複数回実施される．

開発担当部門は，自部署内で必要に応じてレビューボード(設計書レビュー，プログラム・コードレビュー，テストレビュー等)を開き，開発者の立場で品質の確認を行うが，工程中間検査におけるレビューボードは，計画担当部門，検査担当部門，サポート部門が参加して行う開発担当部門主催のレビュー会議である．

工程中間検査は，開発するソフトウェアの規模が大きくなればなるほど工程期間が長くなり，工程期間が長くなればなるほど要求定義ミス・設計ミス・製造ミス等による手戻りの無駄(手戻りを起こした作業の無駄と，修正のための再作業の無駄)が大きくなる危険性を回避するために行う．開発担当部門と検査担当部門以

検査内容	検査項目	検査対象	作業標準の遵守度	生産物の完成度 満足度	生産物の完成度 充実度
1. 形式仕様（続き）	（17）当該 CPSD のベースになる PSD or SPSD or プロダクト・プランの名称と番号及び当該プロダクトの有効範囲が記述されているか。				
■ はじめに	（18）当該 CPSD の目的が明確に記述されているか。				
2. コンポーネント・サマリ	（1）PSD or SPSD に記述したソフトウェア・コンポーネントの利用者機能の要約が的確に記述されているか。				
3. デザイン・トレードオフとプロダクト目標	（1）PSD or SPSD or プロダクト・プランに記述したプロダクト目標を達成すべき安定性、保守性の目標が明確に記述されているか。				
	（2）PSD or SPSD or プロダクト・プランに記述したプロダクト目標を達成すべき機械からの独立性の目標が明確に記述されているか。				
	（3）PSD or SPSD or プロダクト・プランに記述したプロダクト目標を達成すべきパフォーマンスの目標が明確に記述されているか。				
	（4）PSD or SPSD or プロダクト・プランに記述したプロダクト目標を達成すべき使い易さ、記憶容量の目標が明確に記述されているか				
6. ARM	（1）当該ソフトウェア・コンポーネントの誤り条件及び回復の可能性について正しく定義されているか。				
	（2）PSD or SPSD or プロダクト・プランに記述した ARM を達成する為に行うべき事が詳細に記述されているか。				
9. スタンダードからの逸脱	（1）NUL 及び UNISYS スタンダードから逸脱する全ての事項が明確に記述されているか。				
	（2）逸脱する事項について、逸脱せざるを得ない理由が明確に述べられているか。				

［記入要領］

□ 遵守度
- ○ ・・・ 作業標準を遵守している場合。
- × ・・・ 作業標準を遵守していない場合。
- － ・・・ 検査項目が検査対象外の場合。

□ 完成度
- 5 ・・・ 作業標準の遵守度が"○"で、生産物の完成度が優秀な場合。
- 4 ・・・ 作業標準の遵守度が"○"で、生産物の完成度が良好な場合。
- 3 ・・・ 作業標準の遵守度が"○"で、生産物の完成度が普通な場合。
- 2 ・・・ 作業標準の遵守度が"○"で、生産物の完成度が不良な場合。
- 1 ・・・ 作業標準の遵守度が"○"で、生産物の完成度が不可な場合。
- 0 ・・・ 作業標準の遵守度が"×"の場合。
- － ・・・ 作業項目が検査対象外の場合。

検査結果表	作業標準の遵守度	生産物の完成度 満足度	生産物の完成度 充実度
	／	／	／
採点表	点	点	点

（注）
- ■ 遵守度 ・・・ 作業標準の守り具合。
- ■ 満足度 ・・・ 生産物の記述の範囲。
- ■ 充実度 ・・・ 生産物の記述の内容。

得点 / 満点

図9 外部仕様書工程検査票

Fig. 9 Process inspection check list of product specification description

外の部署が参加するのは，工程検査対象プロダクトの選定条件に加えて，開発期間の観点から多数のユーザを支援することになり，当事者以外の部署によるレビューによって工程生産物の品質を確保することが最大のねらいである．

検査担当部門は，レビューボードでの結果と工程作業の進捗状況をもとに，工程検査票を使って中間的な工程生産物を検査し，当該時点での品質を評価する．

2) 工程最終検査……検査担当部門が工程検査計画書と工程検査票をもとに検査を行う方法で，当該工程完了の合否を判定するために，1回のみ行われる最終的な検査である．工程中間検査が実施されている場合は，レビューボードでのコメントの対処状況も検査される．

表4　品質評価項目と品質評価基準
Table 4　Quality evaluation items and criteria

評価項目		品質評価尺度	判定基準
作業計画の達成度		(計画期間/実績期間) ×100	90%以上
作業標準の遵守度		(遵守項目数/全検査項目数)×100	100%
生産物の完成度	満足度	(Σ評価点/最大評価点) ×100	80%以上
	充実度	(Σ評価点/最大評価点) ×100	80%以上

3.2.4　品質評価基準と品質評価

工程検査による工程完了の合否判定は，表4の評価項目・評価尺度・評価基準に基づいて行われる．

1)　作業計画の達成度……工程検査実施計画書の工程作業の計画期間と実績期間との対比により，作業計画の達成度(作業の進捗状況)を評価する．計画に対する実績の達成率が90%以上であれば合格とする．

各工程作業の達成度が品質評価基準を満足していても，開発計画全体のスケジュールから判断して納期遅れが予測される場合は，開発担当部門と協議し，計画変更あるいは要員変更等の妥当な対処策を勧告する．

2)　作業標準の遵守度……各工程作業が定められた作業標準に従って進められたかどうか，工程検査票を使ってその遵守度を評価する．

工程検査票(図9参照)の“作業標準の遵守度”の欄の“—”(検査対象外の項目で，工程検査計画作成段階で合意した項目)を除くすべての検査対象項目について，工程生産物をチェックし，標準に従った記述があれば“○”，記述がなければ“×”として遵守度を算出する．作業標準の遵守度は，100%(“×”があってはならない)が合格の基準である．

定められた作業手順を遵守していない場合は，検査不合格として再作業を指示する．遵守できない場合は，検査担当部門がその妥当性を再評価する．また，以後の工程作業に関連して作業手順や作業方法の変更が必要であれば，開発担当部門と協議し適切な対処策を勧告する．

3)　工程生産物の完成度……工程生産物の完成度は，記述の範囲がどの程度満足しているかを評価する“満足度”と，記述の内容がどの程度充実しているかを評価する“充実度”の二つの観点から評価する．

工程検査担当者は，それぞれの検査対象項目について，以下の5段階で満足度と充実度の評価を行い，その結果を工程検査票(図9参照)の“生産物の完成度”の欄に記入する．ただし，検査対象外の項目は“—”とする．

5：作業標準の遵守度が“○”で，生産物の完成度が“優良”の場合
4：作業標準の遵守度が“○”で，生産物の完成度が“良好”の場合
3：作業標準の遵守度が“○”で，生産物の完成度が“普通”の場合
2：作業標準の遵守度が“○”で，生産物の完成度が“不良”の場合
1：作業標準の遵守度が“○”で，生産物の完成度が“不可”の場合
0：作業標準の遵守度が“×”の場合

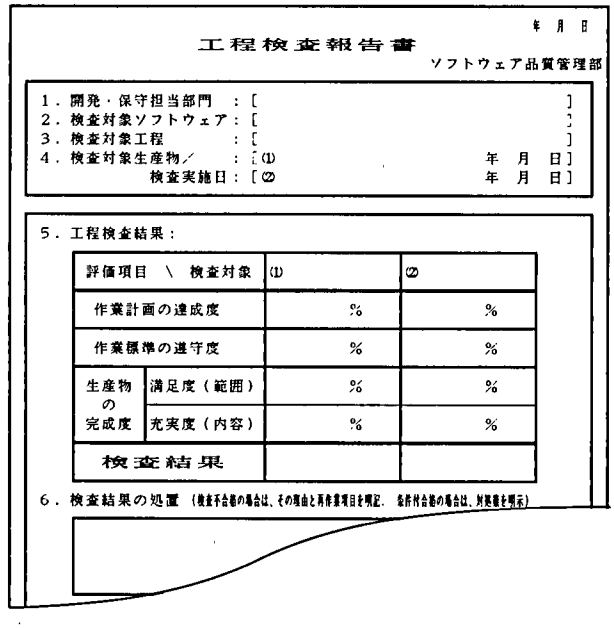
最大評価点は，全検査対象項目数×5点であり，最大評価点に対する各検査項目の総和の比率が80%以上であれば合格とする．ただし，評価点が2点以下の項目は無条件に再検査実施項目になる．

3.2.5 工程検査報告書の作成と報告

品質評価項目がすべての品質評価基準を満足した場合は，検査合格として工程完了を承認する．検査不合格の場合は不合格の理由を明確にし，その程度に応じて再検査あるいは条件付合格(次工程で回復可能と判断された場合，次工程での必須作業項目を明示)の処置をとる．

工程検査担当部門は，工程検査の合否結果とその理由・特記事項等を工程検査報告書としてまとめ，開発担当部門および関連部署にその状況を報告する．

図10は，工程検査報告書を抜粋したものである．



年 月 日

工程検査報告書

ソフトウェア品質管理部

1．開発・保守担当部門 ：[ ]
2．検査対象ソフトウェア：[ ]
3．検査対象工程 ：[ ]
4．検査対象生産物 ：[(1) 年 月 日]
検査実施日：[(2) 年 月 日]

5．工程検査結果：

評価項目 ＼ 検査対象		(1)	(2)
作業計画の達成度		%	%
作業標準の遵守度		%	%
生産物の完成度	満足度（範囲）	%	%
	充実度（内容）	%	%
検査結果			

6．検査結果の処置 (検査不合格の場合は，その理由と再作業項目を明記．条件付合格の場合は，対策案を明示)

図10 工程検査報告書

Fig. 10 Process inspection report

3.3 工程検査の実験結果と実施上の留意点

本節では，過去3回行われた工程検査の実験結果を紹介し，実験結果から得た工程検査の有効性，実施上の留意点および工程検査担当者(工程検査官)に求められる技術的な知識等について触れる．

3.3.1 実験例の特徴と実験結果

開発方法と検査方法の面から，それぞれの実験例の特徴を以下に列挙する．

1) 実験例1(X)は，新規開発の端末ハンドラ(TH)であり，ウォータフォール型の標準的な開発モデルに基づいて開発作業が行われた．検査は，工程ごとに出力される所定の工程生産物に対し，工程最終検査と工程中間検査を併用して実施した．

2) 実験例2(Y)は，新規開発の簡易型DBMSであり，開発はモジュール単位にそ

れぞれの担当者が設計から単体試験までの作業を並行して実施し，機能試験工程で，モジュールの結合と機能試験を行う方法が採られた．標準的な方法による検査が現実的に困難であったため，検査は開発担当部門の設計書レビューやプログラム・コードレビューへの参加と問題点の指摘という，工程中間検査の方法で実施した．

3)　実験例3(Z)は，保守段階のソフトウェアで修正コードの作成から単体試験までの工程について，保守作業と検査作業の手順と方法を新たに設定し，検査は保守担当部門のテスト・レビューボードへの参加と問題点の指摘という，工程中間検査の方法で実施した．

表5，表6は，二つの新規開発ソフトウェアの実験結果を品質面とコスト面からまとめたものである．

表5　品質面からの工程検査実験結果

Table 5　Test result of process inspection (quality data)

実験例	開発段階での不良検出数			出荷後1年間の運用段階での不良発生件数
	総数	DR・CR	UT・FT	
実験例1 (X)	229件 (100%)	161件 (70%)	68件 (30%)	2件 (4ユーザ)
実験例2 (Y)	582件 (100%)	439件 (75%)	143件 (25%)	18件 (22ユーザ)

DR：Design Review　　CR：Code Review
UT：Unit Test　　FT：Function Test

表6　コスト面からの工程検査実験結果

Table 6　Test result of process inspection (cost data)

(単位：人月)

担当部門	実験例1 (X)		実験例2 (Y)	
	計画	実績	計画	実績
開発担当部門	10.5	8.5	52.0	49.4
検査担当部門	4.0	3.7	3.0	2.4
合計	14.5	12.2	55.0	51.8

3.3.2　工程検査の有効性

実験例の特徴と実験結果から工程検査の有効性を評価すると，以下のようになる．

1)　工程検査は，ウォータフォール型の標準的なモデルに基づく開発作業では，工程品質の確認と後工程での不良発生の未然防止を図るための有効な手段である．

［補足］過去のTH開発では，年・ユーザ当たり6件の不良が運用段階で発生していたが，Xでは年・ユーザ当たり0.5件になっている．

2)　モジュール単位に，しかも時系列的に並行して工程作業が行われるソフトウェアの開発に対しては，標準的な工程検査の方法を適用するのはむずかしく，開発担当部門の開発方法に適合した検査手順と方法を設定する必要がある．

［補足］Yの運用段階での年・ユーザ当たりの不良発生件数は0.82件で，Xの0.5件に比べ多くなっている(1.64倍)．

3)　保守工程での工程検査は，保守担当部門の作業手順と方法を明確にし，問題の

ある作業工程に対して重点的に工程検査を実施すれば，その有効性は高い．

［補足］　Zの修正コードに起因して発生したトラブルは，現時点で1修正コード当たり0.05件である．過去の実績では，修正コードが不良のために発生したトラブルは，1修正コード当たり0.2件であった．

4)　工程検査は，工程作業の達成目標を明確にし生産効率の向上を促進するため，開発コストを低減させる効果がある(表6の開発担当部門の実績より)．

5)　4)の成果として，工程検査は検査コストを含めた全体のコストを引き下げる効果がある(表6のX，Yの実績合計より)．

3.3.3　工程検査実施上の留意点

工程検査の実験結果から，検査担当部門による工程検査の実施を円滑に進める上でのポイントあるいは留意点をまとめると，以下のようになる．

1)　工程検査の目的の明確化

2)　開発担当部門および検査担当部門の実施体制(協力体制)の明確化

3)　開発担当部門のソフトウェアの開発手順と方法の明確化

4)　検査担当部門の工程検査方法の明確化

5)　開発作業と工程検査作業の調整

6)　工程検査結果を評価するための開発段階および運用段階のデータの収集

それぞれの項目については，前節で概説した通りである．

3.3.4　工程検査官の技術的な知識と心構え

1)　技術的な知識……工程検査は，「工程での品質の確認」と「後工程での不良発生の未然防止」のために行うものであるから，混入した不良を確実に摘出し，工程生産物の品質を第三者の立場で正しく評価できなければならない．

したがって工程検査官には，以下のような技術的な知識が求められる．

・生産管理についての基礎的な知識
・開発技術についての基礎的な知識
・仕様書のレビュー方法についての基礎的な知識
・検査技術についての専門的な知識
・検査対象プロダクトについての基礎的な知識と，競合他社プロダクトについての一般的な知識

また，具体的な実施の過程では，実施計画の作成と調整，開発計画や工数の変更等の勧告と調整，検査不合格時の代替案の作成と調整等，能力面からは以下のようなものが求められる．

・コミュニケーション能力
・ドキュメンテーション能力
・プレゼンテーション能力
・コーディネーション能力
・危機管理能力

これらの技術的な知識や能力は，努力と経験により身に付けれらるものであるが，過去2〜3個のプロジェクトを経験したことがある7〜10年のソフトウェア技術者であれば，自然に身についている能力でもある．

2)　工程検査官の心構え……工程検査は，開発担当部門との協同作業であり，両部門の連帯感に基づくチームワークが，結果として開発するソフトウェアの品質を左右する．したがって，両部門が互いに敵対関係で作業が行われるようでは，工程検査の所期の目的を達成することはできない．

ソフトウェア開発の主役はあくまでも開発担当部門であり，工程検査官は開発担当者の気持ちになって検査作業を進めることが肝要である．しかし，合否の判定は厳正でなければならず，工程検査官の心構えをあえて挙げるならば，“連帯を求めて検査を恐れず”ということになろう．

4. おわりに[14]~[18]

ソフトウェア生産の目的は，ソフトウェア工学の研究成果である生産技術(技法，ツール)を駆使して，高品質なソフトウェアを効率良く，納期通りに作り出すことである．しかし，生産技術を駆使して開発したソフトウェアが高品質かどうかは，生産物の品質を評価してみなければわからない．また生産技術を駆使しても，生産性の向上や納期を達成できるとは限らない．

今後，生産技術の研究・開発が進み，各種の技法やツールが生産現場にどんどん適用されていくとしても，生産物がユーザの要求を正しく満たしているか，開発作業は計画通り進められているか，開発コストは当初の予算通りか等について，生産の各工程でその状況を確認する作業は依然として残るであろう．

ソフトウェアの開発部門におけるレビューやテストによる工程生産物の品質の確認は，当事者が行うべき基本的な品質保証活動であり，検査担当部門による工程生産物の検査活動は，開発担当部門の開発者志向(Product-out)の視点に加え，ユーザ志向(Market-in)の視点からも，ユーザの要求した品質が正しく製品に反映されているかを確認しようというものである．

本稿で概説した工程検査は，各工程での生産物を第三者がユーザの立場で確認するための一つの方法である．また，生産コストの経済性の観点から，提供するソフトウェアをそれぞれの品質水準に応じて検査するやり方は，ソフトウェアの品質を確認するための効率的かつ現実的な方法であろう．

ソフトウェアの品質を確認(評価)する技術は，5年前からISO* を中心に本格化し，今年中にユーザ視点からの六つの品質特性(機能性・信頼性・効率性・使用性・保守性・移植性)がDIS** となり，1991年には，これらの品質特性のJIS*** 化の作業が開始される予定である．

生産物の品質を正しく評価できなければ，生産のプロセスが正しく行われたかどうかを評価することはできない．生産のプロセスを評価できなければ，生産のプロセスを改善することはできない．生産のプロセスを改善できなければ，生産物の品質を向

---

* ISO：国際標準化機構(Internatioal Organization for Standardization)．ソフトウェア品質評価の標準化作業は，ISO/IEC JTC 1/SC 7/WG 3で進められている．

** DIS：国際規格案(Draft International Standard)．DISとして承認された後は，形式的な手順を踏んでISO規格として制定される．

*** JIS：日本工業規格(Japanese Industrial Standard)．日本規格協会のINSTAC/STD/WG 5がISO/IEC JTC 1/SC 7/WG 3と対応する日本の作業組織．

上させることはできない．生産物の品質を向上させることができなければ，結局は開発作業の生産性も向上させることができなくなる．

したがって，ソフトウェアの品質評価に関する技術は，ソフトウェアの開発方法の標準化，品質および生産性向上の前提となる技術であるとも言える．

当面は，前述の“ソフトウェア生産管理の原則”に基づいた地道な生産活動が求められるとしても，近い将来，ソフトウェアの品質評価技術の確立と生産技術の洗練により，高品質なソフトウェアを効率良く，納期通りに生産できる時代がやってくることを，多くのプロジェクト経験者とともに期待したい．

参考文献

［1］ B. W. Boehm, “Software Engineering Economics”, Prentice-Hall, 1981.
［2］ 宮本勲, “ソフトウェア・エンジニアリング：現状と展望”, TBS 出版会, 1982.
［3］ R. S. Pressman, 岸田孝一監訳, “ソフトウェア・エンジニアリング序説”, TBS 出版会, 1983.
［4］ F. P. Brooks Jr. (山内正彌訳), “ソフトウェア開発の神話”, 企画センター, 1977.
［5］ 情報技術標準化研究センター編, “昭和 63 年度ソフトウェア開発・システム文書化標準化調査研究報告書”, 日本規格協会, 1989.
［6］ 情報技術標準化研究センター編, 平成元年度ソフトウェア開発・システム文書化標準化調査研究報告書”, 日本規格協会, 1990.
［7］ 森口繁一監修, “ソフトウェア品質管理ガイドブック”, 日本規格協会, 1990.
［8］ 石井康雄編, “ソフトウェアの検査と品質保証(日科技連ソフトウェア品質管理シリーズ第 4 巻)”, 日科技連出版社, 1986.
［9］ 日本規格協会編, “標準化と品質管理(ソフトウェアの品質保証特集)”, 日本規格協会, Vol.41, No.2, 1988.
［10］ 佐々木脩, “検査の知識(経営・生産実務シリーズ(5))”, 日刊工業新聞社, 1981.
［11］ A. V. Feigenbaum, (日立製作所訳), “総合的品質管理”, 日科技連出版社, 1967.
［12］ 西島政信, “ソフトウェアの品質管理と品質コスト”, ソフトウェア生産における品質管理シンポジウム発表論文集, 日本科学技術連盟, 1983, pp. 15～22.
［13］ 日刊工業新聞社編, “工場管理(ソフトウェア生産における QC 活動特集)”, 日刊工業新聞社, Vol.30, No.2, 1984.
［14］ 菅　忠義編, “ソフトウェア開発の実際：標準とその活用”, 日本規格協会, 1988.
［15］ 日本規格協会編, “JIS ハンドブック(品質管理)”, 日本規格協会, 1989.
［16］ 菅野文友, “ソフトウェア・エンジニアリング”, 日科技連出版社, 1979.
［17］ 菅野文友, “ソフトウェアの品質管理(日科技連ソフトウェア品質管理シリーズ第 1 巻)”, 日科技連出版社, 1986.
［18］ 吉澤　正, 東　基衛, 片山禎昭編, “ソフトウェアの品質管理と生産技術”, 日本規格協会, 1988.

執筆者紹介　西　島　政　信 (Masanobu Nishijima)

昭和 22 年生．47 年防衛大学校電気工学部卒業．48 年日本ユニシス(株)入社．シリーズ 2200/1100 OS の開発・保守，同基本ソフトウェアの検査に従事．現在システム・プロダクト本部ソフトウェア品質管理部品質管理課長．日本品質管理学会会員，情報処理学会会員，中小企業診断協会会員，INSTAC/STD/WG 5 委員会委員，中小企業診断士(鉱工業部門)，技術士(情報処理部門)．

CAD/CAMシステムにおけるUIMの実現

The Implementation of a User Interface Manager (UIM) for the CAD/CAM System

松林　毅

要　約　複数の専用CAD/CAMシステムのユーザインタフェースの開発において，① 開発工数が多く仕様変更も多い，② 実現方法はシステム間で大差がない，というような問題が発生した．

これまではCAD/CAMシステム開発のたびに，同じようなプログラムを多大な開発工数をかけて作成していた．このような問題を解決するために，ユーザインタフェースを容易に実現・変更でき，かつシステム間で共通に利用可能なユーザインタフェース・マネージャを作成した．

本稿では，ユーザインタフェースに関する共通の操作ルールのモデル化と，その表現形式および実現方法について述べる．

Abstract　In the efforts to develop and maintain more than one dedicated CAD/CAM system, the author and his team members have noticed that the following problems are involved in the implementation of a user interface:

1) There are many development processes and a large number of specification changes.
2) No wide difference is found between system in their implementation.

For all that, we have gone through a huge number of development processes for similar programs so far for every construction of a CAD/CAM system.

