OUKA
The University of Osaka
Institutional Knowledge Archive

| | |
|---|---|
| Title | ソフトウエア開発過程の関数的記述と開発支援システムの作成に関する研究 |
| Author(s) | 荻原, 剛志 |
| Citation | 大阪大学, 1990, 博士論文 |
| Version Type | VoR |
| URL | https://hdl.handle.net/11094/1242 |
| rights | |
| Note | |

*The University of Osaka Institutional Knowledge Archive : OUKA*

https://ir.library.osaka-u.ac.jp/

The University of Osaka

# ソフトウェア開発過程の関数的記述と開発支援システムの作成に関する研究

１９９０年 １月

荻原剛志

# 内容梗概

本論文は，筆者が大阪大学大学院基礎工学研究科（物理系情報工学分野専攻）に在学中，鳥居研究室において行った，ソフトウェア開発過程に関する研究のうち，開発過程記述言語の設計とその処理系の作成，開発過程記述の作成手法，および開発過程記述の実行に基づくソフトウェア開発支援系に関する研究をまとめたものである．

第１章では，ソフトウェア開発過程の記述およびその実行に関する主要な研究について述べ，この分野の研究の目的と必要性を明らかにする．

第２章では，ソフトウェア開発過程を記述するために段階的詳細化の方法が有効であることを述べ，ソフトウェア開発過程を記述するための動作的記述モデルを提案する．

次に，開発過程を記述するための言語の要件について論じる．開発過程を記述するための言語は，同一の形式的な枠組みの中でさまざまな抽象度のレベルの記述ができなければならない．また，開発作業を支援するために備えているべき機能について述べる．

第３章では，開発過程記述のために設計した関数型言語ＰＤＬ(Process Description Language) およびその実行処理系であるＰＤＬインタプリタについて述べる．

ＰＤＬは代数的言語の意味定義に基づく関数型言語である．同一の形式的意味定義の枠内でさまざまな抽象度の記述を行うことができ，段階的詳細化による開発作業記述の開発に適している．また，ツール起動，ウィンドウ操作など，ソフトウェア開発を行うために必要な組込関数を備えている．

ＰＤＬによる開発過程記述は，ＰＤＬインタプリタ上で実行する

ことができる．ＰＤＬインタプリタは記述に従って自動的にツールを起動したり，ウィンドウを操作したりできる．また，実行中に検出された未定義関数に値や定義を与えて実行を継続できる機能など，開発過程の記述を支援するためのさまざまな機能を備えている．

第４章では，段階的詳細化によって開発過程記述を作成する方法について論じる．ここで提案する方法では，まず，開発過程を開発過程の流れと生成物の流れの２つに分けて抽象度の高い記述を行い，これらを段階的に詳細化して結合し，最終的に実行可能な開発過程記述を得るものである．この詳細化の方法は大変簡潔かつ直接的であるため，さまざまな開発過程の記述に対して適用可能である．

さらに，開発過程記述に基づくソフトウェア開発環境（ＳＤＥ：Software Development Environment）について述べ，さまざまな用途や開発者に応じたＳＤＥを生成できる，適合可能なＳＤＥについて論じる．

第５章では，ソフトウェア開発技法のひとつであるＪＳＤ(Jackson System Development)を例にとり，段階的詳細化によってソフトウェア開発過程の記述を得る手順について論じる．この手順ではまず，自然語によってＪＳＤの各開発過程の入出力生成物の関係や性質を記述する．次にこの記述を開発支援システムの仕様とみなし，ＰＤＬによる形式的な記述に変換した後，段階的に詳細化することによって実行可能なＰＤＬの記述を得る．

この記述を実行すると，各作業に適したツールが自動的に起動され，さまざまな条件判定を容易に行うことができる．矛盾があって先に進めないときには適切な開発過程へ後戻りする．このシステムによって，ＪＳＤに従った開発を容易に行なうことができる．

第６章では，本研究のまとめを行ない，さらに，今後の問題点について論じる．

# ソフトウェア開発過程の関数的記述と開発支援システムの作成に関する研究

## 目　次


関連発表論文 …… 1

第１章　序論 …… 2

第２章　開発過程の記述 …… 5

　第１節　序言 …… 5

　第２節　関連する研究 …… 6

　第３節　開発過程記述モデル …… 10

　第４節　開発過程記述言語 …… 12

　第５節　結言 …… 14

第３章　開発過程記述言語の設計とその実行処理系の作成 …… 15

　第１節　序言 …… 15

　第２節　ソフトウェア開発過程記述言語 …… 16

　第３節　開発過程記述言語ＰＤＬ …… 18

　第４節　ＰＤＬインタプリタの機能 …… 27

　第５節　結言 …… 41

第４章　段階的詳細化に基づく開発過程の記述方法 …… 42
第１節　序言 …… 42
第２節　スクリプトの詳細化 …… 44
第３節　適合可能なソフトウェア開発環境 …… 58
第４節　結言 …… 64

第５章　開発過程の記述による支援システムの作成 …… 65
第１節　序言 …… 65
第２節　ＪＳＤの性質記述 …… 67
第３節　ＰＤＬによる支援システムの作成と実行 …… 71
第４節　検討 …… 83
第５節　結言 …… 86

第６章　結論 …… 87

謝辞 …… 91

文献 …… 92

# 関連発表論文

[1] Katsuro Inoue, Takeshi Ogihara, Tohru Kikuno, and Koji Torii: "A Formal Adaptation Method for Process Descriptions", Proceedings of the 11th International Conference on Software Engineering, pp.145-153 (May 1989).

[2] 荻原剛志, 井上克郎, 鳥居宏次: "ソフトウェア開発を支援するツール起動自動制御システム", 電子情報通信学会論文誌(D-I), Vol.J72-D-I, No.10, pp.742-749 (1989年10月).

[3] 稲田良造, 荻原剛志, 井上克郎, 鳥居宏次: "ソフトウェア開発過程の形式化とその詳細化による支援システムの作成 －ＪＳＤを例として－", 電子情報通信学会論文誌(D-I), Vol.J72-D-I, No.12, pp.874-882 (1989年12月).

# 第1章　序論

ソフトウェア開発は，例えば，仕様の要求分析から始まり，設計書の作成，ソースプログラムの作成，コンパイル，デバッグ，テストと続く，作業の系列を実行してゆくことと見ることができる．このような開発作業の系列をソフトウェア開発過程，あるいは単に開発過程(process)と呼ぶ．ひとつの開発過程は，さらにいくつかの開発過程の組み合せからなると捉えることができる．開発過程は開発者が直接実行したり，テキストエディタやコンパイラなどのユーティリティプログラム（これらをツールと呼ぶ）を利用して行う．各開発過程で参照，変更，作成される対象を生成物(Product)という．例えば，仕様書や，設計書，ソースコードなどは生成物である．

ソフトウェア開発過程全体の構成方法，各段階で行なうべき作業，生成物の記法などは，作るべきソフトウェアの種類や規模，開発者の数や質によって大きく異なる．既にソフトウェアを正しく作成できた開発過程に従って，別のソフトウェアを開発できるとは限らない．また，人によっては一つの開発過程として捉える範囲や行なう作業，生成物の記法などが異なるであろう．さらに，携わる人数も異なり，一人による開発から，少人数のチーム，上限がないような大規模のチームでの開発がある．チームによる開発では，開発者の情報交換や管理までを開発過程の一部と考えるであろうし，新たな生成物（例えば電子メイルなど）も必要になるであろう．

従来のソフトウェア開発では，作業の進め方についてのおおまかな考え方や指針は与えられていても，実際に開発を行う際に，それをどのように解釈し，具体的にどのように作業を進めてゆくかは個々の開発者によって異なる場合が多かった．

このため開発者は，要求仕様の理解やモジュール設計といった，ソフトウェア開発にとって本質的と考えられる作業以外に，自分自身で以下のような点を考えに入れながら作業を進めなければならな

かった.

(1) 開発作業の内容と実行順序の決定, および作業進行の管理
(2) 適切なツールの選択とそれらの起動
(3) 作業の結果として作成する生成物の管理

開発者がこれらを適切に行うには, 作業内容や種々のツール, それを起動するコマンド等に関する多くの知識が必要であり, 特に経験の少ない開発者にとっては大きな負担となる. さらに, 開発する生成物の数, 量, 種類の増大に伴い, それらのデータ構造や作成, 変更の方法, 相互の関係などの管理が大変煩雑になる.

そこで, ソフトウェア開発過程を形式的に定義することが考えられた. 開発過程が形式的に定義できれば, 多くの人がその方法を理解し, あいまいさなしに用いることができよう. そして, その方法に基づいて作成されるソフトウェアが, ある水準以上の品質を持つことを保証したり, 工程の管理を容易に行うことができると期待される. また, 定義されたさまざまなソフトウェア開発技法を比較検討してそれぞれの性質を明らかにしたり, ソフトウェア開発を効率よく行える人の開発方法を定義して誰でも利用したりすることも考えられよう.

さらに, この記述に基づいて作業の内容に合ったツールを順次自動的に起動し, 行うべき作業を開発者に指示するようなシステムがあれば, 開発者は指示された作業のみに専念でき, 一連の作業を効率よく行えるようになるであろう. このようなシステムは開発者の負担の軽減とともに, ソフトウェアの生産性, 品質の向上にも効果があると考えられる.

近年, このような考え方に基づいて, ソフトウェア開発過程を形式的に記述したり, その記述を利用したソフトウェア開発支援環境を作成しようというさまざまな試みがなされている.