To give a solution to such a problem, the author and his team have created a user interface manager which, in developing a new system, helps to easily implement and modify a user interface, and further provides availability common to related CAD/CAM systems.

This paper describes its modeled operation rules acceptable to all user interfaces, and its implementation as well as its representation forms.

1. はじめに

筆者は，さまざまな適用分野の専用CAD/CAMシステムの開発・保守に参画した．基本的にシステムは，ベースとなる各種ライブラリ(図形処理，表示処理，入力処理，データベース処理)，システム固有の機能を実現するコマンド処理群，および使用者とコマンド処理群間のユーザインタフェースを採る処理（ユーザインタフェース・マネージャ：以下UIMと略す）から構成される．

この開発・保守時，ユーザインタフェース実現に際して，以下に示す問題点を認識した．

1) 問題点1……システムの機能は（入出力を含めた）コマンドという単位に分解される．このコマンドのユーザインタフェースを実現する処理の作成には，以下の点を考慮しなければならない．

① 多種多様な入力データの種類と入力方法（PICK, LOCATE…）
② 割り込みやキャンセル等の複雑な制御
③ 表示，入力，データベース等の詳細な内部構造

これにより，処理は複雑でかつボリュウムも多くなり，開発工数が多くなる．

文献[1]によれば，このステップ数はシステム全体のステップ数の 15～58 %と報告されている．

2) 問題点 2……システムの全体構成を考慮した際，UIM には以下の特徴がある．

① UIM の処理構造は，システム間で大差がない．
② UIM の開発は，システム開発のクリティカルパスである．

このような特徴があるにもかかわらず，システム開発のたびにゼロから作成していた．

今回，新規専用 CAD/CAM システムを開発する際，上記問題点を解決できるよう次の目標を設け，これを達成できるような UIM を作成した．

① コマンドのユーザインタフェースを容易に実現できる．
② システム間で共通に利用できる．

本稿では，第 2 章でシステム間で共通のコマンドや入力の制御を含めた操作の基本ルールについて述べる．第 3 章では，これらのルールにそったコマンドのユーザインタフェースを記述する CAD 用言語について述べ，第 4 章では UIM の実現方法について述べる．

2. コマンドモデル

開発に先立ち，既存の CAD/CAM システムのユーザインタフェースにおける操作の基本ルール，操作性を向上させるアイデアおよびシステムの実現方法について調査した．これらの調査結果をまとめて，コマンドや入力の制御を含めた操作の基本ルールを設定し，これをコマンドモデルとして定義した．

コマンドモデルは，コマンドのユーザインタフェースを設計・作成する際に，必ず守らなければならない制約である．開発すべきシステムは，これらの制約を設けることにより，多くの機能を UIM に組み込むことが可能となり，ユーザインタフェースの作成負荷を軽減できる．

このコマンドモデルの定義の際，以下の点に留意した．

1) 操作性を損なうような制約を設けない．
2) 特定のシステムやハードウェアに依存しない．

以下，コマンドモデル内のコマンド制御，入力制御およびコマンド階層構造について述べる．

2.1 コマンド制御

コマンド制御とは，コマンドの実行を制御する機能であり表 1 に示すような機能がある．

コマンド制御に関連して，操作性の向上を目指した項目は以下の通りである．

1) コマンド実行中の任意の時点で任意のコマンドが割り込める．
2) コマンド実行中の任意の時点でコマンドを終了できる．

表1　コマンド制御機能一覧
Table 1　Function of command control

制御項目	機　　能
コマンド選択	実行すべきコマンドを選択する.
コマンド終了	実行中のコマンドを終了させる. そのコマンドが割込みコマンドならば, 元のコマンドの実行凍結を解除する.
割込み開始	あるコマンドの実行中にそのコマンドを凍結して別のコマンドを選択可能にする.
割込み取消し	割込み開始を取り消して元のコマンドの実行凍結を解除する.

表2　入力制御機能一覧
Table 2　Function of input control

制御項目	機　　能
繰り返し終了	同一パラメタの繰り返しの終了を指示する.
実行確認	コマンド固有の機能処理を行う前に, 入力パラメタを確認させる.
入力キャンセル	直前の入力を無効にして再入力させる. 連続的にコマンドの先頭まで無効にできる.
実行キャンセル	直前のコマンドの実行結果を無効にする.

このようなルールを設定すると, コマンドの制御に関する処理をすべて UIM に組み込むことが可能となり, ユーザインタフェースの作成者は, コマンド制御を考慮する必要がなくなる.

2.2 入 力 制 御

入力制御はコマンド内の入力操作の実行を制御する機能であり, 表2に示す機能がある.

入力制御に関連して, 操作性の向上を目指した項目は以下の通りである.

1)　コマンド内の任意の入力をキャンセルできる. さらに, コマンドの先頭まで連続的にキャンセルできる.

2)　任意のコマンドの実行結果をキャンセルできる.

このようなルールを設定するとコマンド制御と同様, 入力キャンセルと実行キャンセルに関する処理をすべて UIM に組み込むことが可能となり, ユーザインタフェースの作成者は入力キャンセルと実行キャンセル制御処理を考慮する必要がなくなる.

2.3 コマンドの階層構造

コマンドの機能は, 図1に示すような階層構造で表される.

コマンドは, 基本的にいったん選択されたらコマンド終了指示まで実行を繰り返す. この1回の実行をコマンドサイクルと呼ぶ.

コマンドサイクルは, G (Graphic) 入力部, 機能実行部, G 出力部から構成される. G 入力部は, 機能実行に必要なパラメタを入力する部である. 機能実行部は, そのコマンド固有の機能を実行する部であり, 入出力を含まない. G 出力部は, そのコマンド固有の機能の実行結果を表示する部である.

G 入力部は, 複数の1G 入力を連結・分岐・繰り返しの制御構造と組み合わせて構成する. G 入力部はコマンドごとに構成が異なる.

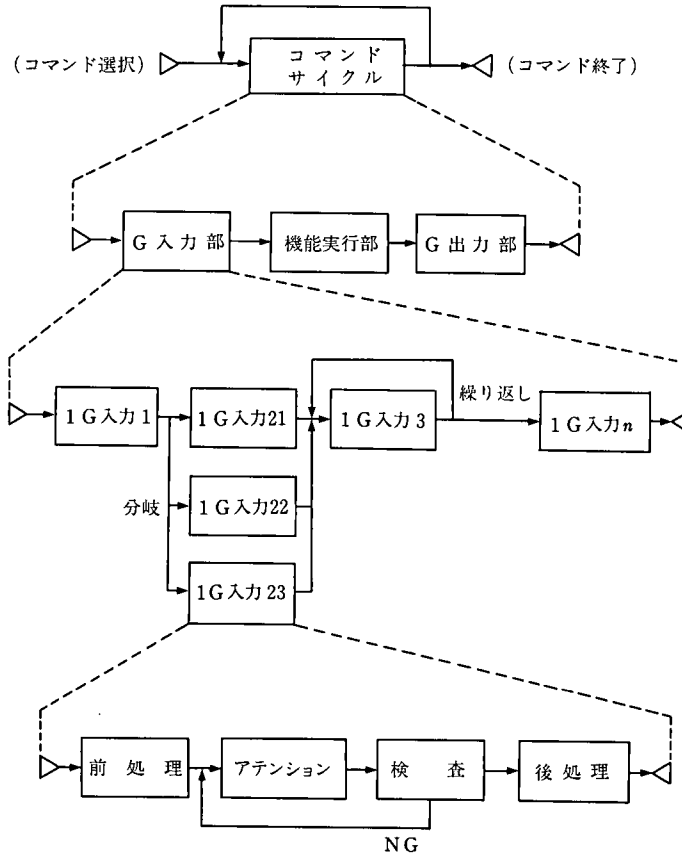


図 1　コマンドの階層構造

Fig. 1　Command structure

なお，文献[2]ではユーザインタフェースの階層モデルとして『Dialog transaction model』と『Dialog cell model』が紹介されているが，筆者の考案したモデルは両モデルを結合したモデルと位置付けられる．

1G 入力は，一つのパラメタの入力に関する部であり，四つのパートから構成される．各パートの機能は表 3 の通りである．

コマンドのユーザインタフェースを設計する場合，1パラメタ入力が設計の基本的な最小単位となる．CAD 用言語は，1回のパラメタ入力を単位とする入力機能を設け，前処理・アテンション・検査・後処理に関連する項目を入力条件として指定できるようにする．

UIM は，入力機能で指定した入力条件をもとに，前処理・アテンション・検査・後処理をこの順に自動的に実行する．検査で NG の場合，自動的にアテンションに戻る．これにより，前処理・アテンション・検査・後処理およびこれらの制御処理を UIM に組み込むことが可能となり，ユーザインタフェースの作成者は，これらの処理を考慮する必要がなくなる．

2.4 入力の順序

コマンド内のパラメタの入力は，基本的に固定の順序が付けられる．この順序が付

表3　1G(Graphic)入力の機能の構成
Table 3　Function of 4-part in single graphical input

パート	機能
前処理	使用者の入力を補助する． ガイダンスメッセージ表示 パラメタ選択メニュ表示 プロンプト表示("半径=") 省略値表示 指示対象データマスク
アテンション	入力すべきパラメタの種類，入力方法等の入力条件のもとでデータを入力する．
検査	入力されたデータの正当性を検査する． 検査NGの場合，アテンションに戻り再入力する． 構文検査：パラメタの種類の正当性検査 意味検査：パラメタの値の正当性検査
後処理	入力データの確認を促す． エコー表示 前処理で表示した入力補助データの消去

けられた入力を『順序付き入力』と呼ぶ．

しかしながら，入力の中には省略値が存在し，必ずしも入力する必要性のない場合がある．このような入力は順序を付けず，任意の入力時点で割り込んで入力できるようにする．このような入力を『任意入力』と呼ぶ．

この『任意入力』は，入力の手数を減らし操作性を向上させる有力な方法である．

3. CAD用言語

前述したコマンドモデルにそったユーザインタフェースの表現手段としてCAD用言語を規定した．ユーザインタフェース作成者は，このCAD用言語を用いてコマンドの操作の流れを記述する．作成されたプログラムを『コマンドシナリオ』と呼ぶ．UIMは，この『コマンドシナリオ』を解釈してシナリオ通りのユーザインタフェースを実現する．

言語形式は，当社のUDL (UNICAD Design Language)[2]をベースに，以下の点に着目して機能を追加した．

1)　定義した『コマンドモデル』にそったユーザインタフェースを少ないステップ数で記述できるようにする．
　①　1回のパラメタ入力の入力条件を一括指定する．
　②　コマンドや入力の制御処理を一切記述しない．
2)　AP（アプリケーション・プログラム）固有の機能を容易に組み込めるようにする．
　①　AP固有の型を定義可能にする．
　②　FORTRANプログラムとのインタフェースを採る．

以下，CAD用言語の特徴的な機能について述べ，簡単な例を示す．

3.1 グラフ入力文

グラフ入力は，指定された入力条件をもとにグラフィック端末から入力されたデー

タを指定変数にセットする.

本節では，グラフ入力に関する代表的な文として，PROCESS 文，INPUT 文，ANYON 文の機能を概説する.

```
PROCESS      var
  OPERATE =オペレーションモード          @  入力方法の指定
  [MESSAGE=ガイダンスメッセージ]          @  ガイダンスメッセージ
  [MENU   =パラメタメニュ]               @  メニュ文字列
  [HEADER =Write & Read ヘッダ]          @  プロンプト
  [CHECK  =CHECK 関数, …]               @  検査用関数
  [ECHO   =ECHON | ECHOF]               @  標準エコー表示制御
ENDP
```

PROCESS 文は，入力条件（1 G 入力の実行に必要な情報）を変数（var）の変数属性として定義する宣言文である.

入力実行文（INPUT 文，ANYON 文）では，この入力条件属性が設定された変数を指定する.

CHECK 関数は，入力されたデータの意味検査を行う関数であり，データの入力直後に実行される.

```
INPUT  var [, END=ENDOK | ENDNG]
```

INPUT 文は，入力を実行する文である.

INPUT 文は，指定変数（var）に設定された入力条件の下で，その変数の型のデータを入力して，その入力値を変数にセットする.

END オプションは，データを繰り返し入力する際の終了指示用のメニュ（『繰り返し終了』）を表示するか否かのオプションである.

INPUT 文で指定された入力は，『順序付き入力』である.

```
ANYON (var {, var})
```

ANYON 文は，『任意入力』を開始宣言する文である.
指定された変数（var）は『任意入力』として宣言され，これ以降の任意の INPUT 文実行時に入力可能になる.

INPUT 文実行時，システムは INPUT 文で指定したデータと ANYON 文で指定したデータの入力を複数同時に入力待ちする. このとき，INPUT 文で指定したデータが入力された場合，実行は INPUT 文の直後に移るが，ANYON 文で指定したデータが入力された場合，実行の制御は移動せず，再入力待ちになる.

3.2 システム機能インタフェース

システム機能インタフェースは，システム固有の機能を実現する処理とインタフェースを採る文である. コマンドモデルにおける機能実行部は，いくつかのシステム機能インタフェースを組み合わせて実現される.

CAD 用言語では，システム機能インタフェースとして以下の 3 種類の文がある.

1) 基本コマンド実行文(#文)……基本コマンドは，システムごとに設定された命令であり，実行に必要な情報を受け取り，その機能に対応した処理を行う．基本コマンド処理には入出力処理は含まれない．基本コマンド実行文は，コマンド名と，実行に必要な情報を指定して実行を指示する文である．

2) MFL (Macro Fortran Library) 実行文 (call 文)……MFL は，基本コマンドの実行結果を参照したり，各種計算をしたり，システムの状態を問い合わせたりする FORTRAN のサブルーチンである．MFL 実行文は MFL の実行を指示する文である．

3) マクロコマンド実行文(##文)……マクロコマンドは，既存の機能を組み合わせたコマンドであり，基本コマンドと MFL を組み合わせて作成される CAD 用言語で記述されたサブプログラムである．マクロコマンド実行文は，このサブプログラムの実行を指示する文である．

3.3 円弧定義コマンドシナリオ例

P1, P2, P3 の 3 点を入力し，P1, P2 を端点として P3 を通過する円弧を定義するコマンドシナリオの例を図 2 に示す．


```
macro  DARC
  real   TOL=0.01
  point  P1, P2, P3
  id     ID1
  process P1              @P1 の入力条件
    ope=pick/key          mes="P1 を入力してください"
  process P2              @P2 の入力条件
    ope=pick/key          mes="P2 を入力してください"
    che=DUPL (P2, P1)     @P2≠P1 検査
  process P3              @P3 の入力条件
    ope=pick/key          mes="P3 を入力させてください"
    che=DUPL (P3, P1, P2),        @P1≠P2≠P3 検査
        =NLIN (P3, P1, P2, TOL)   @P1, P2, P3 が直線上にない
  process TOL ope=key hea=(1, 1) 'TOL='
prgstart
  anyon (TOL)                 @トレランスの任意入力を可能にする．
  input  P1                   @P1 の入力
  input  P2                   @P2 の入力
  input  P3                   @P3 の入力
  ID1=#DARC/P1, P2, P3, TOL   @円弧定義機能
  display                     @定義した円弧の表示
endmacro
```

図 2 『円弧定義』コマンドシナリオ例

Fig. 2 Example: Command scenario of ARC-DEFINE command

4. UIM (User Interface Manager)

UIMは，コマンドモデルにそったユーザインタフェースの実現を制御・管理するソフトウェア・モジュール群である．

UIMは，ユーザインタフェース作成者が記述したコマンドシナリオを解釈しながら，コマンドの実行を制御する．

以下，UIMの機能概要，モジュール構成およびその制御について述べる．

4.1 UIMの機能

UIMの機能は以下の通りである．

①	システム制御	
	システム開始	システム環境の初期化，DB OPEN…
	システム終了	システム環境の終了処理，DB CLOSE…
②	コマンド制御	
	コマンド選択	メニュの制御，コマンドの選択
	コマンド開始	コマンド選択直後のコマンド実行環境の初期化 コマンドシナリオのロード
	コマンド終了	コマンド終了直前のコマンド実行環境の終了処理 凍結されていたコマンドの状態の復帰
	コマンド割り込み	実行中のコマンドの状態の保存とコマンドの凍結
③	コマンドシナリオの解釈	コマンドシナリオの解釈および実行
④	入力制御	
	入力キャンセル	直前の入力値の無効処理と入力状態の復帰
	実行キャンセル	直前の実行結果の無効処理
	1G入力	入力条件の解釈，入力および制御
⑤	システム機能処理の起動	
	基本コマンド処理の起動	コマンドの実行に必要な情報の設定 基本コマンド処理のディスパッチ 実行履歴の保存
	MFL処理の起動	MFLの実行に必要な情報の設定 MFL処理のディスパッチ

4.2 UIMモジュールの構成

UIMは，基本的に10のモジュールと，各モジュールを制御する『全体制御』から構成される．モジュール構成は図3に示す通りである．

4.3 モジュールの制御

UIMに組み込む機能は，全体制御が図3に示した各モジュールを巧妙に制御しながら実現される．全体制御の主処理は図4に示す通りである．

全体制御は，次に実行すべきモジュールを検査し対応するモジュールを起動する．そのモジュール処理終了後，処理結果を基に次に実行すべきモジュールを決定する．プログラムは，各モジュールの処理が並列に並んだ単純な処理構造である．全体制御

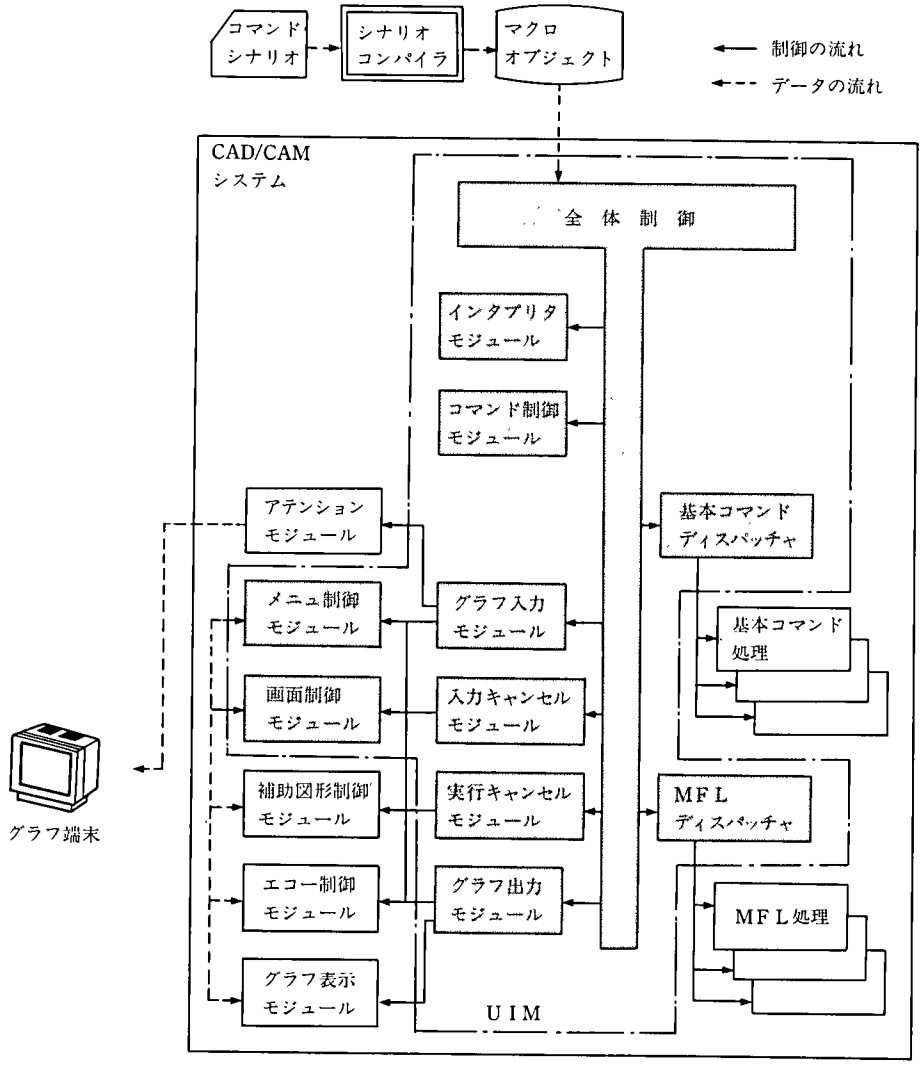


図 3　UIM モジュール構成

Fig. 3　Configuration of UIM modules

は，図5のモジュールの実行の流れになるように，次に実行すべきモジュールを決定する．

5. おわりに

今回作成した UIM は実用に供しており，複数のシステムにも適用されている．この UIM による効果は以下の通りである．

1)　ユーザインタフェースの実現処理をシステムに組み込むことにより，作成者の負荷を軽減し，作成の工数が低減された．

2)　システム使用者が，ユーザインタフェースを設計・実現することが可能となった．

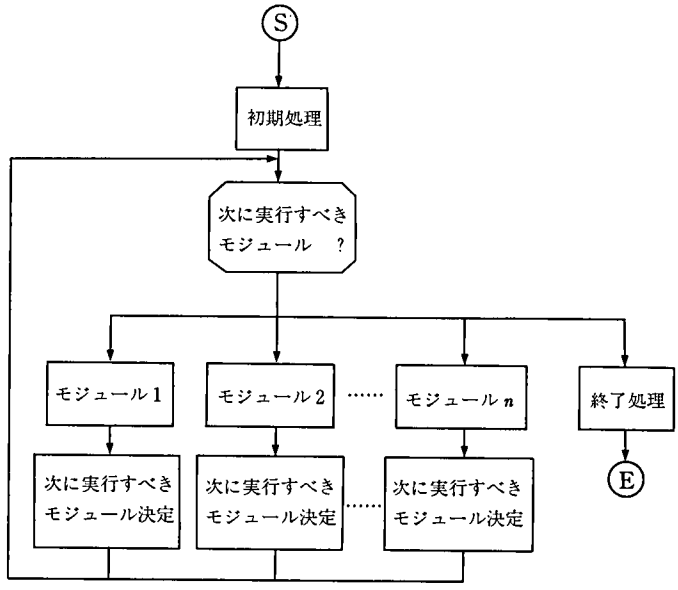


図 4 全体制御の処理概要

Fig. 4 Process flow of module controller

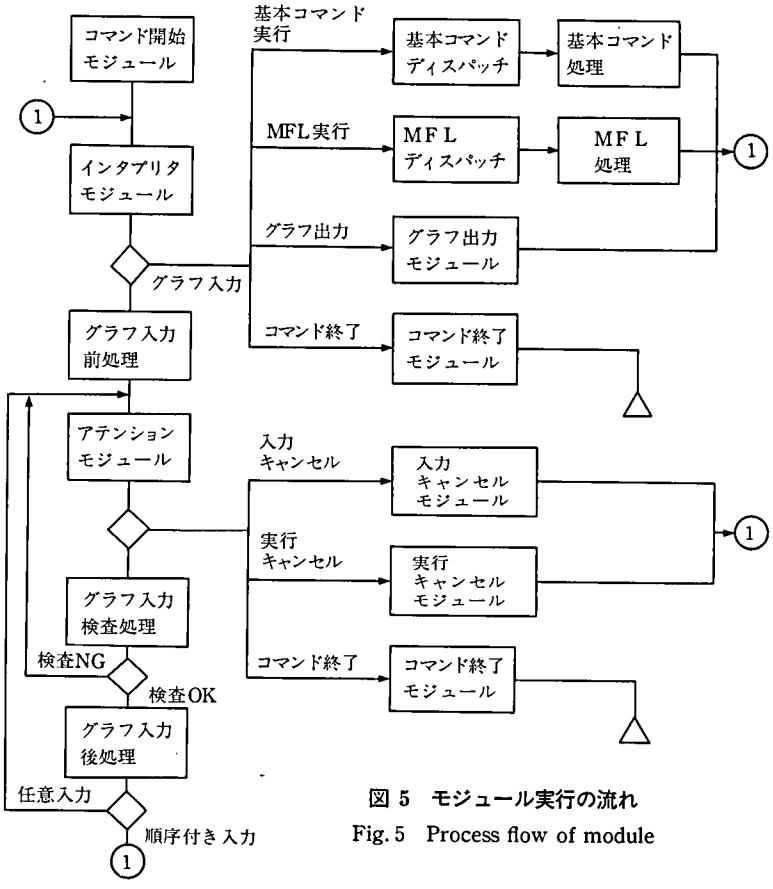


図 5 モジュール実行の流れ

Fig. 5 Process flow of module

3) 他のシステム開発にUIMを適用することにより，開発工数の削減および開発期間の短縮を実現することができた．

4) コマンドモデルにそった操作の統一化により，システム間の操作性のギャップをなくすことができた．

ユーザインタフェースを取り巻く環境において，以前はCAD/CAMシステムを評価する際，機能の豊富さが最大の評価ポイントであった．システムの開発者は，限られた資源を機能実現のために優先的に割り当てたり，ユーザインタフェースの処理効率の向上のため，あえて操作性を犠牲にすることもあった．

しかしながら最近では，ハードウェア能力の向上により，ユーザインタフェースを工夫する環境が充実し，システム使用者の増加に伴い，操作性の向上の必要性も増してきた．システムの評価は，機能の豊富さだけでなく，操作性（ユーザインタフェースの善し悪し）も重要なポイントとなってきた．さらに，EWSのウインドゥ・システム等のGUI（Graphical User Interface）の発展につれて，ユーザインタフェースの重要性も増すと思われる．

今後は，システムの操作性向上に寄与するアイデアをコマンドモデルやUIMに採り込んで，使用者のニーズに合致したシステムのベースとして発展させていきたいと考えている．

参考文献

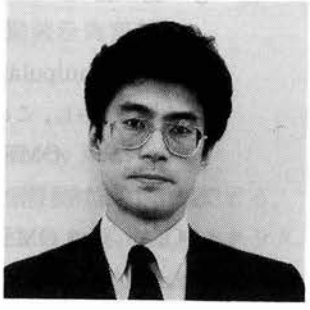
［1］ 守屋慎次，「ユーザインタフェースおよびその管理システムと標準化」，PIXEL No. 51，1986，pp.61～66.

［2］ 宮地恵美，「CAD/CAMシステムを業務に適用させるためのプログラミング言語UDL」，ユニシス技報通巻19号，1988，pp.93～107.

［3］ H. R. Harton, D. Hix,「Human-Computer Interface Development : Concepts and Systems for Its Management」, ACM computing surveys Vol. 21, No. 1, March 1989, pp. 5～92.

執筆者紹介 松　林　　毅（Takeshi Matsubayashi）

1956年生．1980年慶応義塾大学大学院工学部管理工学科修士課程修了，同年日本ユニシス(株)入社．製図，設備配置，ソリッドモデラ等の専用CAD/CAMシステムの開発・保守を経て，現在CAD/CAMシステム1部に所属．新CAMシステムの開発に従事．



OMS/B——直接操作型インタフェース記述システム

OMS/B——A Descriptive System for the Direct Manipulation Interface

川 辺 治 之

要 約 近年，ワークステーションを始めとする高精度表示装置とポインティングデバイスを用いた計算機の普及に伴い，この上の対話型システムのユーザインタフェースとして，直接操作型インタフェースが注目されている．OMS(Object Manipulating System)は，この視覚的対話システムを記述するための方法論の確立を目指すものである．

本稿では，このOMSの基本システムとして，その直接操作の対象をCLOS(Common Lisp Object System)のオブジェクトに限定したOMS/Bの概要について，例題を交えて紹介する．OMS/Bの開発はKS-300シリーズ上で行ったが，CLOSとCLXを用いて記述しているため，容易に他のCommon Lisp処理系に移植できる．

Abstract With a recent growing acceptance of the computing system, including the workstation, equipped with a high-resolution display and a user-friendly pointing device, the direct manipulation interface has become a focus of attention as a user interface operable on the interactive system. The object manipulation system (OMS) is intended to form a methodology by which to describe this graphics-based interactive system.

This paper provides an overview of OMS/B, the basic system of OMS, with some examples also included, where the objects to be directly manipulated are restricted to those of the Commom LISP Object System (CLOS). Although its development was done on the KS-300 series, OMS/B is so made as to allow an easy transplantation to other Common Lisp implementations because it is described with the use of CLOS and CLX.

1. はじめに

高精度表示装置とポインティングデバイスを用いた直接操作型インタフェース(Direct Manipulating interface)は，対話型システムのインタフェースとして優れている[1]．しかし，この優れた特性を持つ視覚的対話システムを記述する方法論は確立されていない．OMS(Object Manipulating System)[7]は視覚的対話システムの記述方法論の確立を目指すものである．

OMS/BはOMSの基本システムであり，その直接操作の対象をCLOS(Common Lisp Object System)[3]のオブジェクトに限定したものである．OMS/Bは移植性を高めるために実現言語としてCommon Lisp[2]を，ウィンドウシステムとしてX Window System*[4]を用いている．

本稿では直接操作型インタフェースを持つ対話システムをOMS/Bで記述する場合の基本概念である表示型を中心に例をあげながら説明する．

2. 基本概念

OMS/BはLispオブジェクトのディスプレイへの表示方法と，ポインティングデバ

* X Window SystemはMITの登録商標である．

イスでその表示を指示しながらの操作方法を容易に記述できるインタフェースを提供する．画面上に現れるオブジェクトの表現を表示(view)と呼ぶ．そして，一つのオブジェクトに対して，同時に複数の表示を対応させることができる．これは，一つのオブジェクトにも，さまざまな表示方法が考えられるからであり，そのオブジェクトが表示される場所によってそれに適した表示方法は異なることが普通である．このオブジェクトの表示方法を表示型(viewtype)によって定義する．OMS/B ではオブジェクトを表示する時に，表示型によって表示方法を指定する．

また，使用者はポインティングデバイス等を用いてこのオブジェクトの表示を指示し，そのオブジェクトに対して操作を行うことができる．

3. 表　　示

オブジェクトの表示方法，および表示をポインティングデバイスで操作する方法を規定するものを表示型(viewtype)と呼ぶ．表示型はクラスに対して定義することができ，そのクラスに属するオブジェクト（そのクラスのインスタンス）を表示する時に用いられる．また，同名の表示型を複数のクラスに対して定義することができる．この場合には，あるクラスのインスタンスが，その表示型で表示する時には，このクラスのスーパークラスのうちで，その表示型が定義されている最も特定的なクラスに定義されている表示方法が用いられる．

たとえば，2次元平面上の点を表すクラス point に対しては直交座標形式で表示する cartesian-view や極座標形式で表示する polar-view が考えられる．この時クラス point には二つの表示型 cartesian-view と polar-view を持つという．このように，一般には一つのクラスに対して複数の表示型を定義することができる．

また，ある特定のオブジェクトに対してのみ有効な表示型を定義することもできる．クラス C のインスタンス i を表示する時には，クラス C あるいはオブジェクト i の持つ表示型のうちの一つを指定する．

3.1 表　示　型

表示型はマクロ defview を用いて定義する．

defview の構文は，

defview(var parameter-specializer-name)&body body

である．body 中では var によって表示するインスタンスを参照することができる．parameter-specializer-name はクラス名または(eql form)であり，前者はそのクラスに対して表示型を定義し，後者は form と eql なインスタンスのみが表示可能な表示型を定義する．body はマクロ make-view を用いて表示をどのように行うのかを記述する．

make-view の構文は，

make-view form &optional viewtype &rest options

であり，form を評価した結果を表示型 viewtype に従って表示する．viewtype が省略された場合にはデフォルトの表示型を用いる．options は表示の位置等を指定するのに用いる．正確にはマクロ make-view はその第1引数で示された form の値を表示するのではなく，その form で表現される場所の内容を表示しているのである．そしてそ

の form が setf 可能な場合に，ポインティングデバイスを用いてその場所の値を変更できるようになる．

3.2 表示の整形（アライメント）

defview の body 中で記述されている make-view によって作られた複数の表示を整形するためにマクロ with-aligned がある．

with-aligned の構文は以下の通りである．

with-aligned(direction &optional alignment tabular)&body body

direction はキーワード：vertical または horizontal であり，body 中で記述された make-view によって作られる複数の表示を並べる方向を規定する．direction が：vertical ならば，alignment は：right，：center，あるいは：left のいずれかのキーワード，direction が：horizontal ならば，alignment は：top，：center，あるいは：bottom のいずれかのキーワードで，それぞれ，body 中で記述された make-view によって作られる複数の表示の位置を規定する．さらに，tabular として上記キーワードのリストを指定することにより，with-aligned は defview の body 中で入れ子にして用いることとあわせて，2次元の表形式でも表示することができる．

また何も表示しない領域を作るためにマクロ

make-fill &optional &key width height

を用意している．make-fill を make-view と合わせて使うことにより，好みの表示形式を定義することができる．

4. 表示型の定義例

4.1 2次元平面上の点

たとえば，先に述べたクラス point は以下のように定義できる．

```
(defclass point ( )
  ((x  :type      number
       :initform  0
       :initarg   :x
       :accessor point-x)
   (y  :type      number
       :initform  0
       :initarg   :y
       :accessor point-y))
  )
```

これにより，クラス point のインスタンス p に対して，x 座標，y 座標を表すスロットの値をそれぞれ(point-x p)，(point-y p)で参照することができる．これらのメソッドを用いてクラス point に対して，直交座標形式による表示型 Cartesian-View を定義する．

```
(defview cartesian-view (p point)
  (with-aligned (:vertical  :left  '(:left  :right))
    (with-aligned (:horizontal)
      (mape-view "x 座標")
      (make-view (point-x p)))
```

```
(with-aligned (:horizontal)
  (make-view "y 座標")
  (make-view (point-y p)))))
```

次にこのインスタンス p に対して極座標形式で参照するためのメソッドを定義する.

```
(defmethod point-theta ((p point))
  with-slots (x y) p
    (atan (/  y x))))

(defmethod (setf point-theta) ((p point) theta)
  (with-slots (x y) p
    (let (((rho (sqrt (+ (expt x 2) (expt y 2)))))
      (setf x (* rho (cos theta))
            y (* rho (sin theta)))))
  theta)

(defrethod point-rho ((p point))
  (with-slots (x y) p
    (sqrt (+ (expt x 2) (expt y 2)))))

(defmethod (setf point-rho) ((p point) rho)
  (with-slots (x y) p
    (let ((ratio (/  rho (sqrt (+ (expt x 2) (expt y 2))))))
      (setf x (* x ratio)
            y (* y ratio))))
  rho)
```

これらのメソッドによって，偏角 theta と半径（原点からの距離）rho はそれぞれ (point-theta p), (point-rho p) として参照することができ，インスタンス p をあたかも偏角 theta と半径 rho をスロットして持つかのように扱うことができる．そして，表示型 polar-view は以下のように定義される．

```
(defview polar-view (p point)
  (with-aligned (:vertical :left '(:left :right))
    (with-aligned (:horizontal)
      (make-view "偏角")
      (make-view (point-theta p)))
    (with-aligned (:horizontal)
      (make-view "半径")
      (make-view (point-rho p)))))
```

4.2 リストの表示例

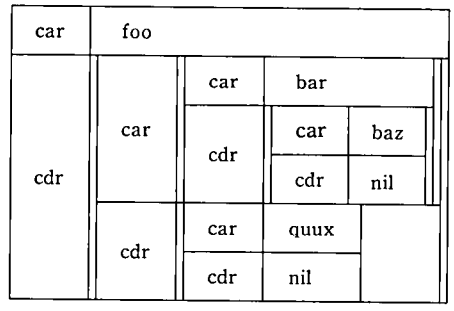
また nil でないリストは cons セルの集まりと見ることもできるし，文字列やベクタと同様に列と見ることもできる．これらの見方をそれぞれ cons-view, sequence-view として以下のように定義する．

```
(defview cos-view (cons cons)
  (with-aligned (:vertical :left '(:left :left)
```

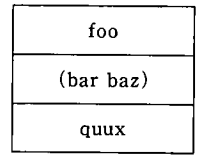
```
(with-aligned (:horizontal)
  (make-view 'car)
  (make-view (car cons) (if (consp (car cons))
                            'cons-view
                            'default-view)))
(with-aligned (:horizontal)
  (make-view 'cdr)
  (make-view (cdr cons) (if (consp (cdr cons))
                            'cons-view
                            'default-view)))))

(defview sequence-view (seq sequence)
  (with-aligned (:vertical :center)
    (dotimes (i (length seq))
      (make-view (elt seq i)))))
```

これを用いて，(foo (bar baz) quux) を cons-view で表示すると



のようになり，sequence-view で表示すると



のようになる．

5. 表示型の継承

あるクラスCのスーパークラスに対して定義された表示型を用いて，クラスCのインスタンスを表示することができる．このため，OMS/Bでは以下に述べるような標準的な表示型が容易に定義できる．

クラスTは表示型 default-view を持つ．すべての Lisp オブジェクトはクラスTに属しているので，表示型 default-view を用いて表示することができる．表示型 default-view はオブジェクトをその印字表現で表示する．

standard-class のインスタンス(ただし standard-object 自身は除く)のすべてのス

ーパークラスであるクラス standard-object は表示型 instance-view を持つ．表示型 instance-view はインスタンスをそのクラス名とともにすべてのスロット名とその値を表示する．

表示型 instance-view の定義は以下の通りである．

```
(defview instance-view (obj standard-object)
  (with-aligned (:vertical  :left)
    (make-view obj) ; default view
    (with-aligned (:vertical  :left  '(:left  :left))
      (dolist (slotd (class-slots (class-of obj)))
        (let ((slot-name (slot-definition-name slotd)))
          (with-aligned (:horizontal)
            (make-view slot-name)
            (if (slot-boundp obj slot-name)
                (make-view (slot-value obj slot-name))
                (make-unbound-view (slot-value obj slot-name)))))))))
```

同様に defstruct で定義されるすべての構造体クラスのスーパークラスであるクラス structure-object に対しても，そのインスタンスを構造体名とともにすべてのスロット名と，その値を表示する表示型 structure-view が定義されている．

6. イベントとアクション

このように defview を用いて表示された表示に対して，使用者が与える指示としては，①ポインティングデバイスの移動，②ポインティングデバイスのボタンの up・down，③キーボードのキーの up・down，等が考えられる．このようなポインティングデバイスやキーボードの状態の変化をイベントと呼び，イベントが発生した時に実行すべき処理をアクションと呼ぶ．

OMS/B では，以下のアクションを組み込みで提供している．

① 表示の移動
② 表示型の変更
③ 表示の消去
④ 表示の複製
⑤ インスタンスの構造の修正

OMS/B ではマクロ define-mouse-action を用いて表示型に対するアクションを記述する．define-mouse-action の構文は以下の通りである．

define-mouse-action key string function

key で記述されたイベントが発生すると，イベントが発生した表示(イベントが発生した時にポインティングデバイスを含んでいる表示) を引数として function で指定された関数が呼び出される．string はメニューを表示する時に用いられる．

7. OMS/B の実装

OMS/B は移植性を考慮に入れ，可能な限り標準的なシステムを用いて実装されている．ウィンドウシステムは MIT の X Window System を使用し，Common Lisp と

XのインタフェースはCLX[5], CLUE[6]を用いた．また，OMS/Bの開発はUNISYS KS-300シリーズ上のCommon Lispを用いて行った．

表示はCLUEのcontactクラスとして定義されている．Contactは一般のウィンドウシステムでいうところのウィンドウに相当するクラスである．またマクロdefviewはそれが定義されたクラスと表示型名に対して特定化(specialize)されたメソッドmake-view-internalとして定義される．総称関数make-view-internalは表示型のインスタンスが作られる時に呼ばれ，その表示で表示する文字列とその場所に関する情報を集める．

8. 今後の課題

OMS/Bの実現すべき今後の課題として，以下のことが挙げられる．

1) メニュー……表示の変更等で用いているメニューは，CLUEが提供するメニューインタフェースを利用しているが，メニューもあるクラス（たとえばリスト）の表示型として表現可能と思われる．そうすることにより，メニューに対しても，より豊富な機能を使用者が実現することができるであろう．

2) 表示の更新……OMS/Bが提供するポインティングデバイスによるオブジェクトの構造の変更については，そのオブジェクトの他の表示も同時に更新を行う．しかし，一般には使用者が定義した関数の副作用として起こるオブジェクトの構造の変更は表示には反映されない．この場合には，使用者は明示的に画面の再表示を指示しなければならない．これを避けるには，表示しているオブジェクトに対して，定期的に構造に変更がないかどうかを調べる必要がある．しかしこれを素直に行ったのではシステムの負荷が大き過ぎるので，別な方法を考える必要がある．

3) 図形表示……OMS/Bでは表示は文字列だけに制限したが，図形等も表示の一部として含めることができれば，利用範囲はさらに広がるであろう．

9. おわりに

本稿は，通商産業省 工業技術院電子技術総合研究所 言語システム研究室での技術指導のもとに行った研究結果に基づいている．OMSの基本システムとしてのOMS/Bは，言語を容易に拡張できるCommon Lispをベースに設計したが，表示型の基本概念は言語とは独立であり，他のプログラミング言語上でも十分に展開できると思われる．

本稿がユーザインタフェースを設計する上での一助になれば幸いである．最後に多くの貴重な指導助言をいただいた二木厚吉言語システム研究室長ならびに戸村哲技官に感謝の意を表したい．

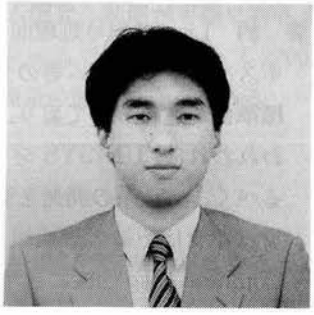
参考文献 [1] B. Schneiderman, "Direct Manipulation: A Step Beyond Programming", Language, IEEE COMPUTER, Vol. 16, No. 8, 1983, pp. 57～69.

[2] G. Steele, "Common Lisp: The Language, 2nd Edition" Digital Press, Burlington, MA, 1990.

[3] D. Bobrow, et al., "The Common Lisp Object System Specification" Lisp and Symbolic Computation, Vol. 1, No. 3/4, 1989, pp. 245～394.
[4] J. Gettys, et al., "Xlib-C Language X Interface, Version 11",
[5] R. Scheifler, et al., "CLX Common LISP X interface Programmer's Reference".
[6] K. Kimbrough, L. Oren, "Common Lisp User Interface Environment", Texas Instruments Inc., April, 1988.
[7] 川辺治之, 戸村哲, 二木厚吉 "Common Lisp Obtect System に基づく Object Manipulating System"日本ソフトウェア科学会第5回大会論文集, 1988.

執筆者紹介 川 辺 治 之 (Haruyuki Kawabe)

1961年生. 1985年東京大学理学部数学科卒業. 同年日本ユニシス(株)入社. 知識システム部所属, KS-300シリーズの日本語機能開発・保守, 1100/2200上のLisp処理系の開発に従事. 現在, 開発1課所属, KEE/U 6000の日本語化に従事.



UNISYS シリーズ 2200/1100 の Common Lisp 処理系

An Implementation of Common Lisp on UNISYS Series 2200/1100

大 田 一 久

要 約 Lispは記号処理向きの言語として他の言語にはない特徴を持っており，数式処理，エキスパートシステム等の分野で使用されてきている．近年，Common Lispとして言語仕様の標準化が進められており，ワークステーションを中心に処理系も徐々に市場に出てきている．われわれは，UNISYS シリーズ 2200/1100 上で Common Lisp アプリケーションを稼働させるべく，処理系の開発を進めている．2200/1100 Lispは，Lisp向きの仮想機械を実装し，コンパイラはその仮想機械をターゲットとするというアプローチをとっている．その他に，Common Lisp仕様に合った各種のライブラリ，およびLispの重要な特徴であるインタプリタをLisp自身で開発している．

本稿では，Lispについて簡単に述べ，開発中の2200/1100 Lispに関して，処理系の内容について紹介する．

Abstract Designed for symbolic computations and possessed of many distinctive features unavailable in any other programming languages, Lisp has come into use in such areas as symbolic algebra, expert systems, and so on. In recent years, Common Lisp, a dialect of Lisp, is on the way toward the standardization of its language specifications, and several good implementations of it have already appeared on the market by degrees for main use on workstations.

Nihon Unisys, Ltd. has been in pursuit of an implementation of Common Lisp with a view to making Common Lisp applications operate on UNISYS Series 2200/1100. The approach employed for the 2200/1100 Lisp is such that it has a Lisp-oriented virtual machine implemented, allowing a compiler to hit the virtual machine as its target. Along with this, Nihon Unisys has developed a variety of Lisp libraries which meet Common Lisp specifications and a Lisp interpreter which serves as an important feature, all using Lisp itself.

This article presents a brief introduction to the Lisp language, and an overview of the 2200/1100 Lisp now under development.