Osterweilは, ソフトウェア開発における手順の記述自体をソフト

ウェアと見なすことができると論じている．この考え方をプロセスプログラミングと呼ぶ[Osterweil 87]．OsterweilはＡｄａに基づいた手続き型言語を用いてさまざまなソフトウェア開発の手順を記述しているが，今のところこの記述を計算機上で実行して開発を行ったという報告はない．

Williamsは開発過程を，前提条件，完了条件および人間がすべき作業の３つに分けて記述するＳＰＭ(Software Process Model)というモデルを提案し，これに基づいて開発過程の記述を行っている[Williams 88]．しかしこの記述自体は自然語であり，計算機上で実行することはできない．

Kaiserらは開発過程をルールに基づいて記述，実行するためのシステム Marvelを設計した[Kaiser 88]．Marvelは実際に計算機上で実行可能であるが，ルールを洩れなく記述しなければ作業を適切に進めることができず，また，さまざまな詳細化の記述を行うことも難しい．

これに対し，本論文で述べる開発過程記述用言語ＰＤＬ(Process Description Language)は，代数的言語に基づいた簡明な意味定義を持つ関数型言語である．同一の意味定義の枠内でさまざまな抽象度の記述を行うことができるため，段階的詳細化を適用して開発過程の記述を作成できる．また，動作的モデルに基づく開発過程記述モデルと，段階的詳細化による開発過程記述の作成方法を提案する．この方法を利用することにより，さまざまな開発過程の記述を容易に行うことができる．

以下，第２章では，本研究における開発過程の記述方法についての基本的な考え方を示す．第３章では，開発過程を記述するために新たに設計した言語ＰＤＬおよびそのインタプリタについて述べる．第４章では，段階的詳細化によって開発過程の記述を作成する手順について述べる．第５章では，開発過程の例として，ソフトウェア開発技法のひとつであるＪＳＤ(Jackson System Development)を例にとり，開発過程記述に基づく開発支援系の作成を行なう．

# 第２章　開発過程の記述

## 第１節　序言

ソフトウェア開発過程を形式的に記述し，その記述に基づいた開発支援システムを作成することは，ソフトウェア開発の効率化，開発者の負担の軽減，およびソフトウェアの品質の向上に有効であると考えられる．

本章では，開発過程を分析し，記述を作成するための記述モデルを提案する．次に，開発過程を記述するための言語に求められる要件について検討を加える．

ただし本論文では，議論を簡単にするため，一般性を失わないようにソフトウェア開発作業および開発環境について次のような仮定をおく．

ワークステーション上で一人のソフトウェア開発者がソフトウェアを開発する状況を考える．開発者はエディタやコンパイラなどのいろいろなツールを利用して，仕様書の作成，要求分析，設計書の作成，コーディング，コンパイル，デバッグなどの作業を繰り返し行い，要求されるプログラムや文書を作成する．開発者はこれらの作業を自分一人で行う．チームによる開発や，発注者との話合いなどの人的側面については考えないものとする．本論文では，このような一人のソフトウェア開発者がＵＮＩＸが稼働しているワークステーション上で行うソフトウェア開発の一連の作業系列を開発過程と呼ぶ．

このような仮定をおくことにより，開発者と計算機との関わりを中心にしてソフトウェア開発過程の記述，および支援についての考察を進めることができる．また，この仮定に基づく研究の結果は上で述べたチームによる開発などの，より一般的な開発過程の記述，支援にも拡張することができる．

## 第2節　関連する研究

ここでは，ソフトウェア開発過程の記述，実行に関するいくつかの研究に対して検討，考察を加える．

Osterweil は，開発過程は生成物を作成するための道具あるいは手段であり，開発作業を行うためのメカニズムであると考える．また，実際に行われる個々のソフトウェア開発過程は，その開発過程の記述のひとつの実現(instance)であり，ある開発過程の記述からは，その記述で解決可能な問題に対する開発過程をいくつも生成できるとしている[Osterweil 88]．

Osterweil の主張の重要な点は，このような開発過程の記述が，計算機上のアプリケーションプログラムと同じように，プログラムの形式で行えるとしたことである．この考え方をプロセスプログラミングと言う．プロセスプログラミングではソフトウェア開発をプログラムとしての開発過程を実行することであると考える．また，開発過程の記述をプログラムであると見なすことにより，ソフトウェアの概念であったプログラムの部品化や再利用などを開発過程の記述に対しても適用することができるようになる．

OsterweilらはＡｄａを拡張した手続き型言語，Appl/A を用いて開発過程を記述しようと試みている[Taylor 88]．Appl/A はＡｄａを下敷にすることによって，プロセスプログラミングのさまざまな種類の要求に応える能力を備え，さらに，プロダクトや資源の間の関係の記述を行えるような拡張を加えられている．しかし，現在までのところ，この言語に基づいた記述が計算機上で実行されたという報告はない．

Williamsは開発過程を，前提条件，完了条件，開発者あるいはツールがすべき作業，および各作業間の通信の４つに分けて記述するＳＰＭ(Software Process Model)というモデルを提案し，これに基づいて開発過程の記述を行っている[Williams 88]．

SPM = { 活動 }

活動 = ( {前提条件}, 作業, {完了条件}, {メッセージ} )

ただし, (...) は組(tuple), {...} は集合を表すものとする.

ＳＰＭのように, いつ, どのような作業を行うのかということを中心に, 繰り返し, 前提条件, 完了条件などによって開発過程を記述しようとするモデルを動作的(behavioral)モデルと呼ぶ[Kellner 88][Williams 88]. 前提条件, 完了条件は各開発活動を結び付ける働きをする. また, 作業間の通信という概念を導入することによって, 多人数による複雑な開発過程の記述を行うことも可能となる. しかし, ＳＰＭは記述のための考え方を示しているものの, それ自体を計算機上で実行することはできない.

Kaiserらは開発作業の進行に伴って, あらかじめ記述したルールに基づいて自動的にツールを起動したり, プロダクトの管理を行うことのできるソフトウェア開発環境 Marvelを設計した[Kaiser 87, 88, 89]. Marvelは実際に計算機上で稼働している.

Marvelは生成物やツールを管理するためのオブジェクト群と, 開発過程を表現する拡張可能なルール群から構成されている. Marvelのルールは前提条件, 完了条件, および開発作業からなる. あるルールの前提条件が満たされた時, そこに記述された開発作業を表すツールが起動される. 完了条件は作業の実行状況に応じて真偽の値をとる, いくつかのアサーション(assertion) からなる. ルールには, 開発作業の基本的な記述を与える strategy と, 特定の開発方法に適合させたりするために補助的に使用される hint などがある. Marvel はこれらのルールを利用して, 自動的に開発作業を進めたり, 後戻りさせたりすることができる.

Marvelは使いやすいツール環境を構築できる可能性を持っているが, その反面, ルールの構築, デバッグが行ないにくく, 複数人で利用できる開発開発環境への拡張が困難であるなどの問題点も報告されている[Kaiser 89].

この他にも，ソフトウェア開発過程の記述，実行についての研究や提案が数多くなされているが，一定の評価ができるほどの具体的な成果をあげているものは少ない．

開発過程の中の生成物（作成される文書やプログラム）に注目して，これをＪＳＤ(Jackson System Development)[Jackson 82] と同様にエンティティ(entity)と考え，エンティティの状態遷移に基づいて開発過程を記述する方法が提案されている．この方法をＥＰＭ(Entity Process Model) と呼ぶ[Humphrey 89]．ＥＰＭは動作的記述モデルのひとつである．Humphreyらはこのモデルに基づく記述を，図形的なソフトウェア設計・解析環境である STATEMATE[Harel 88] システム上で行っている[Humphrey 89][Kellner 88]．

同じく生成物に注目した研究として，PenedoによるＰＭＤＢ＋[Penedo 89]，Roberts によるＰＭＬ[Roberts 88]があげられる．ソフトウェア開発におけるさまざまな生成物をデータベース上で管理するモデルとして，すでにＰＭＤＢ(Project Master Database) [Penedo 85] が提案されている．ＰＭＤＢ＋はこのモデルに開発過程と関係した動作の情報を付け加え，開発過程の記述，実行の可能なモデルへ拡張しようというものである．一方，ＰＭＬ(Process Modelling Language) はオブジェクト指向に基づいた言語である．ＰＭＬは，role，interaction という概念を取り入れ，開発者，ツール，エンティティ，行動などを統一的に構成し，また，これらの構造を動的に変化させて行くことができるとする．現在この言語は Smalltalk-80 システム上でプロトタイプを作成中である．

形式的な記述言語を用いて記述を試みた例としては，Garlanの仕様記述言語Ｚによる記述[Garlan 89]と，片山の属性文法に基づく記述モデル[Katayama 89]がある．Garlanは開発すべきソフトウェアをいくつかの部分に分解し，開発者の与えるパラメータを持つ高階関数として記述する方法を提案している．この記述モデルを Domain-Specific モデルと呼び，開発すべきソフトウェアの構成と同時に，開発者の開発作業を記述することができると述べている．片山は属

性文法の形式に従った，階層的関数型ソフトウェア開発過程記述モデル，ＨＦＳＰ(Hierarchical and Functional Software Process)を提案している．ＨＦＳＰは開発過程およびその分解の記述に，資源管理，実行制御の機構を追加したものである．このモデルによる記述は，同じく属性文法に基づく関数型言語ＡＧによって実行することができる．

また，Prologに繰り返しや並列実行などの実行制御機構，ツールの起動やファイル操作の機能を付け加えた言語を設計し，これを用いて開発過程の記述を試みる研究も行われている[Ohki 88]．

## 第3節　開発過程記述モデル

ソフトウェア開発過程を，はじめから完全な形で記述することは実際には困難である．作業の詳細な点は，各開発過程の作業が実際に行われるまでは決定できない．開発過程の記述は，はじめにその開発過程の抽象的な記述を得て，それに次第に具体的な情報を付加して詳細化して行くという方法によって作成することができるであろう．

このようにして記述を行う場合，開発過程を表現するモデルを用意して，その枠組みの中で詳細化を進めていけば，記述の意味を明確にすることができ，容易に記述を作成することができる．

開発過程の記述では，まず，作業全体をいくつかの作業段階に分け，それぞれの作業段階についての記述を行う方法が考えられる．このような各作業段階もまたひとつの開発過程であると考えることができる．ある開発過程は，開発者が自分で行う作業，ツールを利用して行う作業，またはいくつかの開発過程（サブプロセス）の系列のいずれかである．

各開発過程の性質を明らかにするため，それぞれの開発過程ごとに，参照，変更，作成される生成物，作業を行うための条件，および各開発過程間の関係を記述する．この記述をその開発過程についての性質記述と呼ぶ．具体的には各開発過程に対して以下の4項目の記述を行う．

①開発過程の順序関係

サブプロセスの逐次的な実行順序．あるいは実際に開発者，ツールの行う作業．

②前提条件

開発過程での作業で参照，変更される生成物（入力生成物と呼ぶ）が満たしていなければならない条件．

条件が満たされない場合に実行する開発過程も記述する．

③完了条件

開発過程での作業で作成，変更された生成物（出力生成物と呼ぶ）が満たしていなければならない条件.

条件が満たされない場合に実行する開発過程も記述する.

④生成物の受け渡し関係

いくつかに分かれた開発過程間での，入出力生成物の受け渡し関係.

ソフトウェア開発過程全体はこのような性質記述の集合として記述することができる．ある開発過程は別のいくつかの開発過程の系列として記述することができ，開発過程全体を階層的に記述できる．さらに，記述のはじめの段階ではこれらの項目を高い抽象度で記述しておき，次第に具体的な情報を付加し，作業を細分化して詳細化を進めることによって，計算機上で実行可能なレベルの抽象度の記述までを得ることができる.

従って，この性質記述を開発過程記述のための記述モデルとして用いることができる．性質記述は動作的モデルのひとつである．動作的モデルには，作業の流れが直観的に理解しやすく，記述が容易に行えるという特長がある．また，性質記述では開発過程を階層的に記述できるため，段階的詳細化を適用して開発過程記述を作成する目的に適している.

## 第4節　開発過程記述言語

前節で述べたように，ソフトウェア開発過程の開発には段階的な詳細化が有効であると考えられる．例えば，ある開発過程が，仕様の記述，コーディング，コンパイル，およびデバッグの各作業段階からなるという抽象度の高い記述を最初に記述することができる．この開発過程の記述に，例えば，コーディングに用いるテキストエディタは vi であるとか，出力ファイルの名前は src.c であるといった具体的な情報を順次付加してゆく．

また，詳細化によって付加される情報には，開発対象や使用する計算機環境に依存して変化するものも多い．このため，開発過程記述をそのような情報に依存しない形で作成しておき，目的や環境に応じて個別に詳細化して使用したいことがある．同様に，詳細化の際に開発者の使いやすいツールを選ぶことができれば，各開発者毎に使いやすいシステムを構築することができる．

このような観点から，開発過程記述のための言語はさまざまな抽象度のレベルで記述ができなければならず，また，それらの詳細化，記述の変更を容易に行うことのできる機能が要求される．開発過程の記述を段階的詳細化によって開発する場合，各段階で記述の意味が正しく詳細化されていなければならない．このためには，詳細化の各段階において，同一の形式的な意味定義の枠組の中で開発過程が記述できることが望ましい．

開発過程の記述に従ってソフトウェア開発作業を支援するためには，ツールを自動的に起動したり，開発者に対するメッセージを表示したりする機能が必要である．また，ウィンドウ環境でソフトウェア開発を行う場合には，ウィンドウの操作を行ったり，複数のウィンドウで作業を並列に進めたりする機能も必要である．

ところで，オペレーティングシステム（以下，ＯＳと略）は一般に，それ自身のコマンド言語とコマンド言語のインタプリタあるいはコンパイラを持っており，これを通してツール起動やメッセージ

表示を制御することができる．このような言語はさまざまな開発過程を記述する能力を持っているが，多くのコマンド言語は手続き的言語であり，一般に抽象的な記述を行うための形式的な枠組みを備えていない．さらに，コマンド言語は直接そのＯＳに依存する．使用するＯＳを変更した場合，コマンドスクリプトを異なるコマンド言語で書き直さなければならない．このような点から，通常のコマンド言語は開発過程の記述，およびその開発のために十分な機能を持っていないと言うことができる．

## 第5節　結言

本章では，はじめに本論文で考察の対象とするソフトウェア開発過程の範囲を示し，関連するいくつかの研究に関して検討を行った．

次に，ソフトウェア開発過程を記述するために段階的詳細化の方法が有効であることを述べ，ソフトウェア開発過程を記述するための記述モデルを提案した．このモデルは動作的なモデルであり，開発過程を階層的に記述でき，段階的詳細化による記述の作成に適している．

さらに，開発過程を記述するための言語の要件について論じた．開発過程を記述するための言語は，同一の形式的な枠組みの中でさまざまな抽象度のレベルの記述ができなければならない．また，開発作業を支援するために，ツール起動やメッセージ表示といった機能を備えている必要がある．

# 第３章　開発過程記述言語の設計とその実行処理系の作成

## 第１節　序言

本章では，ソフトウェア開発過程の記述を行なうために設計された言語ＰＤＬ(Process Description Language) およびその実行処理系であるＰＤＬインタプリタについて述べる．

ＰＤＬの記述の対象は，一人のソフトウェア開発者がＵＮＩＸの稼働しているワークステーション上で行うソフトウェア開発であるとする．また，ＰＤＬにおいては，このようなソフトウェア開発の一連の作業系列を開発過程と呼ぶ．ＰＤＬはソフトウェア開発過程を，ツール起動，メッセージ表示，ウィンドウ操作などの系列として記述する．

ＰＤＬは代数的言語[嵩86b]の意味定義に基づく関数型言語である．意味定義の簡明さにより，さまざまな抽象度で開発過程の記述をすることができる．このため，段階的詳細化によって開発過程記述の開発を容易に行える．

なお，ツールの起動などを行うコマンド列の記述は，習慣的にプログラムと呼ばず“スクリプト”と呼ぶ．本論文でもＰＤＬのプログラムをスクリプトと呼ぶ．

ＰＤＬインタプリタは，ＰＤＬのスクリプトを実行し，その記述に従ってツールを起動したりメッセージを表示したりすることができる．さらに，ＰＤＬスクリプトの作成および詳細化を容易にするため，さまざまな機能を備えている．ＰＤＬインタプリタは，いくつかのワークステーション上で稼働中である．

以下では，ＰＤＬの設計方針について述べ，構文と意味，および機能について例をあげて説明する．次いでＰＤＬインタプリタの機能と特徴について説明する．

# 第2節　ソフトウェア開発過程記述言語

## 2.1 詳細化機能

本節では，ソフトウェア開発過程を記述する言語の設計方針について検討する．ただし，前述したようなソフトウェア開発過程を記述の対象とし，チームによる複数の計算機を用いた開発や，発注者との話し合いなどの人的側面については考慮しない．

ソフトウェア開発過程を記述するための言語には，さまざまな詳細化のレベルで記述できることや，詳細化が容易に行えることが求められている．このためには，さまざまな抽象度のレベルの記述を同一の形式的な枠組みの中で行うことが考えられる．

ソフトウェアの仕様を形式的に記述するための言語として，代数的言語ＡＳＬ[嵩86b][東野88]が提案されている．従来のプログラミング言語の意味は，一般にコンパイラや特定のプロセッサの動作に基づいており，複雑なものになっている．これに対しＡＳＬでは，式の合同関係を用いて意味定義が行われており，言語の意味が大変簡潔かつ形式的に定義されている．さまざまな抽象度で記述を行うことができ，どの抽象度の記述にも意味定義が形式的に与えられている．従って，記述の正しさの検証を形式的に，比較的容易に行うことができる．

ＡＳＬの特徴は以下のようにまとめることができる．

① 言語の意味が簡明に定義されている．
② さまざまな詳細化のレベルで書きたいことが自然に記述できる．
③ どの抽象度の記述も，同一の意味定義の枠内で記述できる．

これらの特徴は上で述べた，開発過程の記述，詳細化を行う枠組みに対する要求を満たしていると考えられる．

そこで，ＡＳＬの意味定義に基づく関数型言語としてＰＤＬの設計を行った．ＰＤＬは上で述べたＡＳＬの性質を引き継いでおり，

段階的詳細化によるスクリプトの開発を容易に進めることができる.

## 2.2 ツールの起動

実際に開発作業を支援するためには, ツールの起動やメッセージの表示, ファイルの操作など, 計算機システムの機能が利用できなければならない. また, ウィンドウシステム上で効率的な開発を行うために, ウィンドウの操作や複数の作業の並列実行を記述する機能も必要である. ＰＤＬではこれらの機能を特別な組込関数として用意した.

しかし, 計算機システムに対するこのような操作は, ファイルシステムの状態や画面の表示などを変化させてしまう. 一般に関数型言語でこのような状態変化を記述するのは困難である. また, 関数型言語では, 通常, 複数ある引数の評価順序が任意であるために, ツールの起動などの具体的な実行順序が記述しにくいという問題もある.

これらの問題を解決するため, ＰＤＬではシステム状態という概念を導入した. これによって関数型言語の持つ意味の簡明さを損なわずにツールの起動順序を決定し, 計算機システムの状態変化を表現することができる.

# 第３節　開発過程記述言語ＰＤＬ

## 3.1 ＰＤＬスクリプトの概要

ＰＤＬスクリプトは以下の要素から構成される.

(1) 関数定義
(2) マクロ定義（3.4で述べる）
(3) インタプリタへの指示（4.5）
(4) 評価すべき式

具体的なスクリプトの例を図3.1に示す（この例については4.7で述べる）. ①はインタプリタへの指示, ②～④はマクロ定義, ⑤～⑰は関数定義, ⑱は評価すべき式である.

関数定義は以下に示すように, 定義する関数名と仮引数リストを '==' 記号の左辺に, 定義本体の式を右辺に記述する. 定義は複数行にわたってもよい.

関数名 （ 仮引数1 , 仮引数2 , ... ） == 式 ;

式は組込関数, 定義関数, 定数および左辺に現れる仮引数（変数）から構成される. 表3.1にＰＤＬの組込関数の一部を示す. このうち, 四則演算, 比較演算などは中置記法で用いることができる. また, if関数は

if 条件式 then 式1 else 式2

という形式をしており, 条件が真の場合式1, 偽の場合式2をこのif関数の値とする.

ＰＤＬでは以下のデータ型を使用できる.

整数　　文字列　　論理
システム状態　　タプル（並び）

```
#include /usr/users/pdl/strings                          ①
#let EDITOR: "vi "                                      ②
#let MANUAL: "man "                                     ③
#let CC(src,obj,S): exec("cc -o "+obj+" "+src,S)        ④

C_proc(src,S) ==
    Execute(src,
        element1(Compile(src,Create(src,S)):C),
        element2(C));                                   ⑤
Create(src,S) == Edit(src,S);                           ⑥
Correct(src,S) ==
    S @ wclose(Edit(src,wopen(S)));
Compile(src,S) ==                                       ⑦
    if element1(CCompile(src,Obj(src),S):T)
    then [Obj(src),element2(T)]
    else Compile(src,Correct(src,element2(T)));         ⑧
Execute(src,obj,S) ==
    if element1(Check(Run(obj,S)):T)
    then element2(T):T2
    else Execute(src,
        element1(
            Compile(src,Correct(src,T2)):C),
        element2(C));                                   ⑨
Edit(src,S) == exec(EDITOR+src,S);                      ⑩
CCompile(src,obj,S) ==
    [status(
        CC(src,obj,write("*compile*",S)):T)
     - 0, T];                                           ⑪
Obj(src) == substr(src,0,strlen(src)-2);                ⑫
Run(obj,S) == exec(obj,write("*run*",S));               ⑬
Check(S) == [read("OK?",S)="y", S];                     ⑭

C_dev(src,S) == wclose(C_proc(src,wopen(S)))
              @ wclose(Refer(wopen(S)));                ⑮
Refer(S) == if read("manual> ",S):K="q" then S
            else Refer(Manual(K,S));                    ⑯
Manual(key,S) == exec(MANUAL+key,S);                    ⑰

C_dev("test.c", S);                                     ⑱
```

図3.1 Ｃプログラム開発用ＰＤＬスクリプト

## 表3.1 ＰＤＬの主な組込関数

| 関数と引数 | 結果 | 意　　味 |
|---|---|---|
| I + I | I | 加算 |
| C + C | C | 文字列の連結 |
| I - I | I | 減算 |
| - I | I | 負号化演算 |
| I * I | I | 乗算 |
| I / I | I | 除算 |
| X = X | B | 等しい |
| X <> X | B | 等しくない |
| X > X | B | 大きい (1) |
| B \| B | B | 論理和 |
| B & B | B | 論理積 |
| ! B | B | 論理否定 |
| S @ S | S | 並列演算 |
| exec(C,S) | S | Ｃシェルコマンドを実行する |
| execstr(C,S) | [C,S] | Ｃシェルコマンドを実行し，結果の文字列を得る |
| status(S) | I | Ｃシェルのステータスコードを得る |
| wopen(S) | S | ウィンドウを開く |
| wclose(S) | S | ウィンドウを閉じる |
| read(S) | C | ウィンドウから一行読み込む |
| write(C,S) | S | ウィンドウに文字列を書き出す |
| break(X) | X | ユーザ設定ブレーク |
| element1([...]) | X | タプルの１番目の要素を取り出す (2) |
| strlen(C) | I | 文字列の長さを得る |
| atoi(C) | I | 文字列を整数に変換する |
| itoa(I) | C | 整数を文字列に変換する |
| field(C,I) | C | 空白を区切りとするＩ番目のフィールドを得る |
| if B then X1 else X2 | | Bが真ならX1，偽ならX2を値とする |