1. はじめに

エキスパートシステムは各分野で実用化の段階を迎えているが，利用者の側から見ればエキスパートシステムを容易に利用できる段階には至っていない．その大きな原因の一つとして，エキスパートシステムを支える重要な技術である記号処理が広く利用できる状況にないことが挙げられる．とくに，記号処理を得意とするLisp言語に関しては，実用となる処理系はLisp専用のハードウェアの他には，ワークステーション上の処理系がようやく現れてきたところである．

一方，メインフレーム上にはこれまでに蓄積された膨大なアプリケーションの資産があり，それらを生かしてエキスパートシステムの実用化を考えるならば，メインフレームでの展開を考えるのはごく自然な発想である．ただし，エキスパートシステム

に限らず，システム開発の環境としてはワークステーションの方がメインフレームより優れており，ワークステーションで開発を行い，メインフレームでその運用を行うという形態が合理的である．その場合には，ワークステーションとメインフレームで同一のプログラムが稼働できるならば，実用化の過程での作業が軽減される．ここにメインフレームで Lisp 処理系を稼働させる意味がある．

当社では Lisp 専用ワークステーション[1]の商品化を行っているが，より幅広いプラットフォームで記号処理，ひいてはエキスパートシステム開発，利用が可能になるよう準備を進めている．その一環として当社のメインフレームであるシリーズ 2200/1100 上にアプリケーションの実行を目的とした Lisp 処理系の開発を行っている．

2. Lisp について

2.1 Lisp の特徴

記号処理という時の記号とは，Lisp では文字の列で表される名前のことであると考えてよい．この名前を Lisp ではシンボルと呼び，処理の基本的な単位であり，一つのデータ型である．もう一つの基本的なデータ型として，データを 1 次元に並べたリストがある．シンボルをリストを用いて組み立てていくことにより，木構造あるいはさらに複雑なデータ構造を構成することができる．

Lisp の他のプログラム言語と比べて際だった特徴は次に示す通りである．

1) プログラムとデータの表現が同一……Lisp ではプログラムを構成する文あるいは式はリストで表され，プログラム中の名前はシンボルである．したがって，プログラムをデータとして扱うことが容易にできる．これは，知識ベースを構築する際にプログラムを知識として扱うような強力な知識表現を可能にする．

2) 関数の定義を動的に変更可能……Lisp ではインタプリタとコンパイラを持ち，それらを併用することが可能である．このため，インタプリタから動的に関数の定義を変更することができる．これは，プログラムを段階的に開発していく場合に非常に有効である．

3) データオブジェクト自身が型の情報を保有……多くのコンパイラを使用するプログラム言語は，データを格納する変数にデータ型の属性が付随している．これに対して，Lisp ではデータ自身が型を持っており，型に応じて処理を変えるようなプログラムを書くことができる．

4) 強力なマクロ機能を保有……プログラマ自身がマクロを定義することによって，言語の拡張を自由に行うことができる．マクロはソースプログラムの一部として展開されるので，関数として定義しにくいような処理を定型化することができる．

5) メモリ管理機構を処理系が組み込みで保有……一般に，Lisp のプログラムではメモリの割り当てをプログラマが陽に意識する必要はなく，データオブジェクトが作られる時にメモリが自動的に割り当てられる．また，他から参照されなくなったデータオブジェクトは，ガーベジ・コレクションによって自動的に回収される．このことはプログラマの負担を大きく軽減することができる．

2.2 Lispの応用

Lispは最初は定理証明を行うプログラムを作るために設計された言語であり，そのためのデータ構造として，シンボル*とリストが用意された．その後は人工知能関連の分野を中心に研究開発に使われ，ゲーム，ロボットのシミュレーション，談話理解，さらにはエキスパートシステム等の開発に使われてきた．数式処理の分野では，記号微分，記号積分等の記号計算のためにLispが使われ，いくつかの著名なシステムが開発されてきている．

この両分野とも，数値表現では取り扱いにくい対象を処理するために記号処理が重要な働きをしており，Lispの性質をよく利用している．もっとも，計算機の能力が低かった時代には，本質的に計算機の能力を要求する記号処理は，限られた形でしか使用できなかった．しかし，計算機の能力の向上は記号処理の応用しうる分野を徐々に拡大しつつある．

2.3 Common Lispと標準化

Lispは研究分野での利用が中心であったため，処理系あるいは言語仕様が研究者自身の手で拡張あるいは改良されてきた．このため，数多くの方言を生むこととなり，プログラムの移植性を阻害することになっていた．この状況を改善するために，1980年代の中ごろから米国の研究者，および企業が中心となって，Common Lisp[3],[4]と呼ばれる共通の言語仕様の検討が行われてきた．

80年代の後半にはこの仕様が米国では事実上の標準となり，さらにはこれに基づいてANSIの規格が制定されることになった．日本でも，Common Lispを採用する機運が高まり，この仕様に基づいたJIS規格の作業が進められている．

Common Lispは，豊富なデータ型，強力な制御構文，多数のライブラリ関数をLispの持つ枠組にうまく取り込んでいる．しかしながら，方言の共通部分としての性格から，非常に大規模な言語仕様となっている．

2.4 Lispのデータ型

Lisp(LISt Processorの略)で取り扱うデータは，その名前が示す通りリストが中心である．Common Lispでは，リスト以外にも多くのデータ型を用意しており，その一部を以下に紹介する．

まず，記号処理の基本となるシンボルである．これは名前を持ち，同じ名前を持つシンボルの唯一性を処理系が保証している．このため，アプリケーションの対象分野の事象を名前で表すために用いられ，プログラムの中での変数名，関数名を表すためにも用いられる．単なる文字の列である文字列もシンボルとは別のデータ型として用意されている．

当然のことながら数値も扱うことができ，整数，いくつかの精度の浮動小数点数，および複素数がデータ型として用意される．Lispの特徴として，多倍長整数，有理数が用意されており，数式処理において威力を発揮している．

Common Lispでは多次元の配列が用意されており，文字列は文字を要素とする1次元の配列である．とくに，1次元の配列をベクタと呼び，リストと合せて要素の列として取り扱うことができるような関数が用意されている．

---

* シンボル：当時はアトムと呼ばれた．

Lisp では，これらのデータオブジェクトの外部表現の構文が決められており，プログラムからデータオブジェクトをテキスト・ファイル，標準入出力等に対して読み書きすることができる．また，一度書き出したものを読み込んだ時に，同じデータ構造が作られるように工夫されている．

2.5 Lisp の関数

Lisp は関数型言語としての側面を持っており，プログラムは関数の集まりとして定義される．プログラムの実行は関数を呼び出す式を評価することによって行われる．このとき，関数に引数を与えて呼び出すことを適用と言う．

Lisp の関数はラムダ式と呼ばれる形式をしており，仮引数と本体の式からなる．関数を適用する方法は，ラムダ計算と呼ばれる形式的体系に基づいている[6]．細かい点を無視して説明すると，関数の適用は本体の式に含まれる仮引数の変数を実際に与えられた実引数で置き換えて，さらに評価することによって行われる(図1)．これをラムダ計算の用語でベータ・リダクションと呼んでいる．

プログラムを実行する際には，一般にはどのような順序でベータ・リダクションを行うかの指定が必要である．Lisp では，関数適用の引数を先に評価する Call by Value と呼ばれる方式を採用している．すなわち，ある関数適用を行う場合に，その実引数として与えられている式がさらに関数適用であれば，それを先に評価する必要がある．

コンティニュエーションと呼ばれる概念*を用いると，この順序を明確に表現することができる．コンティニュエーションとは，プログラムの実行中のある時点でそれ以降に実行すべきプログラムの内容を切り離して表したものであり，通常のサブルーチン呼び出しでの復帰番地を表現したものに相当する．Lisp ではこのコンティニュエーションも関数の一種として取り扱うことができる(図2)．コンティニュエーションを用いると繰り返し構造，ブロックとそこからの脱出等も関数適用として表現することができ，処理系を構成する時に各種の制御構造の統一的な扱いが可能になる[5]．

```
(defun foo (x)                    ; 関数の定義
  (+ x 1))

(foo 2)                           ; 関数の適用
    => (+ x 1)                    ; 仮引数の置き換え
    => (+ 2 1)                    ; 本体の評価
    => 3                          ; 結果
```

図 1　Lisp の関数適用

Fig. 1　Function application in Lisp

```
(defun bar (x)                    ; 通常の関数の定義
  (* (+ x 1) x))

(defun cbar (c x)                 ; コンティニュエーション
  (+ #'(lambda (v)                ; 加算を先に評価
         (* c v x))               ; その結果を用いて乗算を評価
     x 1)
```

図 2　コンティニュエーションを用いた関数定義

Fig. 2　Function definition using continuation

* プログラム言語の表示的意味論でのコンティニュエーションと同じものである．

3. 2200/1100 Lisp システム

現在開発中の 2200/1100 Lisp システムは，当社のメインフレームであるシリーズ 2200/1100 上で稼働する Lisp システムである．シリーズ 2200/1100 のアーキテクチャの上に，Lisp 向きの仮想機械を実現し，その仮想機械をターゲットとするコンパイラを開発するというアプローチをとっている．

2200/1100 Lisp システムは，以下に示すような部分から構成されている．

コンパイラ　　：Lisp プログラムを仮想機械で実行可能な形に変換する．
ランタイムシステム：仮想機械の機能を実現する．
ライブラリ　　：Lisp の組み込み関数群．

これを図に表すと図 3 のようになる．

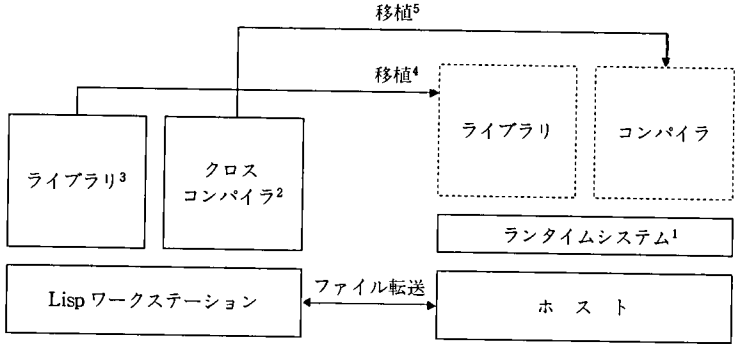


図 3　2200/1100 Lisp の構成
Fig. 3　System structure of 2200/1100 Lisp

3.1 ランタイムシステム

ランタイムシステムは，コンパイラがターゲットとする仮想機械の機能を実現したものである．仮想機械は Lisp で用いるオブジェクトをデータとし，それを取り扱う基本操作を命令とする仮想的な計算機である．以下に仮想機械で扱うデータの形式，仮想機械の構成，仮想機械の命令について説明する．そのあと，関数の構造，関数の呼び出しの方法，メモリ管理について述べる．

3.1.1 データオブジェクトの表現

リストを構成するセル等の Lisp のデータオブジェクトは，通常メモリ上にある大きさの領域を占有し，それに対するポインタとして表現される．これに対して，固定長整数等はメモリ領域をとる必要がなく，ポインタのかわりにデータそのものを持つことができる．前者をポインタオブジェクト，後者をイミディエートオブジェクトと呼ぶ．実行時にデータオブジェクトの型の判定が必要であることから，ポインタに何らかの形でデータ型の情報が含まれているのが普通である．2200/1100 Lisp では，語の一部をデータ型情報のために使用し，残りをアドレスとして使用するタグ付ポインタを用いている(図 4)．

データオブジェクトのためのメモリ空間を大きくとるためには，このアドレスのフィールドをなるべく大きくとることが望ましい．シリーズ 2200/1100 のアーキテクチャでは，アドレッシングはバンク記述子によるバンクの指定と相対アドレスを使用し

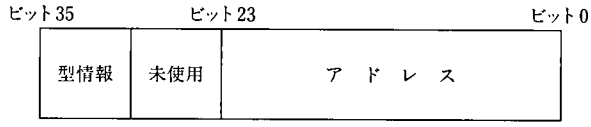


図 4　タグ付きポインタの形式
Fig. 4　Tagged pointer format

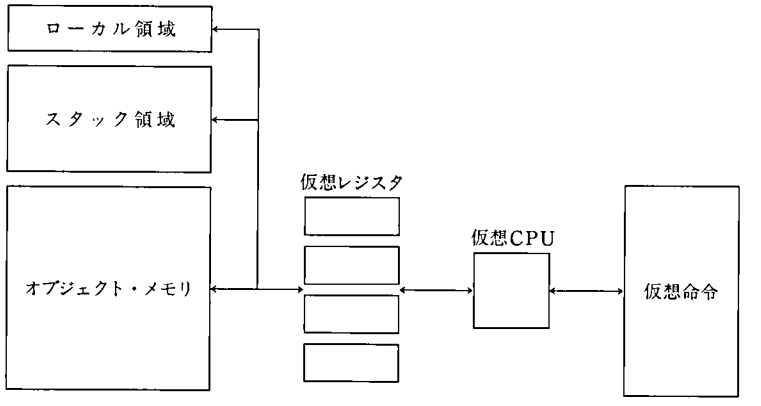


図 5　仮想機械アーキテクチャ
Fig. 5　Virtual machine architecture

なければならないため，データオブジェクトの参照が繁雑になりかねない．シリーズ 2200/1100 の拡張モード[2]では，一つのバンク内で 24 ビットの相対アドレスを使用できることから，現状ではデータオブジェクトを一つの D バンク(データバンク)に置く方法をとっており，最大 16 M 語の空間を使用できるようにしている．

ただし，D バンクを使用する際には実メモリに載せなければならないため，実装されている実メモリの大きさによって実際に使用できる空間の大きさは制約される．これを緩和するために，データオブジェクトの型によって D バンクを分割する方法を検討している．

3.1.2 仮 想 機 械

仮想機械は，仮想レジスタ*，ローカル領域，スタック領域，およびオブジェクト・メモリを持つ計算機として設計されている(図 5)．

データの操作・演算を行うためには，オペランドを仮想レジスタにロードする必要がある．関数の呼び出し・復帰の際に，引数あるいは値を置くためにはローカル領域を用いる．スタック領域は，コンティニュエーション，そのための変数環境，不定個の引数を受け渡すためのリストを作成するために用いる．仮想レジスタは，実際のレジスタの組を用いており，それぞれデータ型の情報およびデータオブジェクトの相対アドレス，あるいはデータそのものを保持する．ローカル領域は D バンクのメモリ領域を使用しているが，これに極力レジスタを割り当てるようにすれば速度の向上が期待できる．

---

* とくに，実際のレジスタと区別するために仮想レジスタと呼ぶ．

```
          $EJECT
.
. Module for function TAILP.
. Entry point F204.
.
$(3)
I_49      CDVTOP                          .
                                          .  Start of code vector for TAILP
E_52      $EQU      F204-FTOP             .
F204                                      .
          VMLOAD    V0, CONT              .
          VMSTORE   V0, ARG2              .
          STKENV    V1, 2                 .  Stack Environment Allocation
          VMLOAD    V2, ARG0              .  SUBLIST
          ENVSET    V2, V1, 0             .
          VMLOAD    V3, ARG2              .  ARGS44
          ENVSET    V3, V1, 1             .
          VMSTORE   V1, ENV               .
          VMLOAD    V0, CONT              .
          VMSTORE   V0, ARG3              .
          VMLOAD    V1, ARG1              .  LIST
          VMSTORE   V1, ARG4              .
FNVAR193                                  .  DO188
          VMLOAD    V2, ARG4              .  L
          ATOM      V2                    .
          JNIL      V2, T206              .
          VMLOAD    V3, ARG3              .  VAL
          POPSTK    V3                    .  Stack Deallocation
          VMLOAD    V0, ARG3              .  VAL
          VMSTORE   V0, FUN               .
          SETNARG   1                     .
          ENVLOAD   V1, 0, 0              .  SUBLIST
          VMLOAD    V2, ARG4              .  L
          EQ        V1, V2                .
          VMSTORE   V1, ARG0              .
          RESUME                          .
T206                                      .
          ENVLOAD   V3, 0, 0              .  SUBLIST
          VMLOAD    V0, ARG4              .  L
          EQ        V3, V0                .
          JNIL      V3, T205              .
          ENVLOAD   V1, 0, 1              .
          POPSTK    V1                    .  Stack Deallocation
          ENVLOAD   V2, 0, 1              .
          VMSTORE   V2, FUN               .
          SETNARG   1                     .
          TLOAD     V3                    .  t
          VMSTORE   V3, ARG0              .
          RESUME                          .
T205                                      .
          NILLOAD   V0                    .  nil
          VMSTORE   V0, ARG5              .
C202                                      .
          VMLOAD    V1, ARG4              .  L
          CDR       V1                    .
          VMSTORE   V1, ARG4              .
          JU        FNVAR193              .
J_50      CDVTAIL                         .  End of code vector for TAILP
$(4)                                      .
.
.
R_51      LCDV      J_50 I_49             .  Code Vector Reference for TAILP
.
. Constants for TAILP
.
.
. Constant Frame Obejct for TAILP
.
.
. Closure object for TAILP
.
F_53      LCLOSE    E_52, 0 R_51 S_NIL, DSYM S_NIL, DSYM S_48, DSYM .
```

図 6　仮想命令の例

Fig. 6　Example of virtual machine instructions

3.1.3 仮想命令

仮想命令は，仮想レジスタとメモリの間のデータ転送を行う命令，データを操作・演算する命令，処理の流れを制御する命令，および他の命令に分けることができる．データ転送命令は転送の相手となるメモリが，ローカル領域，変数環境，グローバル変数の値セル，関数が持っている定数であるかによって対応する命令がある．データ操作命令には，データオブジェクトの内容の参照・更新，データの型の判定，データオブジェクトの作成，数値演算といったものがある．制御命令には，ジャンプ命令，関数呼び出し，スタック操作等がある．

その他に，入出力，オペレーティング・システムコール等のための命令がある．仮想命令は，MASM(Meta Assembler)プロシージャとして作成されており，その中で必要に応じてサブルーチン呼び出しを行っている．仮想命令の例を図 6 に示す．

3.1.4 関数の構造

関数は，クロージャと呼ばれる特殊なデータ構造として表現されている．クロージャは，関数が実行すべき命令の列を含むコードベクタへのポインタ，そのエントリポイント，および関数を実行する際に必要な自由変数の値を含む環境，関数内で用いられる定数の表からなる(図 7)．

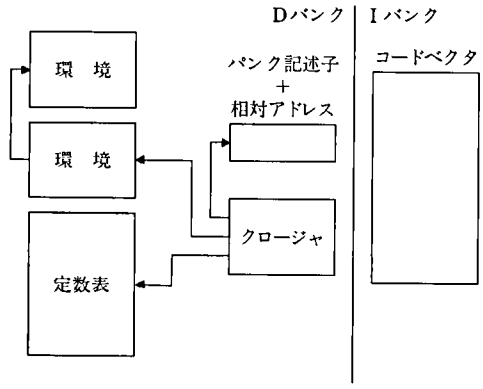


図 7　関数の構造

Fig. 7　Structure of function

コードベクタには，実際にシリーズ 2200/1100 の CPU で実行される機械語命令が含まれているため，その実体は他のデータオブジェクトとは別に複数の I バンク(インストラクション・バンク)に置かれる．このため，コードベクタの実体のアクセスには，バンク記述子と相対アドレスを用いる．また，コードベクタは複数のエントリポイントを持ち，複数の関数クロージャから共有されている．このため，クロージャの中にエントリポイントのオフセットが含まれている．

環境は，関数の中で参照される変数で，関数の外側で束縛されているものの値を保持するためのデータ構造である．プログラムのトップレベルで定義される関数は環境を持たないが，関数の内部で定義されるローカル関数，あるいはコンティニュエーションは，その外側のブロック構造に応じてリンクされた環境を持つ．

3.1.5 関数の呼び出しと復帰

関数を呼び出した後の処理をコンティニュエーションと呼び，それ自体を関数として扱うことができることはすでに述べた．コンパイラは，関数を呼び出す前に必要に応じてコンティニュエーションを作成するコードを生成する．このコンティニュエーションは，呼び出し側の関数とコードベクタを共有しており，環境および実行開始番地が異なっている．

コンティニュエーションの存在は，コンパイラのみが関知しており，プログラムの他の部分から参照されることはない*．したがって，他のオブジェクトのようにメモリ領域を割り当てる必要はなく，スタック領域を使用し，不要になった時点で解放することができる．

ローカル関数，プログラムの中でラムダ式として一時的に作られる関数も同様の扱いをすることができる．ただし関数を値として返すような場合等は，スタックで管理できないので，必要に応じてコンパイラがメモリ領域にコピーするコードを生成する必要がある．

3.1.6 メモリ管理(ガーベジ・コレクション：GC)

Lispでは，データオブジェクトはポインタを介して参照され，一つのオブジェクトが複数のオブジェクトから参照されることが頻繁に起こる．このような環境ではプログラマ自身がデータオブジェクトのメモリ管理を行うことはかなり困難である．このため，Lispでは初期のころから，組み込みのメモリ管理機構が使われてきた．この機構はガーベジ・コレクション(GC)と呼ばれ，他のオブジェクトから参照されていないオブジェクトはもはや不要であると判定し，その領域を再利用できるようにすることによって行われる．GCの良し悪しが，アプリケーションのパフォーマンスに大きな影響を持っているため，各種のGCの方式が研究されてきている[7]．

これらは，GCを行うタイミングによってバッチ型かインクリメンタル型かに分けられる．バッチ型はGCが必要になった時点で，アプリケーションの処理を一時中断し，GCを行うものである．この方式では，対象とするメモリ空間が巨大になってくると中断時間が無視できなくなってくる．インクリメンタル型は，アプリケーションの実行と平行してガーベジ・コレクションを少しずつ行うもので，アプリケーションが長時間にわたって中断されることはない．しかし，並行して行うためのオーバーヘッドから，全体としての実行時間はバッチ型を採用している場合より余分に必要になるといわれている**[5]．

バッチ型のGCを行う前に，データオブジェクトの生存期間と参照関係に関する統計的な性質を利用して局所的なGCを試みる方式も開発されている[8]．

GCは，①マーク・アンド・スイープ方式，②コンパクティング・コピー方式，③リファレンス・カウンティング方式，の3種類に分類できる．

マーク・アンド・スイープ方式は，アプリケーションが使用している生きているオブジェクトすべてに一時的に印を付けた後，印の付いていないオブジェクトを再利用可能なものとして回収する方式である．コンパクティング・コピー方式では，メモリ

---

* Lispの一種であるSchemeでは，このコンティニュエーションをプログラムから参照する方法がある．

** Lisp専用ワークステーションでは，このオーバーヘッドを小さくするためのハードウェア上の工夫がなされている．

空間を二つに分割し，アプリケーションは常にその一方にオブジェクトを作成する．GC は生きているオブジェクトをもう一方のメモリ空間にコピーすることにより，領域の圧縮，および解放を行う．リファレンス・カウンティング方式は，オブジェクトごとに，それを参照するポインタの数を保持しておき，その数が 0 になったオブジェクトは他のオブジェクトから参照されていないので，再利用するというものである．

これらの方式はそれぞれ一長一短があるが，メモリ空間を圧縮・解放できることから，コンパクティング・コピー方式が最近ではよく用いられている．2200/1100 Lisp ではメモリ管理機構はまだ実装されていないが，コンパクティング・コピー方式を採用することを計画しており，仮想命令等はすでに対応している．