引数と結果の型は以下の通りとする．

S : システム状態　　I : 整数　　C : 文字列
[...] : タプル　　B : 論理　　X : 任意

(1) 大小比較には >, <, >=, <= がある． X は整数あるいは文字列．
(2) element1,element2,...,element10まで用意されている．

タプルはいくつかのデータの組で，

［ 式1， 式2， ... ］

のように表現する．システム状態については 3.2で述べる．

ＰＤＬでは，記述のしやすさから各定義関数の引数や結果のデータ型を宣言しない．ＰＤＬの意味定義上，それらのデータ型は固定されていなければならないため，場合によって異なるデータ型を返すような関数は（表記上可能であるが）定義してはならない．

## 3.2 ＰＤＬ関数の意味

関数の意味は式の合同関係を用いて定義される[嵩86b]．組込関数の定義（例えば 1+1==2，1+2==3，... ）および定義関数の定義は，左辺と右辺の式が関数の適応という演算で閉じた合同関係を表すものとする．この合同関係から得られる最小の同値類分割を考え，評価すべき式の同値類中の構成子項（定数）をその関数の意味（値）と定義する．

ＰＤＬはデータ型のひとつとしてシステム状態を持つ．システム状態とは，ファイルシステムやその他の計算機資源の状態すべて，および開発者の状態を抽象的に表すもので，実行中のどの時点においてもある値が１つだけ存在する．組込関数（ツールの起動やウィンドウ操作など）は，あるシステム状態を引数として受け取り，その値を更新し，次の関数に渡す．

形式的には，このシステム状態のある値に，順次ツールの起動，ウィンドウのオープン等の遷移関数を施して目的のシステム状態にすることが，ＰＤＬのスクリプトの意味である．

図3.1の例では，Ｓという名前の仮引数はすべてシステム状態型である．

ＰＤＬの組込関数の意味定義は，代数的言語の枠組みの中で形式的に与えることが可能である．記述の基本となる組込関数の意味定義が形式的に与えられているため，記述全体が代数的な枠組みのな

かで定義されていることを示すことができる．ここでは組込関数の意味定義の具体的な記述などについては触れないこととする．

### 3.3 システム状態を更新する組込関数

ここではＰＤＬ特有の，システム状態を更新する組込関数について述べる．

(1) システム機能の呼び出し — exec

ＵＮＩＸ上でツールを呼び出すには，Ｃシェルというコマンドインタプリタにコマンドの文字列を与えるのが容易な方法である（本論文では 4.2, 4.3 BSD あるいはこれらと同等な BSD系のＵＮＩＸについて考える）．ＰＤＬからＣシェルの機能を利用するには組込関数 exec を用いる．execの引数は文字列とシステム状態であり，文字列をＣシェルにそのまま渡して実行させ，新しいシステム状態の値を返す．

図3.1⑩の関数Editは，マクロで定義した実際のツール名 "vi " と引数のファイル名，例えば "test.c" を連結して文字列 "vi test.c" をＣシェルに渡す．ここでは，演算子 + は文字列の連結である．

ＰＤＬスクリプトではこれらの基本的な操作をもとに，より抽象度の高い記述を行うことができる．例えば，

```
list(file,S) == exec("more "+file,S);
edit(file,S) == exec("vi "+file,S);
```

と定義しておいて，関数 list, edit を使用するように記述すれば，実際のツール名 more, vi を陽に用いないでスクリプトを構成することができ，後からの変更が容易である．

(2) ウィンドウの操作 — wopen, wclose

ＰＤＬではウィンドウをいくつか開いてその中でツールを起動し，作業を効率よく行うことができる．ウィンドウを操作するための関数として wopen, wclose がある． wopenは新しいウィンドウを開き，wcloseはウィンドウを閉じる．どちらもシステム状態を引数とし，新しいシステム状態の値を返す．

例えば，

```
wclose(exec("vi tmp",wopen(S)));
```

という式では，新しいウィンドウを開いてその中でエディタ vi を起動し，編集が終了してからウィンドウを閉じることを表している．

(3) 並列演算子 — @

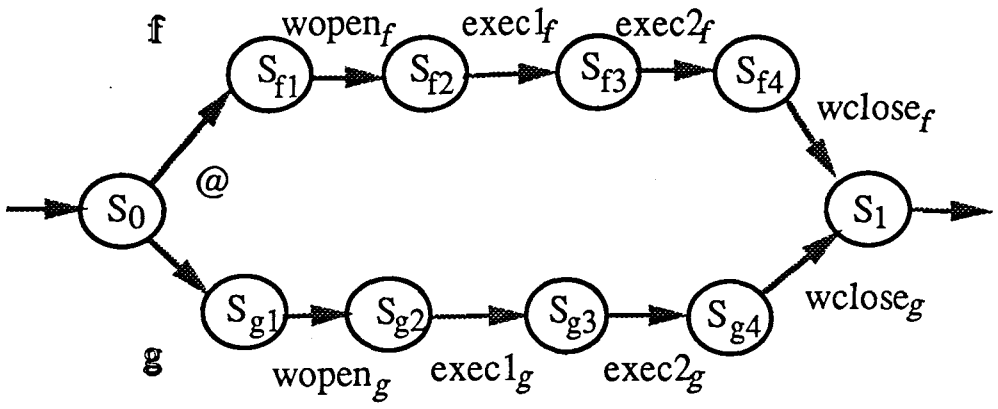
ソフトウェア開発では複数の作業を並列に行いたいことがある．例えば，別のプログラムやオンラインマニュアルを参照しながらプログラムの作成を行うような状況が頻繁に生じる．

このような並列した作業を記述するために，ＰＤＬでは新たに組込関数 @ （並列演算子と呼ぶ）を導入した．この関数は２つの作業を並列に開始させ，最後に双方の同期をとる．引数は２つのシステム状態の値であり，新しい１つのシステム状態の値を返す．

今，関数 f, g, w が以下のように定義されているとする．

```
f(S) == wclose(exec("vi src",wopen(S)));
g(S) == wclose(exec("less spec",wopen(S)));
w(S) == f(S) @ g(S);
```

関数 w を起動するとウィンドウが２つ開き，一方でテキストエディタ vi が，他方でファイル表示ツール less が起動され，２つのツールを並列に使うことができる．両方のツールの使用が終了するとウィンドウは閉じられ，関数 w は新しいシステム状態の値を返して




(a) システム状態の遷移

wopen$_f$ wopen$_g$ exec1$_f$ exec1$_g$ exec2$_f$ exec2$_g$ wclose$_f$ wclose$_g$

wopen$_g$ wopen$_f$ exec1$_f$ exec1$_g$ exec2$_f$ exec2$_g$ wclose$_f$ wclose$_g$

wopen$_f$ wopen$_g$ exec1$_g$ exec1$_f$ exec2$_f$ exec2$_g$ wclose$_f$ wclose$_g$

wopen$_g$ wopen$_f$ exec1$_g$ exec1$_f$ exec2$_g$ exec2$_f$ wclose$_g$ wclose$_f$

wopen$_f$ exec1$_f$ wopen$_g$ exec1$_g$ exec2$_f$ exec2$_g$ wclose$_f$ wclose$_g$

...

wopen$_f$ exec1$_f$ exec2$_f$ wclose$_f$ wopen$_g$ exec1$_g$ exec2$_g$ wclose$_g$

(b) 関数の適用の組み合せ

図3.2 並列演算子とシステム状態

終了する.

並列演算の返す値は，引数のシステム状態に対して @ の左側の式で得られる値と右側の式で得られる値の“合成”と考える．ここでいう合成されたシステム状態とは，@ の左側の関数列と右側の関数列によるシステム状態の更新が，互いに重なり合って起こった結果のシステム状態である．重なり方は引数のシステム状態 S，すなわち，その時の計算機のハードウェア，ソフトウェア，およびユーザの操作に依存する.

上で示した関数 w の場合を例として図3.2に示す．ここで，組込関数 exec はさらに，「ツール起動開始」と「ツール起動終了」を表す関数 exec1，exec2 の2つからなると考える．図3.2(a)に示すように，関数 f，g で定義された関数の列はそれぞれ逐次的に実行されるが，これらの関数がシステム状態に適用される順番としては図3.2(b)のようにさまざまな組合せを考えることができる．実際にどのような順番で関数が適用されるかは，その時のシステム状態に依存して決定される.

並列演算子は並列した作業間の通信や共通の資源へのアクセス制御などの機能は備えていない．しかし，多くの開発過程の作業は逐次的に進行するため，この並列演算子で簡明に表現できる.

## 3.4 マクロ定義

ＰＤＬではグローバルマクロとローカルマクロの2種類のマクロ記法を用いることができる.

グローバルマクロの定義を行うと，関数定義内に現れるマクロ名は定義した文字列で置換される．グローバルマクロには図3.1②，③のように引数を持たないもの，④のように引数を持つものがある．図3.1⑩，⑪のマクロはそれぞれ次のように展開される.

```
Edit(src,S) == exec("vi "+src,S);
```

```
CCompile(src,obj,S) ==
    [status(
        exec("cc -o "+obj+" "+src,
                write("*compile*",S)):T) = 0, T];
```

グローバルマクロはスクリプトの段階的な詳細化に有効である. 例えば図3.1の例のように, 始めは EDITOR や MANUAL といった抽象的なツール名を用いて記述を行い, 後でグローバルマクロを用いて "vi " や "man " という具体的なツール名を与えることができる.

ローカルマクロは通常の関数定義の記述の一部として定義され, その関数定義内でのみ有効である. ローカルマクロは式の記述の中で以下のように定義する.

因子 : マクロ名

因子とは仮引数, 関数呼び出し, または括弧でくくられた式のことである. 関数定義内部のマクロ名は定義した因子に置き換わる. 例えば, 図3.1⑤の定義式はマクロ名 C の置換によって以下のようになる.

```
C_proc(src,S) ==
  Execute(src,
    element1(Compile(src,Create(src,S))),
    element2(Compile(src,Create(src,S))));
```

ＰＤＬは関数型言語であり, 手続き型言語でいう変数の概念を持たない. このような言語では同一の式の値を複数箇所で利用する場合が多く, それぞれの箇所にその式全体を記述したり, 別の関数にしたりする必要がある. ＰＤＬではローカルマクロを利用することによってこのような煩雑さを取り除くことができ, わかりやすい記述を行うことが可能である.

# 第4節　ＰＤＬインタプリタの機能

ＰＤＬインタプリタは端末，あるいはファイルからスクリプトを入力し，これらを解釈実行する．このインタプリタは単にスクリプトを実行するだけではなく，スクリプトの記述や詳細化を容易にするようなさまざまな機能を備えている．

## 4.1 式の評価

インタプリタは式の記述を直接与えられるとそれを評価し，結果の値を返す．式は，関数定義の内側，左側から順次評価される．関数の引数には実引数を評価した値が渡される(call by value)．この評価方法は容易に実現でき，実行効率もよいが，結果の値が存在するのに評価が停止しない場合が原理上起こりうる．しかし，これまでに試みた通常の開発過程の記述ではそのようなことは起こらず，実用上問題がなかった．

関数の定義式中に同一の部分式が複数現れた場合，その部分式は一度だけ評価され，保存される．その値が再び必要になったときには保存された値が参照される．すなわち，部分式の評価はたかだか１回である．また，組込関数に渡される値の型検査を実行中に行う．

## 4.2 システム状態の取扱い

システム状態型の値は整数型などの他のデータ型の値と異なり，値のコピー，保存ができない．

いま，下の (1)の関数について考える．これを実行すると，まずif関数の条件判定部で組込関数exec によってシステム状態が更新される．その結果によって then部，else部いずれかが実行されるが，いずれの場合も，writeの引数のシステム状態は条件判定部で更新される前の値 S を指定しており，この通り実行することはできない．

一方，(2)は，更新されたあとのシステム状態の値 S1に対して

write を実行することを記述しており，実行可能である．通常，プログラマは (2)のように，常に最新のシステム状態の値に対して関数を施すように記述する必要がある．(1)の例のように，誤ったシステム状態が指定された場合には，インタプリタが実行時に発見し，ユーザに警告を出すことができる．

```
compile(src,S) ==
    if status(exec("cc "+src,S))=0
        then write("success",S)
        else write("error",S);          (1)

compile(src,S) ==
    if status(exec("cc "+src,S):S1)=0
        then write("success",S1)
        else write("error",S1);         (2)
```

## 4.3 未定義関数の処理

組込関数以外で，その定義が与えられていない関数を未定義関数という．段階的詳細化によってスクリプトの開発を行う場合には，未定義関数とは，ある詳細化のレベルでは実行可能な定義の与えられていない関数のことを表す．

スクリプトを実行中に未定義関数の値が必要になった時，ＰＤＬインタプリタは実行をそこで一時中断してブレークモードに入る．そこでユーザは次のいずれかの動作を選択することができる．

(a) その関数値を一時的に与え，実行を再開する
(b) その関数の定義を与え，実行を再開する
(c) 実行を中止する．

ブレークモードとは，実行が中断している時のインタプリタの状態をいう．ユーザはブレークモード内でも通常と同様に式を評価したり，インタプリタに指示を与えたりすることができる．

(a)の場合，ユーザは

```
#cont 値
```

という指示を与える．(b)の場合は関数の定義をしてから

```
#cont
```

とする．(c)の場合には

```
#top
```

とすると実行は中止され，ブレークモードも終了する．

この未定義関数処理機能により，未定義関数を含むスクリプトでも，ユーザが実行中に値を決めたり関数定義を行いながら実行を進めることができる．この機能はスクリプトを実際に実行しながら詳細化を進める場合に有効である．

## 4.4 デバッグ

ブレークモードに入るには以下の3つの場合がある．

(a) 未定義関数を評価した場合
(b) 端末から中断文字（通常 control-C）が入力された場合
(c) 組込関数 break が実行された場合．

ブレークモードでは，引き続く実行を関数単位で行わせることができる．これをステップ実行という．

```
#step
```

この指示を与えるとインタプリタは関数をひとつ評価してその関数

名と値を表示し，再びブレークモードに戻る．

また，実行した関数名とその値を表示させながら実行するトレース実行の機能も備えている．これらの機能により，スクリプトのデバッグを容易に行うことができる．

また，デバッグや関数の編集を容易に行うことができるように，ウィンドウを活用したデバッグ専用のインタフェースを用意している．図3.3にデバッグ用インタフェースを示す．

上半分のサブウィンドウは，関数定義を編集するためのウィンドウである．下半分のサブウィンドウはＰＤＬインタプリタへの指示を入力したり，インタプリタからのメッセージが表示されるウィンドウである．ウィンドウの右側にはいくつかのボタンが用意されている．#step や #cont などの指示はこのボタンをクリックするだけで与えることができる．また，関数の編集中は，このボタンの操作だけで，指定した位置に break関数を挿入したり，削除したりすることができる．

## 4.5 インタプリタへの指示

インタプリタに指示(directive)を与えることによってさまざまな機能が利用できる．指示にはスクリプト内部に含めることができるものもある．インタプリタへの指示を表3.2 に示す．

以下では，これらのうちの主なものについて説明する．

### (1) スクリプトファイルの読み込み

インタプリタには，以下のような指示を与えることによって，あらかじめスクリプトを記述したファイルを読み込むことができる．

```
#include  ファイル名
```

また，この指示はスクリプト内に記述しておくことができる（図3.1①）．この指示を利用すればスクリプトをモジュール化して記述できる．また，頻繁に使われる関数定義やＯＳなどに依存する記述

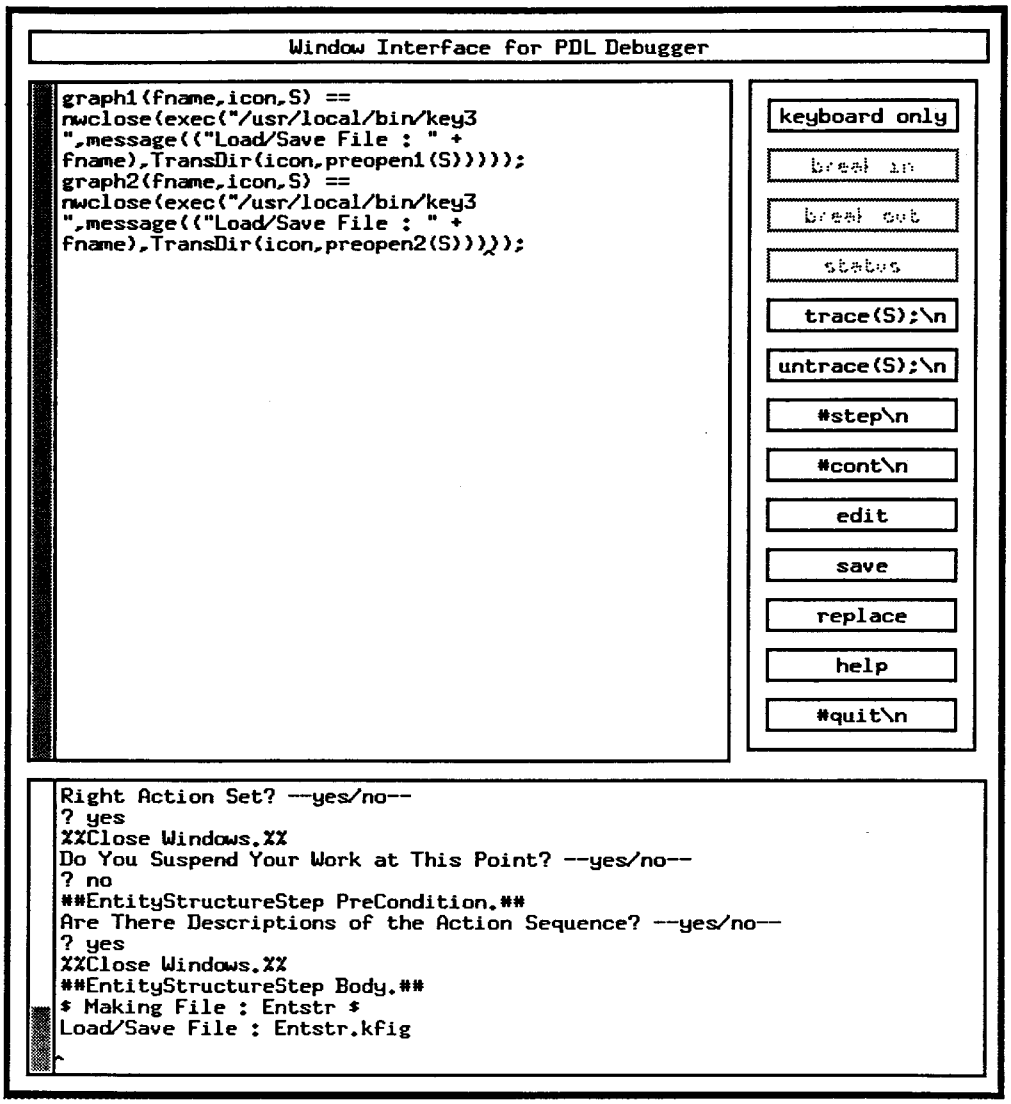



図3.3 ＰＤＬデバッグ用インタフェース

## 表3.2 ＰＤＬインタプリタへの指示

| | |
|---|---|
| #let マクロ名... : 文字列 | マクロ定義 |
| #unlet マクロ名 | マクロ定義の消去 |
| #unletAll | すべてのマクロ定義の消去 |
| #undef 関数名 | 関数定義の消去 |
| #undefAll | すべての関数定義の消去 |
| #new | すべての関数，マクロ定義の消去 |
| | |
| #ifdef 関数名 | 条件ブロックの開始（関数が既定義なら実行） |
| #ifndef 関数名 | 条件ブロックの開始（関数が未定義なら実行） |
| #iflet マクロ名 | 条件ブロックの開始（マクロが既定義なら実行） |
| #ifnlet マクロ名 | 条件ブロックの開始（マクロが未定義なら実行） |
| #else | else部の開始 |
| #endif | 条件ブロックの終了 |
| | |
| #include ファイル名 | ファイルからのスクリプト入力 |
| | |
| #list [関数名...] [> ファイル名] | 関数定義の表示 |
| #show [マクロ名...] [> ファイル名] | マクロ定義の表示 |
| | |
| #history | ヒストリの表示 |
| #n | ヒストリの呼び出し（n は数字） |
| #edit 関数名... | 関数定義の編集 |
| | |
| #step | ステップ実行 |
| #cont | 実行の継続 |
| #top | ブレークモードを終了する |
| #debug | デバッグインタフェースを使用する |
| #quit | デバッグインタフェースを終了する |
| | |
| #exit | ＰＤＬインタプリタの終了 |
| #help | インタプリタへの指令の一覧を表示 |

をファイルにまとめてライブラリとして利用すれば，スクリプトの記述をわかりやすく容易に行うことができる．使用しているＯＳやウィンドウシステムに依存する部分をライブラリ化することにより，特定の環境に依存しないスクリプトを作成することも容易である．

インタプリタの起動時に，読み込むスクリプトファイル名を指定することができる．また，ユーザのホームディレクトリに .pdlrc というファイルがあれば，インタプリタの起動時にこのファイルが自動的に読み込まれる．これによって，それぞれの開発者に適した開発環境の設定を行うことが可能である．

(2) 条件付き解釈

スクリプトの一部を条件付きで解釈させることができる．

```
#iflet マクロ名
  (1)
#else
  (2)
#endif
```

スクリプト内にこのように記述しておくと，与えたマクロ名が定義されていた場合に (1) の部分が解釈され，定義されていなければ (2) の部分が解釈される．#else および (2) の部分はなくてもよい．逆に，マクロ名が定義されていない場合に (1) の部分を行うには，#iflet の代わりに #ifnlet を用いる．例えば，

```
#ifnlet FILES
#let FILES: 10
#endif
```

と記述しておけば，マクロ名 FILES が定義されていない場合のみ，

2行目の定義が有効になる. #iflet - #else - #endif の構造はネストさせることができる.