3.2 コンパイラ

3.2.1 コンパイラの目的

コンパイラの目的は，リストで表現された Lisp のソースプログラムに対して仮想機械で実行可能な命令およびデータを作り出すことである．この過程で構文上の置き換えを行い，Common Lisp の各種の制御構造をコンパイラが取り扱うことのできる基本的な制御構造に帰着させる．同時に，データオブジェクトに関する操作も仮想命令と 1 対 1 に対応するプリミティブ関数の組み合わせに展開する．また，コンパイル時に得られる情報を元に実行時に行われる計算の一部を行うことである．これはいわゆる最適化であり，実行されない命令を除去したり，定数のみからなる計算式を計算したりすることが一般に行われる．

Lisp の場合には，さらに関数の呼び出しをインラインに展開したりデータ型の情報に基づいて型判定の命令を除去したりすることが行われる．Lisp のソースプログラムはリストで表現されるため，コンパイラを Lisp 自身で記述するのが開発しやすい．2200/1100 Lisp のコンパイラも Lisp で記述している．

3.2.2 コンパイラの処理

2200/1100Lisp コンパイラの処理は，以下の段階を追って行われる．このうち中間コード生成までの段階は，他の Lisp コンパイラでもほぼ同じ処理が行われている*[5],[9]~[11]．

① 予備解析
② ソースプログラムの最適化
③ 関数呼び出しの変換
④ 変数参照の解析
⑤ 中間コード生成，局所最適化
⑥ 仮想命令生成

予備解析では，プログラマが定義したマクロ，およびコンパイラが使用するマクロの展開を行い，名前の系統的な付け換えを行う．ソースプログラムの最適化では，定数式の計算，関数のインライン展開等を行う**．関数呼び出しの変換で，コンティニュエーションを用いた形式に変換し，さらに変数参照の解析を行った後，中間コードを生成する．この段階でさらに局所最適化を行い，仮想レジスタの割り当てを行いな

* 2200/1100 Lisp コンパイラは，主に文献[9]にそっている．
** 評価の順序が結果に影響しないベータ・リダクションの一部をコンパイル時に行うものと考えられる．

がら仮想命令を生成する．

3.3 ライブラリ

仮想機械とそれをターゲットとするコンパイラがあれば，非常に限られたものではあるが，Lispプログラムをコンパイルして実行することができるようになる．Lisp処理系としては，インタプリタおよびライブラリ関数を用意しなくては一人前とはならない．これらのライブラリ関数は仮想命令と直接対応するプリミティブ関数の組み合わせで記述することができ，Lispプログラムとして開発することができる．インタプリタも，リストで表現されたプログラムを取り扱うので，Lispで記述することができる．インタプリタ，ライブラリ関数群を他の言語で記述することも可能であるが，開発，保守のしやすさから2200/1100 LispではLisp自身で開発する方法を採用した．

3.3.1 データ処理ライブラリ

各種のデータ型のデータを取り扱うライブラリとしてCommon Lispが定義しているものには，リスト処理，文字列処理，配列処理，列データの総称関数等がある．リスト処理に関しては，プリミティブ関数として，リストの要素を取り出す関数があり，これを用いてリストの切り貼り，コピー等を行う関数がある．さらに，連想リスト，属性リスト*を取り扱う関数，リストを集合として取り扱う関数がある．

配列は格納できる要素の型によっていくつかの種類があるが，それぞれ1次元の場合の要素の参照はプリミティブ関数である．多次元配列および動的拡張が可能な特殊な配列は，配列制御ブロックと呼ばれるデータ構造と，1次元のベクタで表現しており，これを取り扱う関数はLispで記述されている．文字列に関しては，要素の取り出し・置き換えをプリミティブ関数として仮想機械が実装しており，大文字小文字変換等をライブラリとして用意している．

また，ベクタとリストの区別なく1次元の要素の列として取り扱うことができる総称関数群もある．これは，たとえば与えられた条件の要素を見つけ出したり，取り除いたりするものである．

3.3.2 数値演算ライブラリ

数値演算に関しては，型を限定した四則演算および大小比較等は仮想命令で実現されている．型を混合して演算できるような関数は，Lispで型判定を行うプログラムを書く必要がある．これらは，型宣言がある場合等にコンパイラによってインライン展開することによって，型判定を除去した形に最適化できる場合がある．また，Common Lispの仕様では，対数関数，三角関数等の数学関数も含んでおり，これらの関数もライブラリとして開発する必要がある．

3.3.3 入出力ライブラリ

入出力ライブラリは，データオブジェクトの外部表現と内部表現の間の変換を行う．この他に，ファイルあるいは標準入出力デバイスから文字の形でデータを交換する方法があり，そのためのデータオブジェクトとしてストリームがある．このストリームの処理もLispで記述されており，実際の入出力を行うプリミティブ関数を用いている．Common Lispでは，ファイル名を表すデータオブジェクトとしてパス名と呼ばれるものを使用する．これは，ファイル名をその構成要素に分けて管理するもので，文

* 属性リスト：いずれもキーに対して値を登録しておくようなリストによるデータ構造

字列で表現されたファイル名を構文解析を行って，パス名オブジェクトを作り出す関数等を Lisp で開発している．

3.3.4 その他のライブラリ

Common Lisp ではシンボルの唯一性を管理するための表をパッケージと呼んでいるが，パッケージに関する機能も Lisp で実現している．パッケージは，シンボルの名前をキーとするハッシュ表であり，これに対する検索，登録等が主な機能である．さらに，インタプリタ自身，およびそのためのマクロ展開機能等を Lisp ライブラリの一部として開発している．

4. 開発の形態

2200/1100 Lisp の内容を述べてきたが，本章ではその開発の形態について簡単に解説する．

4.1 開発戦略

2200/1100 Lisp のターゲットは，言うまでもなく 2200/1100 ホストである．しかし，今日開発環境が優れているワークステーションが容易に入手できる状況では，ホスト上で直接開発を行う必然性はない．とくに Lisp の処理系の開発には，Lisp 自身が適していることは処理系の内容についての記述から明らかであろう．したがって，処理系開発のターゲットとしているホスト上に Lisp がない場合，クロス開発の形態をとることになる．

このような事情から，2200/1100 Lisp の開発は，仮想機械の設計ができた段階から，Lisp 専用ワークステーション上でクロスコンパイラの開発を行ってきた．並行して，仮想機械を実装するランタイムシステムの開発をホスト上で行い，中核部分を早期に稼働させることができた．このとき，とくにホスト上での作業を軽減するために，中間コードレベルのシミュレータを作成し，ワークステーション上でコンパイル結果の検証が行えるようにしている．

一方，ライブラリの開発もワークステーションの Lisp 処理系を用いて行い，開発が完了したものから順次ホスト上への移植を行っている．このため，ライブラリのテストもワークステーション上で行えるような環境を用意している．ライブラリが揃ったところで，インタプリタの開発・移植を行い，さらにクロスコンパイラ自身を移植することによって，ホスト上にフルセットの Lisp 処理系が完成することになる．図 3 に，このステップが示されている．

4.1.1 現状と今後の課題

現在のところ，ライブラリを限定すれば Lisp プログラムをホスト上で実行できる段階まで来ているが，完全な Lisp 処理系としての形を成すには至っていない．仮想機械は，数値演算関連の命令が完全には揃っておらず，さらにメモリ管理機構が実装されていない．クロスコンパイラは，型宣言を利用した最適化に対応する必要があるものの，ほぼフルセットの Common Lisp プログラムのコンパイルが可能である．ライブラリ関連では，基本的な入出力，データ処理関連の開発はほぼ終了しており，ファイルシステム・インタフェース，書式付入出力，数学関数等の開発が必要である．また，インタプリタは基本機能は稼働しており，デバッグの目的に使用している．

現在，使用可能な機能に限定すれば，Lispプログラムをクロスコンパイラでコンパイルし，ホストにファイル転送したのち，リンクして実行することが可能である．実際に，ある診断型エキスパートシステムをこの方法によりホスト上で評価している例がある．現状では，クロスコンパイラの出力はMASMソースプログラムであり，アセンブル，リンクのステップが必要である．今後専用のオブジェクトコード・ファイル形式を用意する予定であり，これによってインタプリタからコンパイル済みファイルを直接ロードすることが可能になる．

今後の課題としては，残った開発を進めるとともにホスト上で稼働するという特徴を活かすことができるような拡張機能を付加することである．具体的には，他のシステム，アプリケーション，とくにUCS(Universal Compiling System)系言語とのインタフェースを設計し，実装することが大きな目標である．また，実用アプリケーションのデリバリという見地から，ライブラリを選択的に利用するためのメカニズムが実現できると，実行時のメモリ使用量を削減することができ，ホストシステムへの負荷を小さくすることができると考えている．

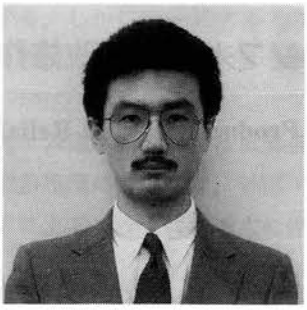
5. おわりに

Lisp専用ワークステーションを用いてクロス開発を行ってきた2200/1100 Lispについて処理系の内容を簡単に紹介した．使用できるライブラリは限定されているものの，Lispプログラムをホスト上で実行するところまでこぎつけることができた．まだ解決しなければならない問題は残っているが，ホスト上でのLisp処理系実装への見通しが得られたと考えている．

参考文献 [1] 大田一久，小林幸一，森澤好臣，“Explorer(KS-301)”，bit別冊“高機能ワークステーション”，共立出版，1987，7，pp. 135～143.
[2] 日本ユニシス(株)，UNISYSシリーズ2200/1100拡張モード(EM)導入準備ガイド，1984，4.
[3] G. Steele Jr., COMMON LISP-The Language, Second Edition, Digital Press, 1989.
[4] D. Touretzky, Common Lisp：A Gentle Intoroduction to Symbolic Computation, The Benjamin/Cummings Publishing Company, Inc., 1989.
[5] W. Hennesey, Common Lisp, McGraw-Hill, 1989.
[6] W. Stark, LISP, Lore, and Logic, Springer-Verlag, 1989.
[7] J. Cohen, “GarbageCollection of Linked Data Structures”, ACM Computing Surveys, Vol. 13, No. 3, 1981, 9, pp. 341～367.
[8] H. Lieberman, C. Hewitt, “A Real-Time Garbage Collector Based on the Lifetimes of Objects”, Communications of the ACM, Vol. 26, No. 6, 1983, 6, pp. 419～429.
[9] G. Steele Jr., “RABBIT：A Compiler for SCHEME”, AI-TR-474, Artificial Intelligence Laboratory, MIT, 1978.
[10] R. Brooks, “Design of An Optimizing, Dynamically Retargetable Compiler for Common Lisp”, Proceedings of the 1989 ACM Conference on LISP AND FUNCTIONAL PROGRAMMING, 1986, pp. 67～85.
[11] R. Brooks, R. Gabriel, G. Steele Jr., “An Optimizing Compiler for Lexically Scoped LISP”, Proceeding of the 1982 ACM Conferecce on LISP AND FUNCTIONAL PROGRAMMING, 1982, pp. 261～275.

執筆者紹介　大　田　一　久 (Kazuhisa Ohta)

1958 年生．81 年慶応義塾大学工学部数理工学科卒業．同年日本ユニシス(株)入社．86 年から知識システム部所属．KS-300 シリーズの日本語機能開発，保守等に従事．現在，開発 2 課で 2200/1100 上の Lisp 処理系の開発に従事．情報処理学会，日本ソフトウェア科学会会員．



ソフトウェア改修作業の生産性と信頼性の実体

Productivity and Reliability in Software Maintenance/Modification

林　雅彦

要　約　システム開発の生産性と信頼性を高めるための報告書は多い．しかしシステム開発後の改修・保守作業の生産性と信頼性について述べたものは非常に少ない．

本稿では稼働しているシステムにおける改修・保守作業について述べる．客先からシステムの開発を受託した場合，稼働後も該システムを改修する作業を委託されるケースが少なくない．この場合にはそのコストの見積りに大きな差が発生し，作業の依頼側と受託側で不一致が発生する可能性がある．

コストの評価には次の二つの考え方がある．

1)　実績型……改修作業に要したパワーを人月またはステップ数で表示し，費したコストを実績ベースで把握する．

2)　SI型……改修作業の範囲を調査・分析し，作業量・難易度・作業環境を考慮して予想のパワーを人月またはステップ数で表示し，作業着手前に依頼者と受託者で確認する．

後者の場合，評価には各種の条件が関係するため，事例の現状を分析し，生産性を明らかにする．

Abstract　There are many reports and papers aimed at improving productivity and reliability in systems development, but very few are seen which depict productivity and reliability involved in the work of modification and maintenance required after systems are developed.

This paper describes how to perform the task of such modification and maintenance on systems in practical operation. Cases are not infrequent where once a computer system is developed on a contract basis, the user continuously requests the same developer to pursue the follow-on maintenance of the system after it gets into operation. In this case, there is a possibility of there being wide difference in the size of required costs, causing a disagreement between the two parties. There are two different approaches to the cost evaluation that can be adopted on this occasion:

1)　Actual result approach —— By representing all the man-power actually required for modification in man-months or in the number of program code lines, all costs are summed up after the task has been completed.

2)　Systems integration (SI) approach —— By representing all the man-power projected in advance with special attention paid to workloads, the level of difficulty and working environment through a close survey and analysis of the work coverage, estimated costs are confirmed between the two before a modification task starts.

This paper discusses how higher productivity is attained by the SI approach by giving a sample case where there is great difficulty in leading to agreement in evaluation because of various interrelated requirements.

1. はじめに

安定したシステムの稼働を維持するためのポイントは，安定性の高いハードウェア

と信頼性の高いソフトウェアおよび効果的な運用に依存する．しかしハードウェアの品質・性能が向上した現在，システムの安定稼働はソフトウェアの信頼性に大きく依存している．

ソフトウェアの信頼性をより高く維持するために，システム運用の関係者は日常各種の対策と改善に努力している．ソフトウェアの中でも業務処理については，外部・内部要因からますます拡大する機能強化や仕様変更の要求によるバックログ増大に伴ない，システムの硬直化を招きつつある．

増大する傾向にあるバックログを，具体的に解決するためには実際の作業レベルでの作業量を正しく把握する必要がある．バックログの正確な見積りにより，全体の作業スケジュールが決まる．バックログがクリアされると新規の開発に着手できる環境ができる．しかし，一般的に作業の依頼者側においても，また作業の支援をする外部のSEにおいても，作業量を正確に見積り，作業スケジュールを作成し，予定通り作業を完了させるプロセスを遂行するためには各種の問題を解決する必要がある．ソフトウェア改修作業の生産性と信頼性が期待通りにならず，作業量の誤差が拡大しないようにするためその要因を把握し，システムが持つ特性を知る必要がある．

そのために実際の作業環境とこれまでの作業を通して経験した結果を整理し，その要因を分析し，これからの作業に役立つ道標を見い出そうと試みた．客先においても，担当するSEにおいても共通のデータから解決案や改善案を追求し，新しい作業環境作りを目標にしている．

2. 作業の対象業務

2.1 業務の概要

事例として作業状況の分析を行うLシステムは大きく3種類の業務から成り，サブシステムとして8種類の業務からなる．各サブシステムは，独立しているが一部には共通ファイルが存在する．各サブシステムが開発され，本番稼働するまでの過程は，各々開発方法や時期が異なり，全体的には必ずしも統一的環境ではない．

昭和56年までに開発したシステムは業務ごとに独自に作成しており，昭和57年以降の作業は比較的共通した作業環境下にある．そのため，初期の開発作業環境に起因した問題に対するシステム運転開始後の変更はむずかしい．

1) 業務Aの概要……業務Bおよび業務Cとはシステムとして独立しており，開発時は環境が整備された状況である．主要業務はリアル処理である．
2) 業務Bの概要……業務Cと共通のハードウェアを使用する環境にあり，主要業務の開発以降も毎年各種の変更作業が発生している．主要業務はリアル処理とバッチ処理である．
3) 業務Cの概要……業務Bと共通のハードウェアを使用する環境にあり，サブシステムが数年ごとに追加されている．主要業務はリアル処理とバッチ処理である．

2.2 各業務の規模概要

各業務のリアル処理時間帯やリアル処理の件数，プログラムの総ステップ数等の概要を表1に示す．

またシステム構成の概要を図1に示す．

表1　プログラム・ステップの概数と処理件数
Table 1　Outline of program steps and process numbers

ケース	業務	リアル処理時間帯	リアル処理件数	プログラムの総ステップ数	処理形態	使用言語
1	業務A	9：00 〜 17：00	約 10〜30件/秒	約130万	リアル処理 ＋ バッチ処理	主に COBOL
2	業務B 業務C	9：00 〜 17：00	約 5〜10件/秒	約170万	リアル処理 ＋ バッチ処理	主に COBOL

上記の数値は処理対象システムを理解する上での概数である．

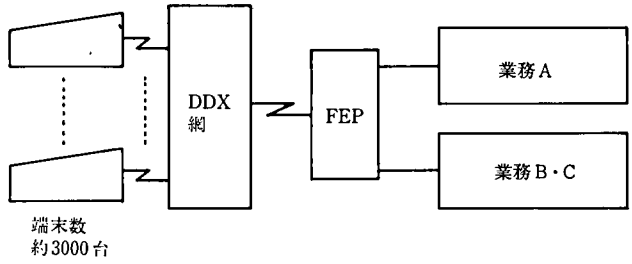


図1　システム構成の要素
Fig. 1　Outline of the system

3. 開発作業と保守作業

3.1 開発作業

システムが本番稼働するまでの開発作業と本番以降に追加で新規に開発する作業がある．客先と受託側の作業分担は，業務により異なる部分はあるが，業務仕様書が確定すると要件の確認作業を関係者で実施する．その後に，開発スケジュールを作成し，各工程表に従い作業が進む．

新規開発作業であっても，類似したシステムが既存システムの中に存在している場合には業務仕様書が確定すれば，全体のスケジュールは把握しやすい．

スケジュール通りに工程ごとの成果物が出て，検証テスト結果も良好であれば開発作業は満足できる内容である．本番稼働後に安定したシステムであり，プログラムが期待される構造になっていれば，その後の仕様変更にも対応しやすく，改修作業を確実に実施することにより長いシステムライフが維持でき，経済性の高いシステムとなる．Lシステムは ADMS(ADMinistrator Support System，プログラム開発管理支援プログラム)で管理しているプログラムの総ステップ数が初期には約200万ステップ強であったが，毎年10〜20万ステップ程度の新規追加があり増大している．

3.2 保守作業

新規開発されたシステムは，カットオーバと共にトラブル修復はもとより，必要に応じて機能追加および仕様変更等改修作業によりシステムの利用価値を維持し，システムを存続させる努力が重要である．

開発当初には各種の環境（開発環境，ドキュメント）が整備されていても，適切な管理運用がなされていなければシステムの機能は時と共に低下する．一度低下したシ

ステムの機能の向上を計る場合には，かなりの余分な作業を伴なう．実際のところコスト面から見ても再構築をするか，細々と稼働させるのみという場合が多い．

経験的にも保守・改修作業は開発作業と深く連動し，機能的に結合している生き物である．改修作業で一度手を抜くと，その後の向上を期待できないと考える．開発時に作成したドキュメントが改修時に更新されなかったり，ドキュメントの一部が紛失すると，その後の改修作業は生産性でも品質の面でも低下してしまう．

3.3 保守作業の種類

ソフトウェア保守の改修作業は次の3種類に分類される[1]．

1) 修理保守(corrective maintenance)……プログラムの異常終了への対処等，いわゆるバグの修正作業である．処理上の誤りを修正するもので，設計内容と機能仕様との不具合等への対処である．
2) 適合保守(adaptive maintenance)……OSのレベルアップ等，処理環境の変更に適合させるために修正する作業である．
3) 完全化保守(perfective maintenance)……処理効率の向上や出力情報の追加等の機能向上を目指すために修正する作業である．

これらの3種類の保守作業の内で，最も時間的制約のきびしい作業は修理保守である．難易度の高いものであると，高度の業務知識と技術力を必要とする．これに比較して，適合保守や完全化保守はスケジュールを作成し，時間的余裕が多少ある．

ここで保守作業全体の生産性や信頼性を向上させるための環境作りのポイントは，要員育成計画作成時に，業務経験の少ない者に対して，計画的に適合保守や完全化保守作業を通じての目的を持った作業実施をさせることである．

3.4 開発時とその後の保守開発体制

システムの開発とその後の運用時の支援体制についての概要は，図2の支援体制の

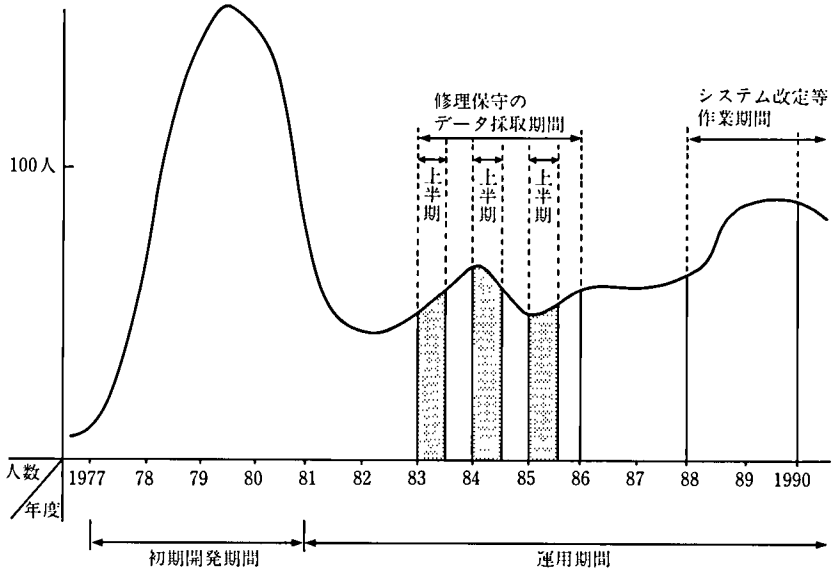


図 2　支援体制の概要

Fig. 2　Outline of system support organization

概要に示す通りである．この図には客先の開発要員は含まない．初期のシステム開発時には，客先側で開発業務に精通したキーマンや支援体制もそれなりに充実している．客先では人事ローテーションが数年程度の経験者に対して行われる．数回繰り返すと，定常的な運用については手順書に従い運用可能となる．しかし，新規や追加機能の業務仕様書の作成やシステム化の検討においてパワーダウンは免れない．

従来は，作業依頼者側で作成できた上流工程の作業が目標に達していないということが発生する可能性がある．結果的に従来の依頼者側と受託者側の作業工程の仕切りが変化してくる．そうすると受託者側の投入パワーやコストが同一作業にも係わらず拡大してくる．図2の要員数は各年間でピークがあるが，平均化している．

3.5 支援体制

初期のシステム開発時には各業務の担当ごとにSE・プログラマを割り当てている．その組織は図3(a)に示す通りである．また，システムの運用期間に入り，特定業務のピークが過ぎてからの組織は図3(b)に示す通りである．