(3) エディタの呼び出し

関数名を指定してエディタを呼び出し, 定義の編集を行うことができる.

#edit 関数名...

とすると, 指定した関数を画面上で編集できる.

(4) ヒストリ機能

インタプリタに対する以前の指示を繰り返し用いることができる.

#history

とするとこれまで与えた指示の履歴が番号つきで表示される. ここでその番号(例えば15)を使って #15 と入力すれば, 対応するコマンドを再び入力したのと同じことができる.

## 4.6 PDLインタプリタの実現方法

PDLインタプリタの基本的な実行方式は, ASL/Fなどの関数型言語における関数評価方法[井上84]と同様である. ここでは, システム状態を更新するいくつかの組込関数の実現方法[荻原88]について簡単に触れる.

なお, 本節の文中の“プロセス”とは, UNIXにおけるプログラム実行の単位である.

(1) ツールとウィンドウを操作する組込関数の実現

PDLインタプリタは自由にウィンドウを開き, その中でツールを起動できる必要がある. このためには, PDLインタプリタからウィンドウプログラム(例えば X-Window では xterm など)を起動

し，そのウィンドウプログラムからＣシェルを起動する．そして，ＰＤＬインタプリタからこのＣシェルにコマンドの文字列を次々に渡さなければならない．

通常ＵＮＩＸでは，プロセス間通信の方法としてパイプやソケットという機構が用いられる．しかし，パイプでは直接起動したプログラム以外との通信が困難である．また，ソケットもソケットで通信できるように作られたプログラム間でしか利用できない．

そこで，ここではＰＤＬインタプリタとＣシェルとの通信のためにＵＮＩＸの擬似端末(pseudo terminal)機能を利用した．擬似端末とは仮想的な端末装置（デバイス）であり，実際の端末装置と同様の振舞いをするが，それへの入出力はプログラムによって制御することが可能である．ＵＮＩＸでは装置も一種のファイルであると考えているため，擬似端末をＣシェルのスクリプトファイルとして渡せば，Ｃシェルは擬似端末からコマンドを読み込んで実行を行うことができる（図3.4）．この実現方法はウィンドウプログラムにもＣシェルにも変更を加えず，しかも移植性が高い．

(2) 並列演算の実現

ＰＤＬインタプリタではまた，組込関数 @ の引数が表す作業をそれぞれ並列に実行しなければならないが，複数の作業をひとつのインタプリタで同時に制御するのは大変困難である．そこで，並列に動作する作業のそれぞれについてインタプリタのプロセスをひとつずつ作って作業を制御させるという方法を用いている．

いま，

```
f(S) == g(S) @ h(S);
```

と定義された関数 f を実行すると，ＰＤＬインタプリタはまず子プロセスをひとつ作り，自分（親）は関数 g の実行に移る．子プロセスは関数 h を実行し，実行が終了すると消滅する．親プロセスは関

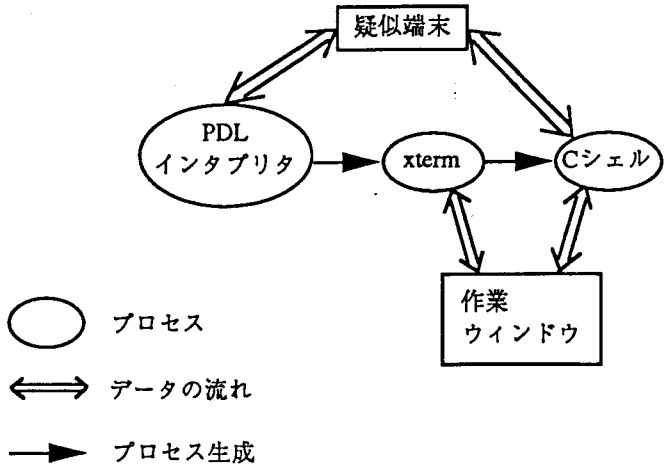



図3.4 インタプリタ，ウィンドウとCシェルプロセスの関係

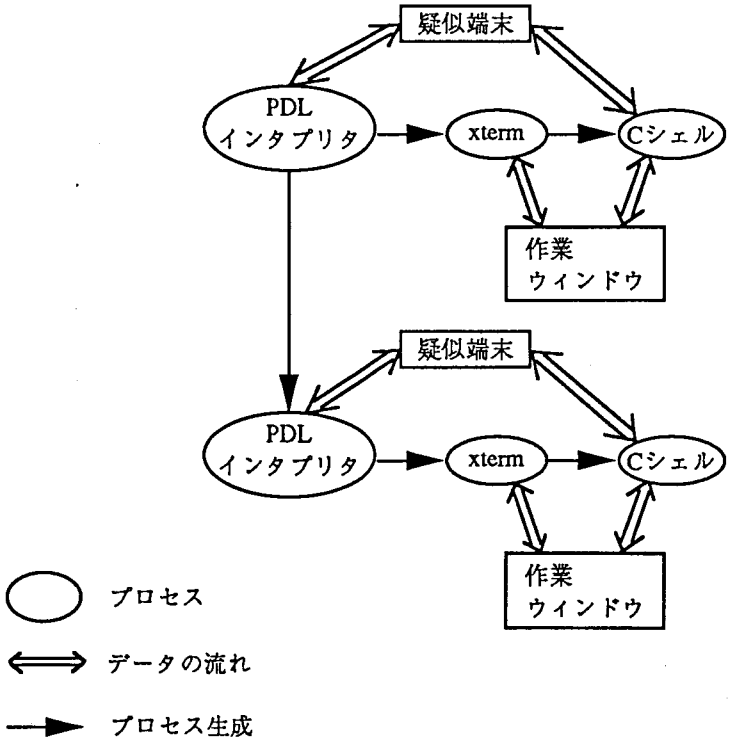



図3.5 並列演算の実現

数 g を実行し終わると子プロセスの終了を確認し，新しいシステム状態を作って関数 f の値とする．

組込関数 @ を使わないで実行している場合，ＰＤＬインタプリタ，作業ウィンドウ，Ｃシェルの関係は図3.4で説明した通りである．関数 @ を使って複数のウィンドウで作業を行う場合，ＰＤＬインタプリタ，作業ウィンドウ，Ｃシェルは図3.5のように複数組生成される．

## 4.7 ＰＤＬによるスクリプト記述と実行例

ここでは，図3.1に示した簡単なＣプログラム開発過程のスクリプトの記述，詳細化の概略と実行例を示す．

### (1) 作業の流れの記述

作業の流れの概要は図3.6のように表される．最初にプログラムを作成し，次にコンパイルを行う．コンパイルが成功すればオブジェクトプログラムの実行を試みるが，失敗すればプログラムの修正を行う．実行の結果が要求を満たしていれば開発作業は終了するが，そうでなければプログラムの修正に戻る．

この開発過程をＰＤＬで記述するため，プログラムの作成，コンパイル，修正，実行のそれぞれの開発段階を，Create，Compile，Correct，Execute という関数で表す．開発過程全体を表す関数を C_proc とすれば，これは図3.1⑤のように定義できる．

### (2) 記述の詳細化

次に，テキストエディタの起動を表す関数 Edit やＣコンパイラの起動を表す関数 CCompile などを用いると，各開発段階を表す関数は⑥～⑨のように定義できる．さらに，Edit，CCompile などはＰＤＬの組込関数を使って⑩～⑭のように定義できる．

以上のような段階的な詳細化によって定義した関数 C_proc は図3.6の手順通りに実行することができる．

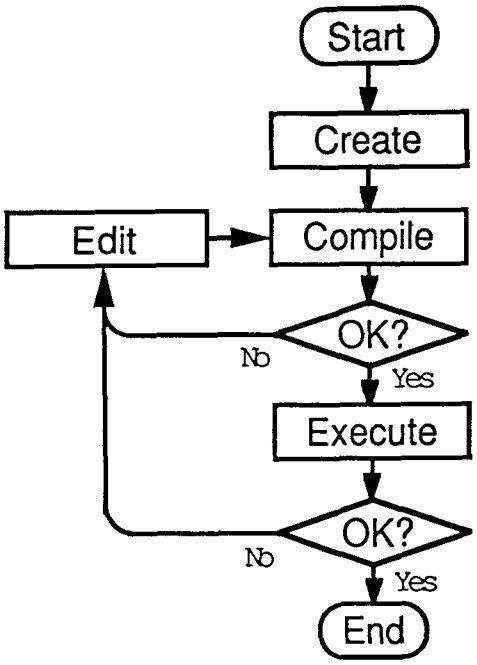



図3.6 Cプログラム開発作業の流れ

(3) 実行例

図3.1ではさらに⑮～⑱の定義を加え，関数 C_proc の実行と並列に，オンラインマニュアルを参照するための関数 Refer が起動できるようにした．

このスクリプトを実際にＰＤＬインタプリタで実行した時の画面の例を図3.7に掲げる．左上のウィンドウから関数 C_dev を起動して実行を開始する．残り３つのウィンドウは左からそれぞれ，エディタ用，コンパイル用，オンラインマニュアル用のウィンドウである．なお，この画面は Sun3の SunView上で実行した例である．

図3.1のスクリプトでは，⑤～⑨が Edit, CCompile などの抽象的なツールの概念を用いた記述であり，⑩～⑭がそのツールの概念を具体的なツールに対応させて定義した記述になっている．ＰＤＬを用いると，このように詳細化のレベルを段階的に分けて，開発過程をわかりやすく記述，詳細化することができ，またスクリプトの変更も容易になる．

このスクリプトを実行すると，開発作業を図3.6の手順通りに進めることができる．ツールの起動を手作業で行う場合のような煩わしさなしに開発を進められ，ツールの具体的な起動方法などについての詳細な知識も必要としない．

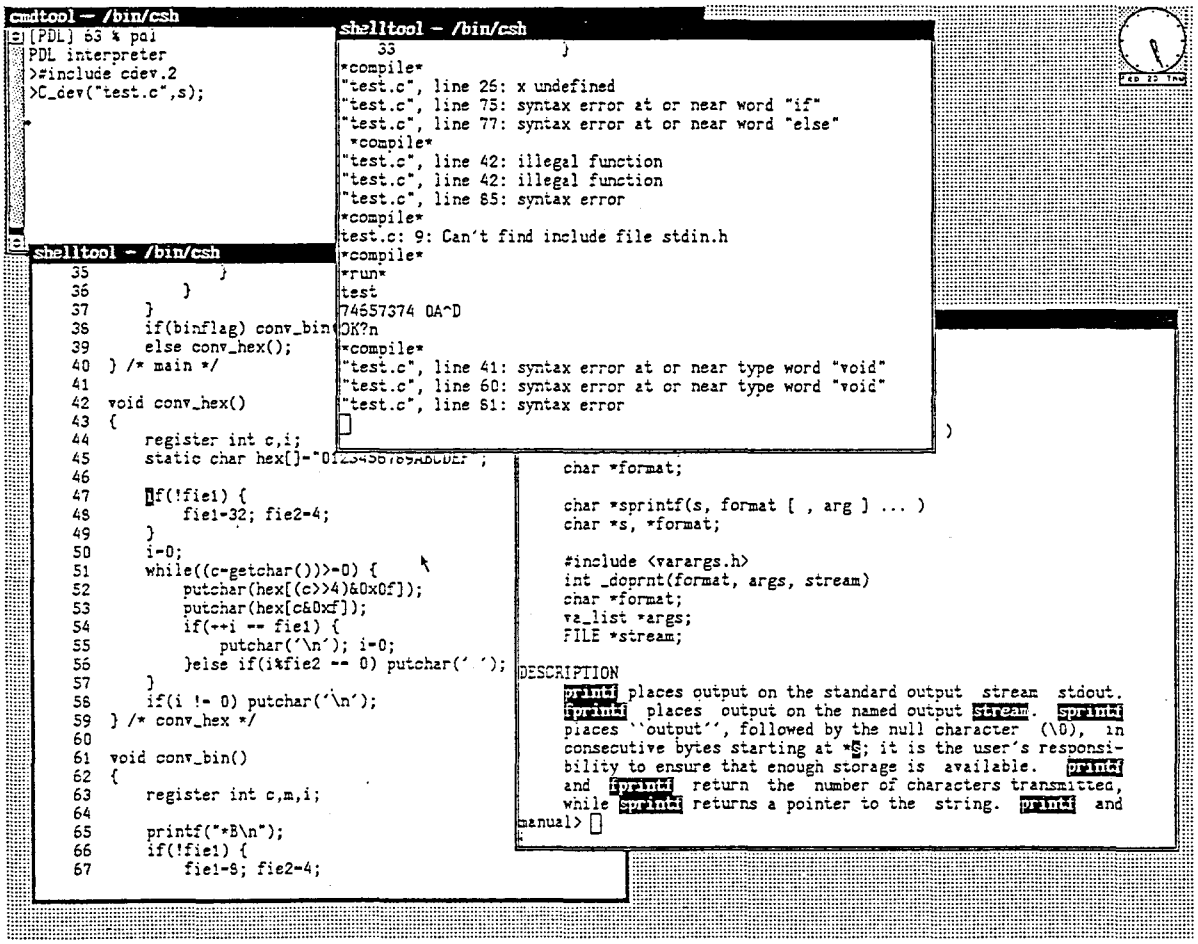



図3.7　ＰＤＬスクリプトの実行例

## 第5節　結言

本章では，ソフトウェア開発過程を記述し，ツール起動やメッセージ表示を制御するための言語ＰＤＬを提案し，言語とインタプリタの特徴について述べた．

ＰＤＬは代数的言語ＡＳＬと同様の意味定義を持つ関数型言語である．さまざまな抽象化のレベルでの記述が可能であり，また，詳細化の正しさなどの検証を比較的容易に行うことができる．このため，段階的な詳細化によるスクリプトの開発に適している．

ＰＤＬは基本関数としてツールの起動，ウィンドウの操作，並列した作業を行う関数などを備えている．また，ソフトウェアおよびハードウェアの状態を抽象的に表すシステム状態を基本データ型として持ち，開発過程を形式的に記述することが可能である．

また，記述しやすさの向上，スクリプトのライブラリ化などに役立つさまざまなマクロ機能がある．

ＰＤＬインタプリタは，抽象度の高い，未定義関数を含むＰＤＬスクリプトでもユーザの指示に従って実行を継続できるような機能を備えている．また，実行をトレースしたり，システム状態の誤用に警告を表示するなど，スクリプトの開発を支援するためのさまざまな機能を持っている．これらの機能を効率よく利用するために，デバッグ用のウィンドウインタフェースも用意している．

ＰＤＬインタプリタは，約６人月の作業量をかけて作成された．使用言語はＣで，ソースコードはおよそ8500行である．ツール起動やウィンドウ操作，メッセージの表示などの目的には十分な実行速度を持つ．

ＰＤＬインタプリタはＵＮＩＸ上のウィンドウシステムを用いて実行する．ＰＤＬインタプリタは現在，IBM RT/PC, Sony NEWS, および DEC Ultrix の X-Window 上，Sun3, SPARC の SunView および X-Window上で稼働している．ただし，デバッグ用インタフェースは X-Window上でのみ利用可能である．

# 第４章　段階的詳細化に基づく開発過程の記述方法

## 第１節　序言

本章では，ソフトウェア開発過程を記述するための，段階的詳細化に基づく手法の提案を行う．次いで，開発過程記述に基づくソフトウェア開発環境（ＳＤＥ：Software Development Environment）の構築について論じる．

第３章では，ツールの起動，ワークステーション上のウィンドウの操作，および開発者へのメッセージの表示などを制御するための言語ＰＤＬについて論じた．ＰＤＬのスクリプトは開発過程の記述と考えることができ，さらに，その開発過程を支援するシステムの仕様であるとみなすことができる．

ＰＤＬは代数的言語[嵩86b]に基づく関数型言語である．他の代数的仕様記述言語と同様に，ＰＤＬのスクリプトは抽象的なスクリプトから具体的なスクリプトへの段階的詳細化によって開発することができる．段階的詳細化を用いることにより，抽象度のレベルの高いスクリプトの持つ性質は，抽象度のレベルの低いスクリプトに引き継がれる．スクリプトを変更する必要がある時には，適切な抽象度のレベルに戻って詳細化をやり直すことによって，具体的なスクリプトを効率的に得ることができる．本章ではソフトウェア開発過程に対する段階的詳細化の方法を示す．この方法はさまざまな開発過程に対して広く適用可能である．

ソフトウェア開発過程の形式的な記述への要求が高まると同時に，開発過程記述を変更したり，目的に合うようにカスタマイズしたりすることの必要性も論じられるようになってきた．開発過程の記述に基づいたソフトウェア開発環境（以下，ＳＤＥと記述）において

は，開発過程記述を変更，カスタマイズすることはＳＤＥ自体を変更，あるいはカスタマイズすることと見なすことができる．

これまでに開発された多くのＳＤＥは，特定のプログラミング言語，プログラム構造，ツール，あるいは開発技法に直接依存しており[Dart 87]，変更や改編の可能性は一般には限定されている．しかし，ソフトウェア開発過程記述に基づいたＳＤＥでは開発過程記述の変更によって，特定の開発者の習慣や特定の開発作業に応じたカスタマイズ，効率向上のための調整や拡張，システムの変更への対応などを柔軟に行うことが可能になると考えられる．

開発過程の記述と同様に段階的詳細化の考え方を利用することによって，このような変更，カルタマイズの可能なＳＤＥを構築することができる．

## 第2節　スクリプトの詳細化

### 2.1 段階的詳細化法の概要

ＰＤＬのスクリプトを作成するためにさまざまな方法を考えることができる．具体的なスクリプトを直接記述する方法や，はじめに抽象度の高いスクリプトを記述しておいて，必要な抽象度のレベルまでいくつかの段階を経て詳細化する方法などがある．後者の方法は，正しいスクリプトを構成したり変更したりする労力を減らすことができる点で有効であると考えられる．

抽象度の高いスクリプトを具体的なスクリプトに詳細化する方法にもさまざまなものがある．一般には，抽象度の高いスクリプトに開発過程の基本的な性質を記述し，次にツール名やファイル名などのより具体的な情報を付加してゆくことによって実行可能なスクリプトを得ることができる．抽象度の低いスクリプトは抽象度の高いスクリプトに記述された性質を保存している．

ここでは，図4.1に示された詳細化の方法の概要を述べ，次節以降でこの方法の適用例を示しながら各ステップについて説明を加えて行く．この方法は，開発過程記述の対象として仮定した環境のもとで大変容易に適用できる．

この方法では，最初に開発過程の流れと生成物の流れ（以下ではそれぞれ，プロセスフロー，プロダクトフローと呼ぶ）に対する，２つのスクリプトが必要である．ステップ１では，開発過程の実行の順序，およびそのための条件を明確にすることによってプロセスフローを詳細化する．スクリプトは各開発過程の前提条件と完了条件，およびこれらの条件が満たされなかった場合に実行（即ち，後戻り）すべき開発過程の名前を含むように記述される．この記述方法は，第２章で述べた性質記述の考え方に基づくものである．

ステップ２では，各生成物にオブジェクトを割り当てることによってプロダクトフローを詳細化する．ただしオブジェクトとは，ファイル，文書などの具体的な生成物であるとする．

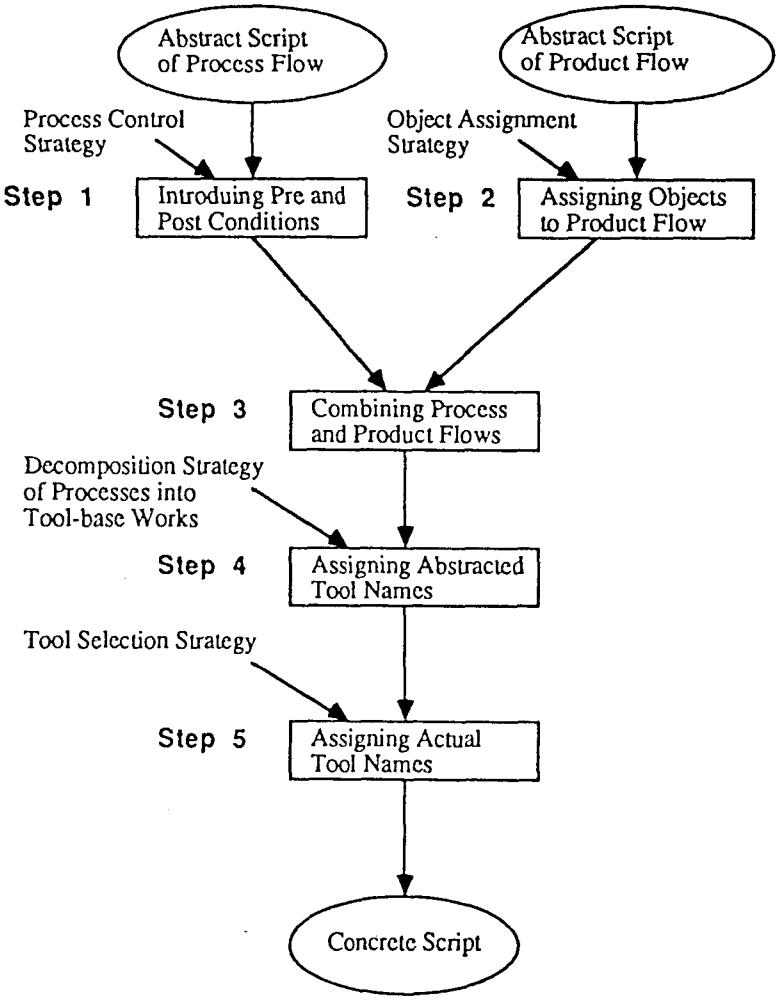



図4.1 スクリプトの段階的詳細化

ステップ３では，ステップ１とステップ２の結果が結合される．ステップ４では，各開発過程に対して抽象的なツール名を選択する．選択するツール名は実際のツール名ではなく，その開発過程に求められている機能を表す，計算機に依存しない名前である．