後者の組織では保守開発グループの若手のプログラマには集中的にプログラミングの技法を教育し，かつ特定業務に片寄らないプール要員とした．共通の要員として確保することでプログラマ間の作業量を平均化し，広く一般的な情報に接するよう試み，要員の技術向上とモラルの維持に努めている．

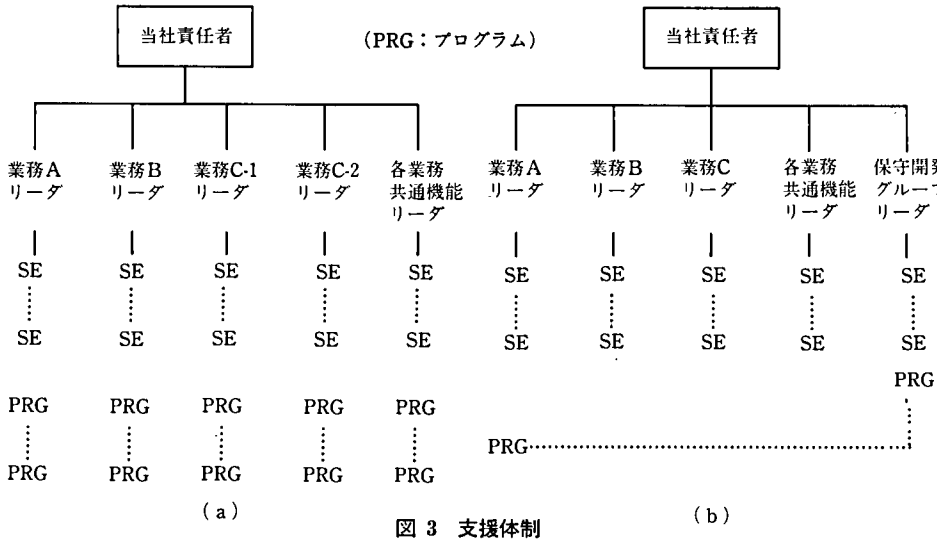


図 3 支援体制

Fig. 3 Support organization

3.6 作業工程と担当範囲

作業工程の各称や区分方法について，業務により一部異なるものがある．従来使用してきた名称を用いて作業範囲を示すと図4の通りである．

図4に示す通り，システム開発当初には客先側で実施されていた工程が，保守・運用期間に入ると変化し，結果として受託者側の作業範囲が拡大している．この傾向は各業務により異なるが基本的には同様である．従来と比較して，保守開発作業を実施すれば生産性は低下しコストは上昇する．実作業の把握が必要である．

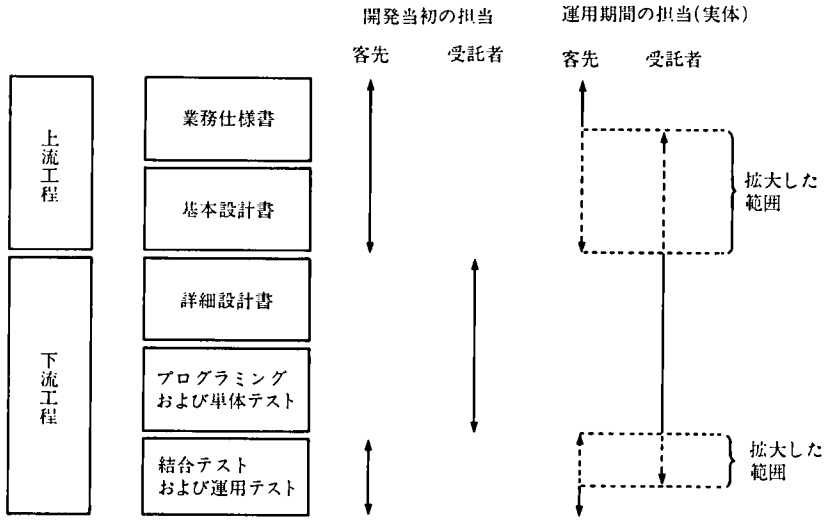


図 4　開発工程の受け持ち範囲

Fig. 4　Support area of the system development

3.7　保守作業の手順

保守作業の手順は開発作業の手順に準じている．作業項目や作業分担および作業ごとの成果物は，表 2 に示す通りである．

表 2　ソフトウェア保守作業手順

Table 2　Step of software maintenance

ステップ	作業項目(概要)	作業分担	推進責任	主な成果物
1	改修案件について事前の相談	U，N		・打ち合わせメモ
2	改修要求仕様書の作成	U	U	・改修要求仕様書 ・作業依頼書
3	要求仕様のすり合わせ・確認	U，N		・改修要求仕様書 (確定版)
4	改修作業計画の立案・確認	U，N		・作業計画書
5	改修作業の実施			
	1) 改修基本設計，テスト設計	N		・基本設計書(改修版)
	2) 設計内容のレビュー・確認	U，N		・詳細設計書(改修版)
	3) プログラムの修正/作成	N	N	・修正コード ・単体テストケース一覧 ・結合テストケース一覧
	4) 機能テストの実施と確認	N U		・テスト結果
6	サンプルJCL等の作成，操作手順書	N		・サンプル JCL ・操作手順書
7	検収テスト	U	U	・テスト結果
8	本番運用開始の準備	U		・運用手順書
9	SGおよび公開	U		
10	納品物の整理および納品		N	・納品物一式

U：客先，N：日本ユニシスを示す．

4. 生産性と信頼性

4.1 生　産　性

ソフトウェアの開発時に，SEやプログラマが人月当たりどのくらいの成果物を作成するかステップ数で表現する場合がある．作業環境や作業条件により大きな差が生じる．開発対象システムで使用する言語，難易度，開発方法，期待される成果物等により生産性が変化する．ある報告書[2]によると生産性の要因として29項目も挙げている例があるが，その影響度を把握して生産性を決定するには実際のところ難点がある．

しかし，一般的にプログラミング（プログラム設計・コーディング・単体テスト）でCOBOLの場合には1,300ステップ/人月程度の基準値[3]を設定している．当社内の報告でもRSDM(Reliable System/Software Development Method)開発工程の考え方からプログラミング（上記と同程度の作業）の基準値を1,300ステップ/人月と設定している．

Lシステムの初年度新規開発作業のプログラミングの生産性は次の通りである．

業務A　700ステップ/人月
業務B　950ステップ/人月
業務C　500ステップ/人月

この生産性の実績は，他の年度にもほぼ適用できる値である．各業務を平均すると，約800ステップ/人月となる．これは新規プログラミングの1,300ステップ/人月の約60％程度である．この値は客先指定の詳細設計書（HIPO*）等の成果物作成作業が影響していると考えられる．上流工程の作業が関連してくると，数値的には低下するが，これは従来の作業とは別項目の作業であり，プログラミングに組み入れるのは適切ではない．

4.2 信　頼　性

ソフトウェアの信頼性はその品質により決まってくる．品質を確保するためには，開発当初の正確な設計からテストまでの作業と，その後の保守作業が期待に添うものである必要がある．一般的に何回も変更を加えていると修理保守が発生し，保守の手間がかかる．

常に品質の向上（低下させない）を目標にすることで，生産性の向上と信頼性の維持が保てる．予定の作業期間内に作業が終了しても，その品質を確認しないと価値の判断はできない．経験的にソフトウェアの信頼性が高い場合には，生産性も比例して高い傾向にある．作業工程ごとに品質が確認できている作業には大きな心配はない．何らかの理由で工程ごとの品質が十分に確認できていない場合には信頼性に欠け，本来の機能を満たすためにコスト高になるケースがある．

4.3 保守作業に対する評価

ソフトウェアの保守には，修理保守・適合保守・完全化保守があることは前述した．いずれの保守も新規開発作業との違いは，すでに対象物が存在することである．また品質や作業環境も多種多様である．そのために改修作業の大きさに関係なく，作業を行う前に該当ソフトウェアを調査・分析し本来の改修に入らなければならない．しか

＊ HIPO：1970年頃IBM社により開発されたプログラム図式の一つである．

し客先側から見ると，最終的に保守作業の対象となる結果だけが関心の的となりやすい．一度でも関係する調査・分析を実施すると，全体のアプローチについて見方が判るものである．保守対象が特定できても改修作業後の確認テストが場合によっては，広範囲のテストになることからコスト高になる．

保守作業に関する作業量・作業範囲を客先と受託側で確認することが重要である．保守作業が常に適切な対象になり得る環境作りと認識作りが期待されている．

4.4 修理保守作業の生産性

3か年の修理保守作業について，毎年の上半期に発生したものを集計した．この修理保守作業の生産性は図5に示す通りである．各年度上半期の修理保守の発生状況は表3の通りである．ただし，初年度は4月～1月の10か月間を対象にしている．

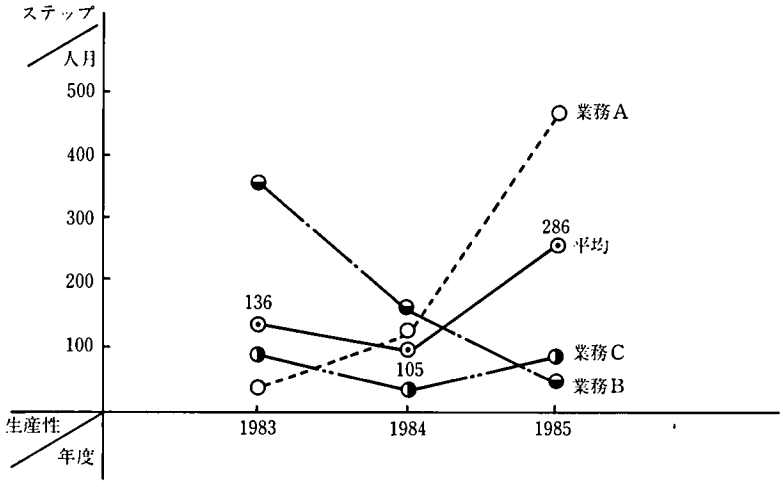


図 5　修理保守の生産性

Fig. 5　Productivity of corrective maintenance

表3　修理保守の発生状況

Table 3　Steps of corrective maintenance

項目 / 年度	件数	改修ステップ数	パワー(人月)	生産性 (ステップ数/人月)
1983	33	1,372	10.1	135.8
1984	19	400	3.8	105.3
1985	12	2,035	7.1	286.6
合計	64	3,807	21.0	(平均) 181.3

4.5 年度別生産性

1984年度と1985年度の上半期の修理保守，仕様変更（適合保守と完全化保守），新規開発ごとの生産性を表4，表5に示す．作業量（人月）は，修理保守と仕様変更・新規開発の二つのグループに対してしかデータを集計できなかった．1984年度の仕様変更と新規開発の生産性が高いのは類似処理が多かったためである．作業担当の内工要員，外工要員は両年度とも同一メンバと考えてよい．

1986年度以降の修理保守作業のデータは整理していない．仕様変更の作業で改修対

表4　1984年度上半期の業務別作業の生産性

Table 4　Productivity of maintenance (Half of 1984)

業務＼項目	作業の種類	件数	改修 新規のステップ数	パワー 人月	左の項目の 内工分(%)	生産性 (ステップ数/人月)
A	修理保守	7	62	0.96	0.96(100)	64
	仕様変更	27	10,079	39.5	12.2(31)	1,949*
	新規開発	9	66,895*			
B	修理保守	9	66	1.2	1.2(100)	55
	仕様変更	4	3,258	90.5	20.9(24)	1,328**
	新規開発	3	116,895**			
C	修理保守	3	272	2.2	2.2(100)	124
	仕様変更	19	3,268	47.7	26.2(55)	713
	新規開発	12	30,725			
合　計	修理保守	19	400	3.8	3.8(100)	1,105
	仕様変更	50	16,605	247.4	59.3(24)	934
	新規開発	24	214,515			

* 新規開発ステップのうち，約80%は類似処理のため生産性が高い．

** 新規開発ステップのうち，約10万ステップが一つのサブシステムである．

表5　1985年度上半期の業務別作業の生産性

Table 5　Productivity of maintenance (Half of 1985)

業務＼項目	作業の種類	件数	改修 新規のステップ数	パワー 人月	左の項目の 内工分(%)	生産性 (ステップ数/人月)
A	修理保守	7	128	2.0	0.5(25)	64
	仕様変更	13	6,483	18.4	2.4(13)	987
	新規開発	20	11,679			
B	修理保守	2	128	1.5	1.5(100)	85
	仕様変更	6	4,033	37.4	11.5(31)	810
	新規開発	8	26,270			
C	修理保守	3	1,779	3.6	0.6(17)	494*
	仕様変更	7	1,426	64.3	22.7(35)	895
	新規開発	6	56,162			
合　計	修理保守	12	2,035	7.1	2.6(37)	286
	仕様変更	26	11,942	120.1	36.6(30)	883
	新規開発	34	94,111			

* 生産性が高いのは類似処理のためである．

象のプログラムの総ステップ数が1万ステップを越える場合は，比較的高い生産性を示している．しかし，システムの初期開発に参加した者が主戦力の時に高い生産性を示している．運用期間に入って参加した者は，作業環境を理解し，業務知識の向上の期間を要するため生産性は低下する．

ところで，1984年度は運用期間に入って3年目であるが，修理保守の担当は内工100%であり，初期開発への参加者が主力である．1985年度の修理保守の担当は内工37%で，外注要員で対応できる環境が整備されている．

4.6 開発・保守作業の生産性の事例

大型システムの新規開発を受託した場合，本番稼働後も引き続き保守作業を担当することが少なくない．客先との間で，作業基準値を設定している事例もある．新規作業に比べて保守作業の生産性は，一般に低く客先の環境によりその値は異なる．分類項目として，次の項目等が考えられる．

- 処理形態　リアル，バッチ
- 作業形態　新規，保守
- 難易度/規模 大，中，小ごとの生産性（ステップ/人月）

4.1節で述べているプログラミング工程の生産性を1,300ステップ/人月と比較すると，保守作業は経験的に新規作業の2倍～5倍程度のコストを要する．

4.7 生産性と作業量の把握

保守作業の生産性変動要因は多種多様であるが，単純な作業量の見積りとして次の式が考えられる．

$$(A/B+C\times D/E)\times F/G \qquad (4\text{-}1)$$

A：予想改修ステップ数

B：期待できる生産性（平均200ステップ/人月）

C：作業範囲の係数（0.01～1.00，1＝全範囲の見直し）

D：修正対象プログラムの総ステップ数

E：生産係数（プログラム設計レベル　3,000）

F：作業担当者の資質の係数（0.5～2.0，1＝経験5年・情報処理1種程度）

G：難易度の係数（0.7～1.5，平均値＝1）

上記の式(4-1)を1986年度に完了した作業に適用すると，実作業に近い数値となった．この式(4-1)を適用するポイントは，各種の係数を作業環境と経験から見つけることである．係数は経験の要素が大きく，変動する値である．

4.8 ソフトウェア保守についての事例[4]

ウェスティングハウス社のSoftware and Information Servicesの部門長であるRon Clanton氏は，ソフトウェア保守の諸問題や該社におけるソフトウェア保守について，次のように整理をしている．

1) 米国におけるソフトウェア保守に関するシンポジウム(1987年)で議論された項目は次の通りである．
   ① 保守要員の確保と適用業務知識の保持
   ② 技術的な挑戦と解決策
   ③ 生産性を低下させない対策
   ④ 古くなったシステムに現在の技術をどう活かすか
2) ソフトウェア保守に関する用語の定義について
   ① 保守（Maintenance）
      ・機能的なパフォーマンスの状態に戻すこと
   ② 拡張（Enhancement）
      ・ユーザの満足度を現存の機能でもっと増大すること
   ③・開発（Development）

・新しい機能的な仕様
・新しい設計
・新しいコード
・新しいドキュメント

3) 古いコードにはいくつかの特徴がある.

① 多くの人々が関与ている.
② 文書化が一部しか表現してない.
③ 品質について一貫性がない.
④ 故障は処理の停止につながる.
⑤ OS の入れ替えにより古いコードが動かない場合がある.
⑥ 修正コードの量が多く，コードの管理がしにくい.

4) 保守と拡張……コードライフ（プログラム・コードの寿命）に対する拡張の役割は図 6 に示す通りである.

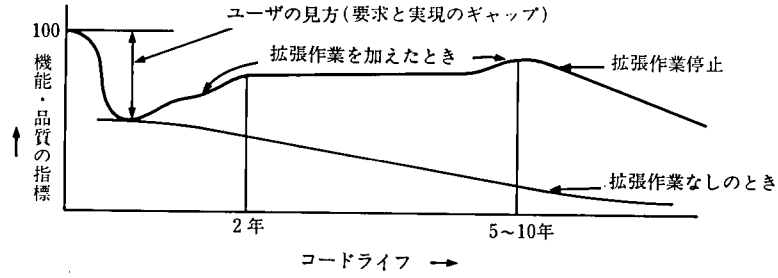


図 6 システム拡張の効果

Fig. 6 Effective enhancement

5) CASE (Computer Aided Software Engineering) ……ソフトウェア・エンジニアリングはプログラミングだけでなく，ソフトウェアのライフサイクルを支援する有効な手段である. Engineering Database を共有しながら CASE を支える各種のツールを図 7 に示す.

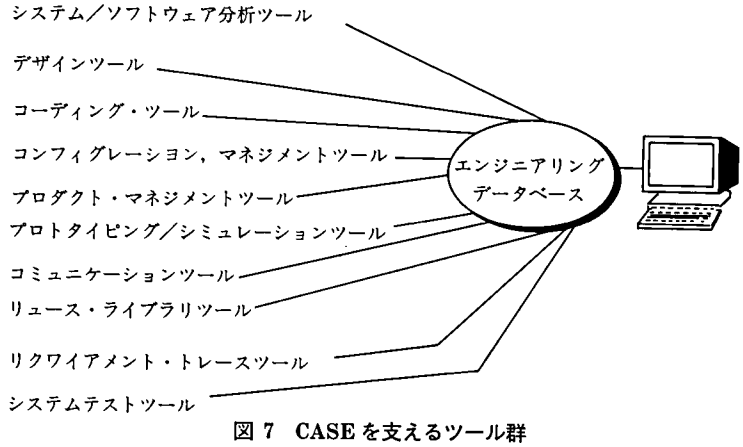


図 7 CASE を支えるツール群

Fig. 7 Support tools for CASE

6) メンテナンスの担当者についての課題……誰が保守をするのか. 良い環境や刺激のある環境を作らないと若いプログラマが離れていく可能性がある. 常に新し

い技術に挑戦し，技術的に成長するテクノロジのマネジメントが必要である．

7)　品質と生産性向上のためのアプローチのポイント

①　既存のコンポーネントの再利用

②　アプリケーションは作るより買う．

③　生産性向上策

④　手戻りを減らす（プロトタイピングの活用）．

⑤　エラーの早期発見（ツールの利用，レビューの実施，段階的テストの実施）．

⑥　プロダクトの変更がやりやすくなる（複雑なモジュールは作らない）．

ソフトウェア保守についてのこれらの記述は，ほとんどわれわれの作業についても適用できる．要員の問題，作業環境の問題，システムライフの問題や品質と生産性向上のためのアプローチについての方法についても同様である．

5. 評価と今後の課題

5.1 評　　価

新規開発作業の生産性と保守作業の生産性の比較を行った場合，同一の客先では，以下のような項目がわかる．

1)　新規開発作業の生産性は平均して800ステップ/人月である．とくに類似処理のため生産性が高いものを除くと，700～900ステップ/人月の範囲にある．

2)　修理保守の生産性は，データ採取期間の3か年の各上半期の平均は181ステップ/人月である．修理保守の工数分布は，55～49ステップ/人月となり約9倍の差がある．

3)　新規作業と修理保守作業の生産性はステップ数の比較で4：1になる．ステップ数のみで見ると，修理保守作業は新規作業の4倍のコストを必要としている．修理保守の工数分布から比較すると，1.6倍から14.5倍のコストに分布する．

信頼性について，今回は修正コードの公開後のトラブル件数や問題点の発生状況を詳細に把握できなかった．経験的に見ると生産性の高い作業が，信頼性も高い傾向にある．信頼性については今後の作業で集計してみる必要がある．項目として，仕様の確定度・担当者の技術力・担当者の経験度合や作業対象の総ステップ数・改修ステップ数およびトラブル件数等である．

生産性や信頼性を述べる時に，単に数値的な比較だけではむずかしい面がある．しかし，システムの特徴や作業条件の異なる個々のシステムの中で関係者により，両者の向上を計る努力が求められる．

最終的には，品質とコストおよび納期を予定した範囲に制御できることである．

5.2 今 後 の 課 題

システムが初期開発から10年以上経過してくると，今後の課題を把握して対処していく必要がある．

1)　保守要員……開発に携わった者もローテーションで入れ替わる．システムの全体を見渡せる力を持つ者を常に育成するとその効果は大きい．

2)　改修作業と新規作業の選択……保守性の良いプログラムは構造化された判りやすいものである．保守性の悪いプログラムは10％以下の改修量であっても，新規

作成の方が生産性や品質の面でも満足が得られるものだ．判断基準を作成するとよい．

3) 保守作業の管理……現状の保守作業を把握し，各種の開発技法の試みを重ね，管理可能な環境作りをすることが必要である．

4) 支援ツールの活用……システムライフが長期になるとプログラムと関連するドキュメントの間で不一致が発生してくる．詳細設計書のレベルのものはツールで内容を更新できるように検討すると効果が期待できる．また，テスト環境はシステム特有の制限があるので，運用期間に適したツールを作成する必要がある．

5) 管理責任者の確認……システム資産を守るには適切な管理者が必要である．適切なシステム監査等を通して，実際に実施可能な改善点の支援が重要である．

6) 保守作業を考慮したシステム開発……システムライフと運用期間の作業環境や要員構成を考慮して，そのシステムに適合した保守用ドキュメントを初期のシステム開発時に作成する．また，保守用の情報を整備しておくことにより，運用期間の生産性や信頼性の向上を計ることができる．

6. おわりに

システムが構築されて10年近くになると，システムの再構築を含めて各種のテーマが話題となる．常にシステムが活動的であるために対応策の策定と実施が大切である．

とくにソフトウェアは，システムの初期開発時の善し悪しがシステムライフに大きな影響を与えていることを実感している．

ご指導をいただいた客先の皆様と社内の先輩や同僚の方に心から謝意を表したい．

参考文献 ［1］ 宮本勲，「ソフトウェア保守の管理」，TBS出版会，1984.
［2］ 国友義久，「プログラミングの生産性と工数見積り」，BIT，1981 1～3月号.
［3］ 特集「システム開発の生産性向上を目指して」，SYSTEMS，ユニシス研究会，1985 5･6月号，pp.2～34.
［4］ 日本ユニシス，「戦略的情報システムの構築」，第15回ユニシスATS 海外セミナー渡米研修用報告書，1988.

執筆者紹介 林 雅彦(Masahiko Hayashi)
昭和21年生．44年青山学院大学理工学部機械工学科卒業．同年日本ユニシス(株)入社．当初シリーズ1100のハードウェア保守業務を担当．その後社会公共関連のシステム開発，SEサービスに従事．現在日本ユニシス・ソフトウェア(株)社会公共統括部に所属．

形式的記述技法への誘い
—— 記述言語はなぜ必要か ——

染　谷　　誠

1. 記述言語以前

情報処理分野の技術状況には根本的な弱点がある．この分野に携わって20年近くなりますが，ずっとそう思ってきました．世間でもそう言われてきました．理由はいろいろあると思いますが，私がとくにそう感じたのは，ソフトウェア開発を支える記述法が欠けているという点です．どんなソフトウェアを開発するのか，それを記述するための枠組みや記述言語がないということです．ソフトウェアを定義するための仕様記述言語はどういうものか．この課題こそ情報処理分野の技術的諸問題の中でも最も基本的な問題のはずです．

それで仕様記述言語に関連した研究にはとくに注目してきました．仕様記述言語といっても，それこそピンからキリまで汗牛充棟ただならぬものがありますが，私達が注目してきたのは仕様記述言語Z[1，2]とかVDLから進化した仕様記述言語VDM-SL[3]，OBJ等の代数的仕様記述言語，それに同期・並行処理問題を記述する枠組みとしてのCSP[4]やCCS[5]位です．

このような仕様記述言語を使って事務処理分野の課題の仕様を実際に書いてみました．その結果このような記述言語が実際の開発に十分使えるとの確信を得ましたので，機会あるごとに，こうした形式的仕様記述言語を使って開発してみませんかと薦めてみましたが，残念ながら今のところ誰も採用してくれません．採用してもらうためには，当然のことですが，やはりそのための環境を整える必要がありそうです．

2. 仕様記述言語より発想法か?

形式的ないしは準形式的仕様記述言語を実際の開発に使用するよう薦めているうちに，何度か同じような意見を耳にしました．それはこういうものです．

「今，開発現場でほしいのは仕様記述言語ではなく，いわば発想法なのだ．発想，つまりアイデアを得るために思考を推進すること，それを支援してくれるツールが欲しいのだ．難しいのはアイデアを思い付くことであって，それさえ思い付けば，その仕様を書くこと等，いとも簡単な仕事なのだ．」

「仕様記述言語より先ず発想法」というわけです．この立場，以下「発想法優先説」と呼ぶことにしますが，それを聞いた瞬間「おかしい」と思いました．が，その時はうまく反論できませんでした．今もできそうもありません．ですから，できることならわざわざ面倒な反論等せず，無視してすませたい．が，形式的仕様，あるいはその便法としての半形的仕様の普及を目論でいる立場上そうもいきません．といった事情もありますし，それにやはり気になります．何しろ上のように言われたことを，今でもありありと思い出せる位ですから．と言うわけで，以下「発想法優先説」に対する反論を展開してみましょう．

3. 「発想法優先説」の言語観

聞いた瞬間「おかしい」と感じた最大の理由は，言語観についての彼我の相違にありそうです．思考と言語の関係，思考を推進する上で言語の果たす役割についての考え方がずいぶん違うようです．

「発想法優先説」つまり，まず必要なのは発想法で，それによってアイデアさえ思い付けばその仕様を書くのは簡単な仕事だ．こういった考え方の背後にある言語観はどういうものでしょうか．もちろんはっきりしたことはわかりませんが，おそらく次のようなものではないかと思われます．

「思考は，この場合アイデアを思い付く作業，言語とは切り離して遂行できるものである．その結果得られたアイデアを記述するのは，ちょうど目の前にある風景をキャンバスや画用紙の上に描く時のように，頭の中（あるいは心の中）にあるアイデアを言語で写し取ればよいのだ．」

ちょうど風景が，たとえば富士山が，それを描いた風景画とはまったく独立に存在するように，考えとかアイデアも言語とは無関係に心の中（あるいは頭の中）にあり，それを記述する作業も，目の前にある富士山を見た通り画用紙の上に描くのと同じように，心の中（あるいは頭の中）にある考えを言葉で表現すればよい，と言うわけです．

このような言語観の問題点は思考や考えが言語表現とは独立に頭の中で結実しているという点です．これが第一．それと，その頭の中に結実している考えを「言葉で写し取る」作業をいとも簡単な仕事ととらえられていること．これが第二です．

もっとも，これらは二つの問題というより，一つの問題なのかもしれません．と言うのは，第二の問題は第一の問題の当然の帰結かも知れないからです．「頭の中に結実している」をどうとらえるかですが，その解釈の仕方によっては，「結実」の仕方がそもそも「言葉で写し取り」やすい仕方で「結実している」のかも知れないからです．そうなると第二の問題は第一の問題に吸収されてしまいます．そこで「結実の仕方」をどう解釈すればよいのか，果たして第一の問題は第二の問題を吸収してしまうのか，が問題になりますが，当然，それを判断する権利は私にはない．で，以下では第一の問題だけを採り上げることにします．

さて，この「思考や考えが言語表現とは独立に頭の中で結実している」と類似の言語観は古くからあるようです．また，それと対立する言語観もあるらしく，近年はこちらの方が優勢らしい．が，そうした事情はここで一切無視することにします．それと，言語観等と言い出すと「ウィトゲンシュタインがどうの・・・」，「フレーゲはこうの・・・」と言わないと格好がつかないようですが，そうした諸大家の説も一切無視することにします．理由は簡単で，こちらにそうした諸家の説を勘案しながら議論を進めるだけの学がないからです．つまり「無視する」のではなく「無視せざるを得ない」だけのことです．それに，仮りに無理してそんなことをしても，どうせ古来からのアポリアにぶち当たって，解決の困難さを思い知らされるところが落ちでしょう．きっとそうなります．ここではわれわれの常識に従って議論を進めたい．

4. 「頭の中で結実」説への違和感

「思考や考えが言語表現とは独立に頭の中で結実している」という考え方の正否について議論するつもりはありません．そんなことは到底不可能です．しかし，この「頭の中で結実」説には何か違和感ないしは反発を感ずる．で，その原因について考えてみることにします．

違和感を感ずる第一の理由は，いくら良い考えが「頭の中で結実」していると言われても，それだけでは誰の役にも立たないことです．良い考えは適切な言語表現を得て初めて良い考えとして「われわれ」に共有できるものであって，たとえどれほどみごとに「頭の中で結実」していても，それだけでは何にもなりません．ある特定の人の考えは，適切な言語表現を通じて初めて公共のものとなり，他の人の共感や批判が受けられるようになるわけです．つまり，考えは「頭の中で結実」するのではなく，「適切な言語表現」を得たときに結実するのではないでしょうか．

第二の理由は，良い考えを公共のものたらしめるために「適切な言語表現を得る」のが大そうむずかしいことです．この点は，どうやら私ひとりの問題だけでなく，「われわれ」に共通する問題らしい．自分では「頭の中で結実」していると思っても，いざ書いてみようとするとなかなか言葉にならない．苦労してやっと思い付いた表現も，実際に書いてみると，ほとんどの場合「適切な言語表現」等ではない．「頭の中で結実」していると思ったものが実に頼りないものであることを思い知らされます．

まだあります．確かに，やっと思い付いた表現は「適切な表現」でないことが多いわけですが，だからそんな苦労はまったく無駄かと言うと，決してそんなことはありません．なるほど，やっと思い付いた表現は「適切な表現」ではないかもしれませんが，それでもそれは「適切な表現」を得るための第一歩なのです．適切ではなくても，ともかく考えを言葉で表現してみることが大切で，そうすることによって「頭の中で結実」していると思ったものが実は不完全であることもわかり，そして何よりも，その不完全な表現を「推敲」することによって「適切な表現」に近づけることができる．この「推敲」が大切です．「推敲」と言うと何となく文学的なイメージが強いかもしれませんが，ここではもっと広く，「文科系」とか「理科系」に関係なく，言語表現をより適切な表現に直す作業の意味で使っています．そして，この「推敲」こそ「われわれ」の思考活動の主要部分ではないのか．そう言いたいのです．つまり，思考や考えは決して「頭の中で結実」するものではなく，むしろ「言語表現として結実」するものなのだ．再びそう言いたいのです．これが「頭の中で結実」説に違和感を感じる第三の理由です．ただ，お断りしておきますが，思考や考えに「内省」的な面がまったくないというつもりはありません．言語表現をより適切な表現に変換する作業が思考活動の主要部分である，そう言っているだけです．

5. 常識的言語観

「頭の中で結実」説に対する違和感をまとめると以上のようになりますが，結局のところ極めて常識的な言語観を開陳しているに過ぎないようです．

いつ誰から聞いたのか，あるいは何かの本で読んだのか，はっきりしたことはわかりま

せんが，ともかく「人は言葉によって考える」とか「思考とは言語使用能力を行使することの一形態である」といった考え方をいつのまにか知っていた．しかもそれを当然のことと思い込んでいた．それで以上のような違和感を感じたのかも知れません．

そうだとすると，以上「発想法優先説」，というより「頭の中で結実」説，を批判してきたつもりですが，結局のところ，この「頭の中で結実」するという古典的な説と常識的言語観を対比させたに過ぎないのでしょうか．ひょっとするとそう誤解されかねませんので，いそいで結論めいたものをまとめておきます．

6. 結　　論

もって回った言い方をしましたが，私が言いたかったのは「仕様記述言語」こそ「われわれ」の最も基本的な課題だ，これに尽きます．

強力な「仕様記述言語」であって初めて作るべきソフトウェアが何なのかが適切に記述できる．この「適切に記述する」ことも一般にはむずかしいことですが，不完全な記述を「推敲」することによって，つまり記述言語の助けを借りて行うことができます．仕様記述言語こそ最強の発想法のツールなのです．

そうしてひとたび作るべきソフトウェアが何なのかが言語表現に結実すれば，つまり仕様が記述されれば，「作るべきソフトウェアが何であるか」が公共の問題となり，それについて客観的な議論ができるようになります．客観的に議論すべき問題は二つあります．一つは，その仕様が本当に要求を満たすのかであり，もう一つは，その仕様を満たすソフトウェアをどう作るのか，つまり実現の問題です．こうした問題が客観的に議論できるわけです．

また，その当然の帰結として，ソフトウェア開発技術が本当の意味で教育可能なものになる．この点も注目してよいことでしょう．が，理屈をこねるのはもうこの位にしましょう．

参考文献 [1] J. M. Spivey, “The Z Notation-A Reference Manual”, Prentice Hall, 1989.
[2] J. M. Spivey, “Understanding Z”, Cambridge University Press, 1988.
[3] C. B. Jones, “Systematic Software Development using VDM”, Prenice Hall, 1990.
[4] C. A. R. Hoare “Communicating Sequential Process”, Prentice Hall, 1985.
[5] R. Milner, “Communication and Concurrency”, Prentice Hall, 1989.

（システム技術本部 生産技術部）

ビジネスUNIXシステムU6000/60

本間康雄
Y. Homma

UNISYS U6000/60は，U6000シリーズ（以下U6000）シングルプロセッサ・システムの最上位機種として，U6000初のインテル80486（32ビット・マイクロプロセッサ）を搭載したシステムとして誕生した．

ハードウェア性能は下位機種のU6000/55（クロック33.3 MHzのインテル80386マイクロプロセッサ使用）の約1.5倍で，ソフトウェアは他のU6000システムとオブジェクトレベルで互換性がある．

1. U6000のプロダクトライン

U6000はU6000/60が加わり，現在6システム，10モデルから構成される．小型から大型までの幅広い能力を提供し，システム規模や業務に応じた機種選択を可能にするばかりでなく，ソフトウェアの互換性を保証し，レベルアップ等においてもソフトウェア資産は継承される（表1）．

U6000は大きく，二つのシステムタイプに分けることができる．一つはシングルプロセッサ・システムであり，もう一つはマルチプロセッサ・システムである．シングルプロセッサ・システムはその名の通りプロセッサは1個，マルチプロセッサ・システムは2個から10個（U6000/70）または，20個（U6000/80）まで必要に応じて増設が可能となっている．U6000/60はこの中で，シングルプロセッサ・システムの最上位機であり，小型高

表1 U6000シリーズのラインアップ

システムタイプ	システム	サポートユーザ数（最大）	プロセッサ数	プロセッサ	
				種類	クロック(MHz)
シングルプロセッサ・システム	U6000/31	16	1	i80386	20
	U6000/51	32	1	i80386	25
	U6000/55	64	1	i80386	33.3
	U6000/60	**80**	**1**	**i80486**	**25** ←
マルチプロセッサ・システム	U6000/70	200	2～10	i80386	20
	U6000/80	400	2～20	i80386	20

表2 U6000のハードウェア機能

		U6000/51	U6000/55	U6000/60
MPU	プロセッサ	i80386		i80486
	クロック	25MHz	33.3MHz	25MHz
	キャッシュメモリ	64KB	64KB	64KB
メモリ		4MB～40MB	4MB～40MB	4MB～40MB
浮動小数点プロセッサ		i80387（オプション） WEITEK3167(オプション)	i80387（オプション） WEITEK3167(オプション)	内蔵（i80387相当） WEITEK4167(オプション)
カートリッジテープ（QIC）		150MB		
フロッピディスク		5¼インチ（1.2MB）		
内蔵ディスク（フォーマット）		340MB～1.98GB	340MB～1.98GB	340MB～1.98GB
磁気ディスク装置(フォーマット)		340MB～1.98GB	340MB～1.98GB	340MB～1.98GB
システム・プリンタ		1台（セントロ）＋n台（RS232）		
磁気テープ（PE/GCR）		1台		
サポートユーザ数		32	64	80
筐体サイズ	高さ	67.4cm	67.4cm	67.4cm
	幅	45.7cm	45.7cm	45.7cm
	奥行き	30.5cm	30.5cm	30.5cm
重量		45kg	45kg	45kg

表3 U6000/60基本構成

構成機器	モデルC仕様	モデルD仕様
CPU	i80486 (25 MHz)	
キャッシュメモリ	64KB	
メインメモリ	4MB	
ディスク・テープコントローラ	オンボード・フロッピ & SCSIコントローラ	
ディスク	380MB (アンフォーマット) 340MB (フォーマット)	760MB (アンフォーマット) 660MB (フォーマット)
フロッピディスク	5.25インチ 1.2MB	
カートリッジテープ	150MB	
パラレルプリンタ・ポート	セントロニクス・インタフェース×1	
通信ポート	SYNC/ASYNCポート×2	
	ASYNCポート×1 (コンソール用)	
通信コントローラ	ASYNCポート×8	

性能のUNIX*プラットフォームとして使用される (表2).

2. U 6000/60 のハードウェア

U 6000/60 は, インテル80486 (25 MHz) マイクロプロセッサをCPUに採用している. 80486は, 80387相当の浮動小数点プロセッサを内蔵しているが, オプションとしてWEITEK 4167浮動小数点プロセッサを選択することができる. サポートユーザ数は最大80であり, メインメモリは基本4メガバイトで4メガバイト単位に最大40メガバイトまで拡張できる. システムメモリとして使用できる領域は4メガバイトから32メガバイトまでで, 残り4メガバイトから8メガバイトまではRAMディスクメモリとして使用する. キャッシュメモリは64キロバイトが標準装備されている.

U 6000/60 はSCSI方式のディスク・テープコントローラを内蔵している. ディスク容量はモデルにより異なり, モデルCは340メガバイト (フォーマット), モデルDは660メガバイト(フォーマット) を1台標準装備している. 基本キャビネットには, 内蔵ディスクとして2台から3台 (ディスクサイズの組み合わせで異なる) の増設が可能で, 拡張磁気ディスク装置を接続すると最大6台/3.96ギガバイト (フォーマット) まで拡張が可能となる. ただし, 1/2インチ磁気テープ装置を接続する場合はディスク数は最大5台となる.

カートリッジテープは1/4インチタイプで, 容量は最大150メガバイト, フロッピ・ディスクはIBM社のPC/AT互換5.25インチが標準装備されている.

オプションスロットは8個用意されており, 各種通信コントローラを支援する. コントローラは通信コントローラ, ユニシス・ホスト通信コントローラ, イーサネット**コントローラ, X.25/RS 232コントローラ, SYNC通信コントローラが用意されている (表3, 図1).

その他オプションとして, 1/2インチ磁気テープ装置 (1600/6250 BPI) が1台, 独立筐体の拡張磁気ディスク装置が1台接続可能である.

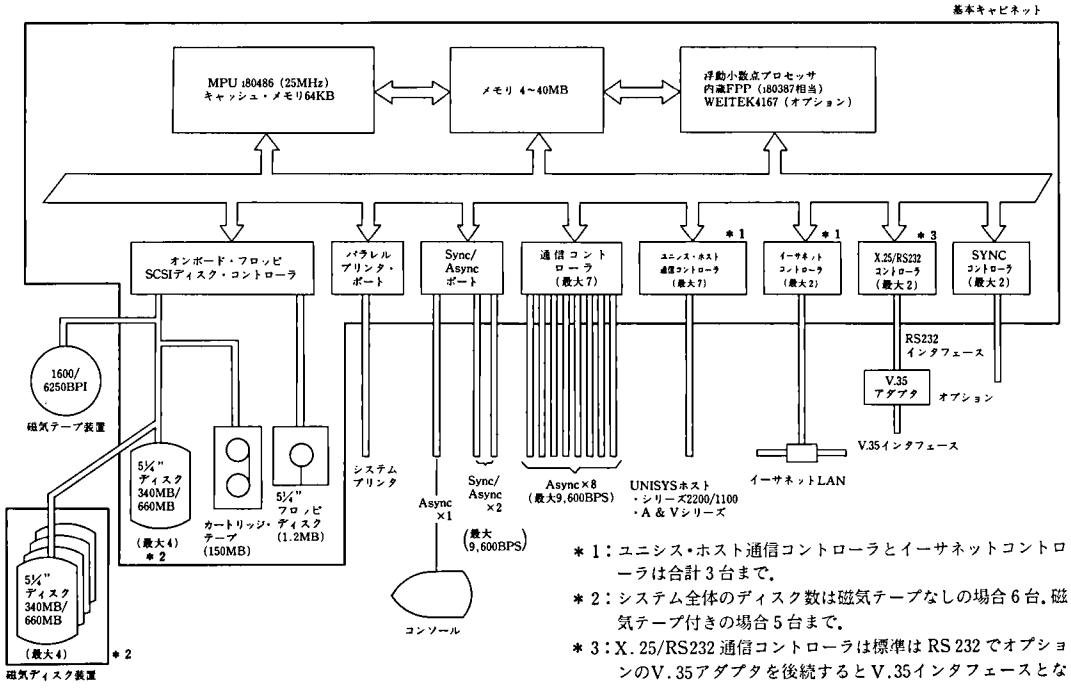
通信コントローラは8回線ASYNC (非同期) ポートを提供し, 回線速度は最大9600 bpsで, 標準装備の1台を含み合計7台まで増設できる.

ユニシス・ホスト通信コントローラは, シリーズ2200/1100, A & Vシリーズと最大19.2 Kbpsで通信が可能(ただしシリーズ2200/1100, A & Vシリーズ各々1台ずつ装備可) である. ソフトウェアとしてシリーズ2200/1100との接続用にユニスコープ・エミュレーションが, A&Vシリーズ用にETエミュレーションが必要である.

イーサネットコントローラは, イーサネットLANに接続するためのコントローラで, 通信速度は10 Mbpsであり, 最大2台まで接続が可能である. ソフトウェアとしてNET-6000が必要である. (なおユニシス・ホスト通信コントローラとイーサネットコントローラは合計3台まで装備できる).

* UNIX オペレーティング・システム：UNIX System Laboratories, Inc.が開発ライセンスしている.

** イーサネット (Ethernet)：米国Xerox社の登録商標である.

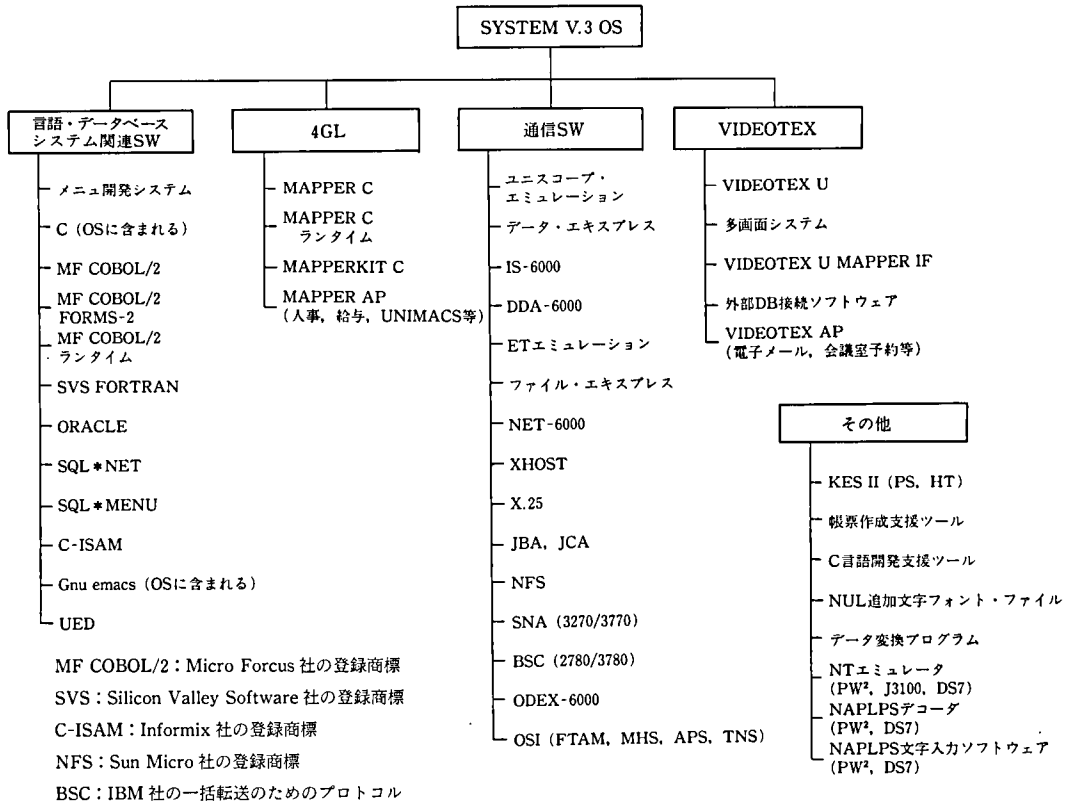


＊1：ユニシス・ホスト通信コントローラとイーサネットコントローラは合計3台まで.

＊2：システム全体のディスク数は磁気テープなしの場合6台.磁気テープ付きの場合5台まで.

＊3：X.25/RS232通信コントローラは標準はRS232でオプションのV.35アダプタを後続するとV.35インタフェースとなります.

図1　U 6000/60 システム構成



MF COBOL/2：Micro Forcus 社の登録商標

SVS：Silicon Valley Software 社の登録商標

C-ISAM：Informix 社の登録商標

NFS：Sun Micro 社の登録商標

BSC：IBM 社の一括転送のためのプロトコル

図2　U 6000 のソフトウェア本系

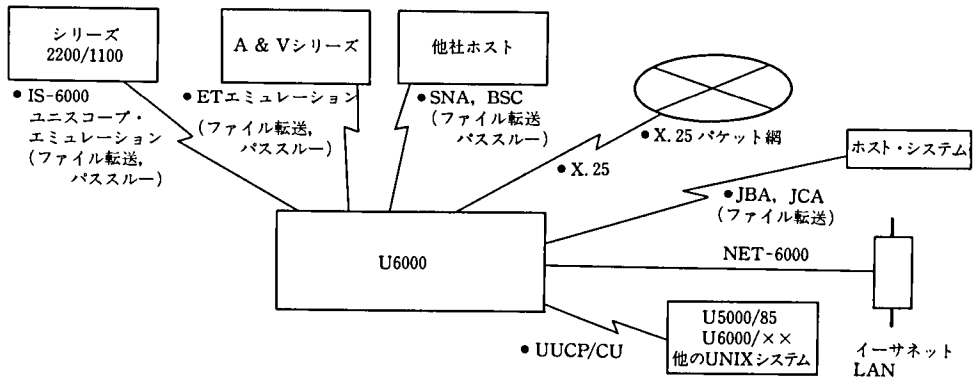


図3 U 6000の通信ソフトウェア

X.25/RS 232 コントローラはX.25パケット網と最大19.2 Kbpsで，またV.35アダプタを接続すると48 Kbpsまでの通信が可能となる．V.35アダプタは最大2台まで接続でき，ソフトウェアとしてX.25が必要である．

SYNC通信コントローラは，シリーズ2200/1100および他のU 5000/6000とULDCで接続するか，またはSNA*手順で接続する場合に使用するコントローラである．ソフトウェアとしてIS-6000(最大19.2 Kbps)，SNAソフトウェア(最大64 Kbps)が必要で最大2台まで接続可能である(図1)．

3. U 6000/60のソフトウェア

U 6000/60のオペレーティング・システムは，SYSTEM Vリリース3.2に対しBSD版の機能強化，さらに米国ユニシス社独自の機能拡張，当社の日本語機能付加により，日本語UNIXシステムとして充実した機能を提供している．

4 GL（第4世代言語）としてMAPPER Cが提供されており，会話機能を使用した初歩的な利用から，ラン機能を使用した複雑な業務システム開発まで，エンドユーザによる開発実行が可能である．

さらに，RDBMSとして世界的評価の高いORACLE**が用意されており，SQLの全機能を提供している．

VIDEOTEX Uは画像情報を利用者に提供するシステムで，利用者にわかりやすいビジュアル・コンピューティングの世界を構築することができる（図2）．

通信ソフトウェアとしては，図3に示すソフトウェアが用意されている．

その他のソフトウェアとして，①言語関連ではC言語，MF COBOL/2，SVS FORTRAN，②エキスパート・システム構築用ツールのKES II，③ウィンドゥ環境を提供するXHOST，分散ファイルシステムを支援するNFS等，ビジネスアプリケーションで必要とされる各種ソフトウェアが用意されている（図2）．

また，U 6000には第三者ベンダが開発・移植した各種ソフトウェアとしてUTSライブラリ(Unisys Third party Products)が100種類以上用意されており，U 6000/60もこれらのライブラリを利用することが可能となっている．

4. おわりに

U 6000/60はU 6000の最新モデルとして，幅広いユーザニーズを満足させるパワフルなコンピュータ処理能力を提供する．

オープンシステムへの展開が急速に進み，OS，アプリケーションインタフェース，ユーザインタフェース等標準化の進展，コンピュータテクノロジの飛躍的発展等，このような変化に対応して，ユニシスはUNISYS & UNIXをモットーにオープンシステムに積極的に取り組んでいる．

---

* SNA：IBM社のネットワークアーキテクチャである．

** ORACLE：米国ORACLE社の登録商標である．

Michael Jackson 著
大野侑郎・山崎利治監訳

システム開発 JSD 法
共立出版，A5 判 xix＋527 pp.，
1989，12,360 円

JSD は Jackson System Development の略であり，プログラム設計法として日本でもよく知られている JSP (Jackson Structured Programming) を発展させたものである．一部に，JSP はプログラム設計を対象としており，JSD はシステム設計を扱うものであるかのように誤解している向きがあるが，そうではない．確かに JSD は，時間に依存するシステムの開発全般を対象としているため，JSP よりも広い範囲に適用される．著者の言葉を借りていえば，JSD は JSP と同一の原理をより広い範囲の問題や，開発作業のより広い部分に適用するようにした JSP の拡張であるとみなすことができる．

JSP の基本原理は，計算の対象を入出力データ列の構造およびそれらに基づいたプログラム構造として記述する．プログラムの詳細機能はその後で記述する．つまり，最初にモデルを作り，後から機能を追加する．JSD ではこの原理はより鮮明になる．計算対象は，システムの外部にあって時間的にはっきりした実世界であり，まずこれをモデル化し，詳細機能はシステムからの出力を考察する時にモデルに追加する．

JSP は完全な仕様から出発した．JSD は与えられた仕様から出発するわけではない．JSD の開発はシステムの仕様を作り出すことから始まる．伝統的には，まずシステム分析，仕様の作成およびシステム設計があって，つぎにプログラム設計とコードの作成が続くと考えられている．この視点によれば，JSD を用いてプログラムの仕様をつくり，それを受けて JSP でプログラムを設計すると考えるかもしれない．これは必ずしも正しくない．

JSP のモデル化は，一つの（構造不一致がある場合は複数の）逐次プロセスの構造を作ることであった．JSD のモデルは，複数の逐次プロセスからなり，その相互通信システムとして実現される．この逐次プロセス作成の原理は JSP と同一である．このことが示すように，JSP のいろいろな側面が JSD の開発手順の中にばらまかれているのである．この意味で，JSD は JSP の前置なのではなくて，JSP の原理を拡張したものとみなすことができる．

本論は 3 部からなっている．

第 I 部	JSD 入門	80 ページ
第II部	JSD の開発段階	333 ページ
第III部	各種の話題	50 ページ

第 I 部は JSD 開発法の紹介である．JSD の基本原理を日常用語で解説し，小さな課題で JSD 法の開発手順を説明する．著者は，JSD の大要を理解したいだけの読者は第 I 部を読むだけで十分であると言っているが，第 I 部だけ読んで JSD の基本原理を理解できた，と言える人は相当できる人である．平易な日常語で書かれているからといって，JSD を理解することは決してやさしくない．その理由は後で述べる．

第 I 部の章立ては以下のようになっている．

第 1 章　モデルと機能
第 2 章　プロセス・モデル
第 3 章　仕様の実現
第 4 章　JSD の開発手順

第II部が本書の主要部分であり，JSD を本当に理解するためにはここをじっくり読まなければならない．三つの事例を開発段階ごとに交互に採り上げ，適切な場所で必要な技術解説を行っている．三つの事例とは，倉庫の部品管理，エレベータ制御，新聞社の懸賞応募問題であり，技術上の問題点が事例ごとに散らばる仕組みになっている．JSD の六つの開発段階の説明になるので，第II部の章立てを記そう．

第 5 章　三つの事例
第 6 章　実体行動段階
第 7 章　実体構造段階
第 8 章　初期モデル段階
第 9 章　機能段階
第 10 章　システムタイミング段階
第 11 章　実現段階

伝統的開発法に毒された開発者が JSD を理解することは相当困難な作業だろう．記述が数学的記法を用いるとか，難解な用語を使うとかいうの

ではない．アプローチが伝統的方法とあまりにも違うからである．伝統的な方法では，仕様化から実現へ進む経路がはっきりと見えない．システム設計者は対象となる課題を口頭による説明的な仕様を与えられ，最終的にシステムの流れ図，ファイルの定義，プログラムの仕様等を作成する．仕様書を眺めると，システムの概要説明に続いて突然ファイルの定義書が出てくることもまれではない．どの時点で何を決定したか明らかではない．この仕様書を書いた設計者は，おそらく何を構築すべきかを決定する前にシステムの構築作業を行っているのである．

JSDでは最初に機能を考えるのではなく，対象世界のモデル作りから出発する．伝統的開発者にとってこのモデルを作ることが最初の難関になるだろう．多くの伝統的開発者にとって機能なしのモデル化はまったく異質な作業であるからである．M.ジャクソンはこうしたアプローチが機能中心の開発法に比していかに安定した，保守しやすいシステムを構築するか，功みな比喩を用いて説明する．第Ⅰ部のこの解説はジャクソンの真骨頂を示している．

JSDの開発手順の最初の2段階で，「実体」，「行動」の用語で対象世界の抽象記述をする．部品管理システムでいえば，実体は顧客や部品である．実体は行動の時系列とみなす．顧客は，発注する，変更する，納入する等の行動の列で構成され，JSP流の木構造で表現する．開発手順の第3段階で，行動の列として表現された実体を実体プロセスとし，プロセスとプロセスの間を通信文でつなぐ．でき上ったプロセス間通信体系はモデル全体であり，いわば実世界を模擬したものといえる．伝統的開発者にとっては，ここが第2の難関となるだろう．実世界をプロセス間通信で模擬することに多くの伝統的開発者はなじんでいないだろうからである．離散型シミュレーションやAdaのタスクの使用経験者にはとっつきやすいかもしれない．

プロセス体系図に機能を加え，機能付プロセスとしたのが仕様である．さらに必要に応じて時間的配慮を加えて仕様は完成である．伝統的開発者にとっては奇妙な仕様であるに違いない．完成した仕様は実世界を模擬したものであるから，実体プロセスの数は実体の数だけ存在し，しかも増減する．顧客プロセスは顧客の数だけ存在し，部品プロセスは部品の種類だけ存在する．したがって，プロセスごとに割り当てるだけのcpuが存在し，それらが故障なしに永遠に動くとすれば，最終的な仕様はまったく非効率的なシステムであるけれども，理論上はそのまま動く仕様である．実際にはこうはいかないから，一つのcpu上で動くように仕様を変換しなければならない．これが開発手順最後の実現段階での仕事である．ここが伝統的開発者にとって最後の難関であろう．仕様が正しければ正しい実現を導くような，変換のためのいくつかのアイデアが準備されている．その解説は本書にお任せしよう．

本書を読むにはJSPの知識は前提としないことになっている．実現段階ではJSPの，プロセスの分割やプログラム転換の技法を必須としている．世の中には構造不一致問題を扱わないでJSPを実践しています，という人が多いが，ここではそうはいかない．

第Ⅲ部では，誤りデータと入力副システムでの扱い，それにシステムの保守とJSDの管理的側面について述べている．

伝統的開発者にとって，本書がむずかしいことを少し強調し過ぎたかもしれない．それは，日常語で書いてあるものはやさしく，形式的方法のアプローチはむずかしい，という認識が蔓延しているように思えるので多少過敏に反応したせいかもしれない．一見やさしそうに見えるけれども，実はやさしくない原因は，背景にしっかりした形式的発想を持っているが，それを平易な日常語で語る，というジャクソン流に由来するのだろう．所詮，ソフトウェア開発法はやさしくない．学習時にはやさしい開発法，たとえばシステムを大きな機能と把え，序々に詳細化しようとするトップダウン開発法は実践の場で行きづまってしまうようである．

学習がむずかしくても，いったん習得してしまえば目から鱗が落ちること必定である．

本書が出版されたのは1983年である．ジャクソン自身はその後あまりJSDには関与していないようで，弟子達が教育・実践の場を経て方法の洗練をはかっている．実体と行動の切り出し方，開発手順分け等にその違いが見られるが，本質的な違いがあるわけではない．新しいJSDに興味のある方は参考文献を参照頂きたい．

参考文献

[1] JSP & JSD, THE JACKSON AP-

PROACH TO SOFTWARE DEVELOPMENT Second Edition, John Cameron, IEEE Computer Society Press, 1989.
[2] JACKSON SYSTEM DEVELOPMENT, Alistair Sutcliffe, Prentice Hall, 1988.

（研究開発部　峰尾欽二）

山崎利治 著

計算機科学/ソフトウェア技術講座 3
プログラムの設計
共立出版，A 5判，212 pp.
1990年1月，2,680円

本書はプログラムの設計，とくに事務計算プログラムの設計に有効な理論と形式的方法を紹介する．形式言語，表示的意味記述，CSP，抽象データ型，VDM等である．いわば形式的方法の中辞典になっている．ソフトウェア工学を勉強する人には，形式的方法の案内になるであろう．

ところで，多くの人は形式的方法に不慣れであり，それゆえに半ば否定的である．また形式的方法の提唱者たちと否定者たちの主張には深いギャップがあり，何が事実なのかわからなくなっている．そこで本書の紹介の後に，形式的方法の七つの神話と事実も紹介することにした．これはIEEE Softwareの形式的方法特集号で，英国の実務家が発表したものである．

高級プログラム言語といっても，命令型の言語で自由に書かれたプログラムは複雑でわかりにくい．検証には多くの時間を費す．プログラムの仕様や設計を，一定の方法と枠組みで形式的に記述し，これを検証する方がもっと効率的である．また，形式的な記述を命令型プログラムに書き換えるのは比較的やさしい．

ソフトウェア工学が，開発の初期，つまり仕様や設計が問題になる時期に，もっと人と時間を投入すれば全体の工期と経費を低減できると主張するのは，このことを言っているのであろう．ソフトウェア工学には，形式的方法の導入が不可欠である．

以下，本書の構成にそって内容を紹介する．本書の性格上，理論と方法は基本を示すにとどまり，例題も小さい．参考文献等でさらに勉強し，広く応用できるようになりたい．

1章は，典型的な事務計算の課題とそのCOBOLプログラムを提示する．プログラムはループと場合い分けで構成されている．この構成の根拠は4章で明らかになる．ジャクソンのプログラム設計の原理に慣れている人は，自然に読めるプログラムである．

2章はデータ型の復習をする．事務計算は種々のレコードの組み合わせに対する情報処理であることが多い．レコードは直積集合の要素，ファイルはレコードの列と形式化される．

3章は，仕様・設計・命令型プログラムの関連を論ずる．仕様は問題を定式化して提示したもので，問題に対して疑問の余地のない議論ができなければならない．そのため，どうしても形式的な記述になる．

設計は仕様とプログラムの間の具象化・詳細化の過程である．同じ課題に対して，設計は幾通りもあり，作られるプログラムもいろいろになる．このとき，全体の構成要素と道筋を明確に示すのが設計である．

仕様イコール設計の時もある．プログラム言語の構文がある条件を満たすとき，機械的に構文から構文解析プログラムを作る方法がある．すなわち構文はプログラム言語の仕様であり，またコンパイラの設計でもある．

4章は，LL(k)文法と構文解析および表示的意味記述を説明する．これらは，コンパイラの設計法の研究とプログラム言語の仕様を形式的に記述する研究から生まれた．この理論に基づく方法を，事務計算プログラムの仕様記述と設計に応用するのが本章の主題である．

事務計算における種々の入力データの組み合わせは形式言語の文法で記述でき，プログラム言語の構文に対応付けられる．入力に対する計算と出力は，構文の意味に相当し表示的意味記述で表すことができる．表示的意味論の枠組みをそのまま事務計算に当てはめることができるのである．

さらに，コンパイラの設計法を当てはめれば，文法からプログラムの構造を決めることができ，基本設計にもなる．課題のデータ構造は正規文法で記述できるので，ループ構造のプログラムにし，もっと複雑なLL(1)文法の記述になるものは，再帰呼び出しを使うプログラムにする．

本章の理論と方法はPascalやCの実用コンパイラで実証済みである．事務計算に限らず広く応

用できるので，基礎知識として身につけておきたい．

5章は，同じ事務計算の課題をCSP（交信逐次プロセス）の考え方で仕様記述し，Adaでプロトタイプしたものを提示する．CSPは環境と相互作用を持つシステムの仕様・設計・実現のための基礎理論である．

仕様は課題の意味するところを直観的に余すところなく表している．どうしたらこのような仕様を書けるようになるだろうか．この仕様からプログラムを導くにはどうすればよいだろうか．この二つを方法論にまとめたのがジャクソン・システム開発法JSDといってよいであろう．

JSDは本書では採り上げていないが，JSDのすべてを説明する本が出版されている．その図書紹介が本号にあるので詳しくはそちらを読んでいただきたい．

6章はプログラム設計の一般論である．プログラムの正しさに対する考察から，go to文の追放，段階的洗練，プログラムの検証等構造的プログラミングの重要な主題が提唱され，これらは抽象データ型という枠組みに集約する．

抽象データ型，とくに代数的仕様記述は抽象的である．高度に汎用の仕様を記述するには，それだけ抽象的な記述法が必要であろう．実際，ISO（国際標準化機構）がOSI（Open Systems Interconnection）の通信プロトコルを記述するために制定したLOTOSは，この方法でデータ型の仕様を記述する．

一般プログラム向きの抽象データ型は，記述対象のモデルを想定し代数型ほど抽象的ではない．それでも論理型や関数型の仕様記述は，操作型に較べて抽象的といえるだろう．だが開発システムの全体を検討し理解するのに抽象化以外の方法があるだろうか．数百ページにわたる具体例は人間の理解の範囲外であろう．

ウィーン開発技法VDMは，論理型の仕様記述法を採り入れた形式的プログラム作成法である．形式的な仕様作成を行い，段階的洗練の各ステップで設計の検証をする．近年，欧州を中心に産業界で利用が盛んになり，標準化も検討されている．VDMの使用経験，道具の開発，教育・規格等を目的としたシンポジウムも頻繁に開かれており，勉強しておきたい方法である．

VDMと同じ考え方で，ファイル処理の事務計算プログラムの形式的仕様を書き，検証支援系を用意したのがVFPL-VSである．これは三菱総合研究所において，玉井哲雄，福永光一によって作成された．

以下は，静岡大学教育学部・白井靖人先生の紹介記事（情報処理，1990年12月 Vol. 31 No. 12, P. 1692）からの引用である．

形式的手法にまつわる七つの神話

1) 形式的手法は，ソフトウェアが完璧であることを保証してくれる．
2) そもそも形式的手法とは，プログラムの証明に関することである．
3) 形式的手法とは，安全性が重視されるシステムにおいてのみ有用である．
4) 形式的手法を使うには，高度な訓練を受けた数学者が必要となる．
5) 形式的手法は開発経費を増大させる．
6) 形式的手法は顧客にとって受け入れ難いものである．
7) 形式的手法は，現実的な大規模ソフトウェアには適用されていない．

事実は次のようである．これには，CASEソフトウェアの開発に仕様記述言語Zを使った事例が付いている．

1) 形式的手法は，開発初期において誤りを発見するのに大いに役立つ．また，ある種の誤りに関してはそれをほとんど除去することが可能である．
2) 形式的手法とは，開発者自身が，自分の開発しようとしているシステムについて詳しく検討することによって機能するものである．
3) 形式的手法は，ほとんどどの応用分野でも有効である．
4) 形式的手法は数学的仕様に基づいているが，これはプログラムよりずっとわりやすいものである．
5) 形式的手法は開発経費を低減させる．
6) 形式的手法は，顧客が何を購入しようとしているのかを理解する手助けになる．
7) 形式的手法は，現場での現実のプロジェクトに適用され成功をおさめている．

（研究開発部　宗像清治）

掲載論文梗概

直接操作型インタフェース記述システムは，OMS (Object Manipulating System：視覚的対話システムを記述するための方法論の確立を目指すもの）の基本システムとして，その直接操作の対象をCLOS (Common Lisp Object System) のオブジェクトに限定したOMS/Bの概要について，例題を交じえて紹介している．

Lispは記号処理向きの言語として，数式処理，エキスパートシステム等の分野で使用されてきている．近年，Common Lispとして言語仕様の標準化が進められており，ワークステーションを中心に処理系も徐々に市場に出てきている．大田一久は，**UNISYSシリーズ2200/1100のCommon Lisp処理系**の中で，Lispについて述べるとともに，開発中の2200/1100 Lispに関して，処理系の内容について紹介している．

システム開発の生産性と信頼性を高めるための報告書は多いが，システム開発後の改修・保守作業の生産性と信頼性について述べたものは非常に少ない．林雅彦の**ソフトウェア改修作業の生産性と信頼性の実体**は，客先からシステム開発を受託し，稼働後も該システムの改修・保守を委託されているケースについて，生産性と信頼性を中心に紹介を行っている．

★

掲載論文梗概

ISSN 0914-9996

技　報
UNISYS TECHNOLOGY REVIEW
Vol. 10　No. 4　(No. 28)

発行日　平成3年2月28日
編集人　柳生孝昭
発行人　富田和夫
発行所　日本ユニシス株式会社
東京都港区赤坂2-17-51 〒107
TEL(03)3585-4111（大代表）
印刷所　三美印刷株式会社

UNISYS

またひとつ、核になる力。

ユニシスのシリーズ2200ラインアップがさらに充実。
90年代の基幹情報システムをになう機能をフル装備して、いまデビューします。

ユニシスは、つぎつぎと応えます。時代に、ビジネスに、ユーザーに――。90年代の基幹情報システムに求められる機能を追求し、提案し続けるユニシスは、超大型汎用機2200/600Ⅱを発表。そしていままた、刻々と変化するビジネス環境に的確に対応できる柔軟なシステムを目指して、2200/400Ⅱシリーズを発表。2200/400Ⅱシリーズは、いままでの2200/400シリーズを、使いやすさはそのままに機能をいっそう強化。さらに従来、超大型機でのみサポートされていた拡張トランザクション処理アーキテクチャ「XTPA*」を搭載するなど、処理能力の飛躍的な向上を実現しました。2200/400Ⅱシリーズの誕生により、シリーズ2200のラインアップはさらに充実。先進の開発力とテクノロジーで90年代ビジネスのあらゆる規模のあらゆるニーズに、しなやかにお応えするユニシスです。

●中央処理装置搭載数1～6の6モデル同時発売。●同一キャビネット内で最大5倍のパフォーマンス・アップが可能。●XTPAにより最大4システム(24IP)まで接続、高速レコード・ロック制御機能によってあらゆる規模のトランザクション処理が可能。●プロセッサ、メモリー、入出力機構へ最新のVLSI技術を全面採用。●高密度実装技術の採用により、消費電力・発生熱量を削減、省スペースも実現。●ユニシスの第四世代言語「MAPPER」と「LINC」をさらに強化、アプリケーション開発の効率化を促進。

＊XTPA：Extended Transaction Processing Architecture.

大型汎用コンピュータ

UNISYS 2200/400Ⅱシリーズ誕生。

日本ユニシス株式会社　本社 東京都港区赤坂2-17-51 〒107 電話03-3585-4111(大代表)