ステップ５では，特定の計算機で利用可能な実際のツール名とその起動に必要な引数を抽象的なツール名に対応付ける．結果として得られるスクリプトは，実際のツールの起動とその引数のファイル名など，および，それらのツール起動の前提条件，完了条件，後戻り先の開発過程などを定義している．

この段階的詳細化による方法は，さまざまなソフトウェア開発過程のためのさまざまなスクリプトを作成するために利用できる．

すでに存在する具体的なスクリプトの変更は，適切な抽象度のレベルに戻ることによって行うことができる．例えば，利用している具体的なツールの名前を変更するには，ステップ５へ戻ればよい．また，開発過程の進め方を変更するためには，ステップ１へ戻ればよい．

なお，以下のＰＤＬのスクリプトでは，各関数の形式的な意味を明確に記述するため，各関数定義ごとに，引数と関数の値のデータ型を次のように宣言する．

```
f : type1, type2, ..., typen -> typef ;
```

ここで，type1, type2, ..., typen は引数の型，typef は関数の値の型である．関数ｆとｇが同じ型を持っている場合，これらは次のように宣言できる．

```
f, g : type1, type2, ..., typen -> typef ;
```

関数の型が明白な場合には，型宣言を省略することもできる．

## 2.2 プロセスフローの抽象的なスクリプト

図4.2のようなソフトウェア開発過程の系列を考える．この図は開発過程間の基本的な流れを決定している．この流れは図4.3のようにＰＤＬで記述される．各関数はシステム状態型の引数ひとつをとり，システム状態型を値として返す．関数Mainsは，関数 SPs，PDs，CDs，CPs，および DBs によって構成され，これらの表すすべての開発過程を実行する．

## 2.3 プロダクトフローの抽象的なスクリプト

図4.4に示すように，各開発過程によって生成された生成物は，他の開発過程に渡されていくと仮定する．それぞれの開発過程は，いくつかの生成物をいくつかの生成物に写像する関数であると見なすことができる．これらの関数は，添字 d 付きの名前で表すことにする．プロダクトフローは，図4.5のようにＰＤＬで記述できる．この記述例では，各開発過程はそれぞれ１つの生成物のみを生成しているが，いくつかの生成物からなるタプルを用いることによって，複数の生成物を生成するような開発過程を記述することもできる．各関数の入力，出力のデータ型についてもはじめに定義している．

## 2.4 ステップ１：前提条件，完了条件，および後戻りの導入

プロセスフローの抽象的なスクリプトを，前提条件，完了条件を加えることによって詳細化する．これらの条件が満たされなかった場合（即ち，後戻りが生じた場合）に実行すべき開発過程は図4.6に示すように定まっているとする．これらの開発過程の関係はＰＤＬで図4.7のように記述できる．図4.7では，前提条件，完了条件の判定のための関数以外，すべての関数がシステム状態をシステム状態に写像する．

関数 Before_sp は，仕様記述の段階を表す SP，および引き続くすべての開発過程（開発過程本体および条件判定を含む）を実行するための前提条件の判定を行う．また，関数 After_sp は SP の完

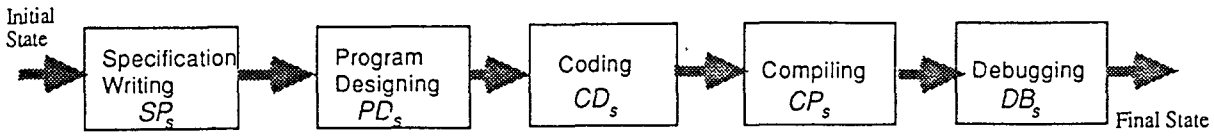



図4.2 ソフトウェア開発過程の流れ

```
Mains, SPs, PDs, CDs, CPs, DBs: state -> state ;
Mains(S) == DBs( CPs( CDs( PDs( SPs(S))))) ;
```

図4.3 プロセスフローの定義

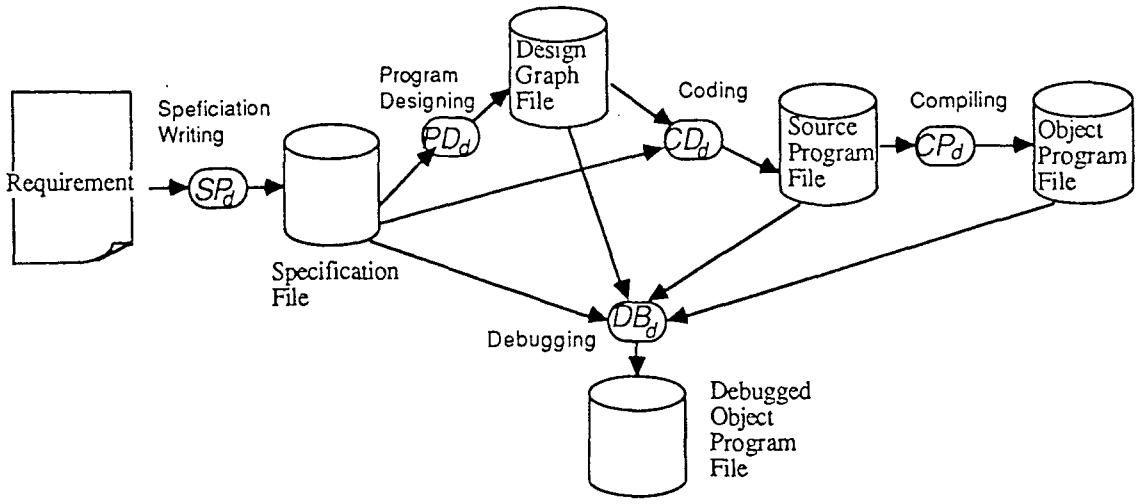



図4.4 生成物の依存関係

```
Maind : requirement -> object_program ;
SPd   : requirement -> specification ;
PDd   : specification -> design_graph ;
CDd   : design_graph, specification -> source_program ;
CPd   : source_program -> object_program ;
DBd   : design_graph, specification, object_program,
                source_program -> debugged_object_program ;

Maind(requirement) ==
        DBd(  PDd( SPd(requirement)) ,
              SPd(requirement) ,
              CPd( CDd( PDd( SPd(requirement)),
                        SPd(requirement)) ),
              CDd( PDd( SPd (requirement)),
                        SPd(requirement) ) ) ;
```

図4.5 プロダクトフローの定義

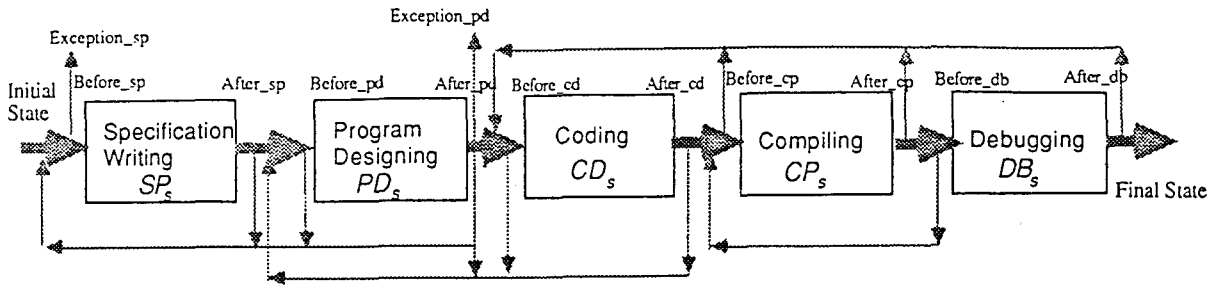



図4.6 後戻りを付加したプロセスフロー

```
Main,
Before_sp, After_sp,
Before_pd, After_pd,
Before_cd, After_cd,
Before_cp, After_cp,
Before_db, After_db :  state -> state ;
precond_sp, postcond_sp,
precond_pd, postcond_pd,
Back_to_sp, Back_to_pd,
precond_cd, postcond_cd,
precond_cp, postcond_cp,
precond_db, postcond_db :  state -> boolean ;

Main(S) == Before_sp(S) ;
Before_sp(S) ==
        if precond_sp(S) then After_sp(SP(S))
                         else Exception_sp(S) ;
After_sp(S) ==
        if postcond_sp(S) then Before_pd(S)
                          else Before_sp(S) ;
Before_pd(S) ==
        if precond_pd(S) then After_pd(PD(S))
                         else Before_sp(S) ;
After_pd(S) ==
        if postcond_pd(S) then Before_cd(S)
        else if  Back_to_sp(S) then  Before_sp(S)
        else if  Back_to_pd(S) then  Before_pd(S)
        else Exception_pd(S)  ;
Before_cd(S) == ......
        ...

After_db(S) ==
        if postcond_db(S) then S
                          else Before_cd(S) ;
```

図4.7 後戻りを付加したプロセスフローの定義

了条件および引き続くすべての開発過程を実行するための条件の判定を行う. Before_ および After_ という名前を持つ他の関数も同様の意味を持っている. これらの前提条件, 完了条件は, さらに他の条件によって詳細化されるか, あるいは未定義のまま残される. 実行時にシステムがスクリプト中の未定義関数を見つけた場合, システムはユーザに対して, その条件が満たされているかどうかを尋ねる.

各開発過程の本体は SP のような関数によって表されるが, これらは上で述べた関数 SPd や SPs とは異なる. すべての前提条件, 完了条件が満たされた場合には, 図4.7のスクリプトは図4.3で示したプロセスフローのスクリプトと同じになることがわかる.

## 2.5 ステップ2: プロダクトフローへのオブジェクトの割り当て

まず, マクロ記法を用いて, 図4.5で示したプロダクトフローのスクリプトを記述し直す. 各関数値（各関数の出力）に対してマクロ名をあてはめ, 次に, できる限りこれらの名前を使って図4.5のスクリプトを書き換える. 結果は図4.8に示すようになる. 出力に対するこれらのマクロ記法は, 最初のステップで図4.5のスクリプトの代わりに直接記述することもできる. その場合にはこのような書き換えは省略できる.

次に, 各入力にマクロ名をあてはめる. 関数 Fd の i 番目の入力には Fd_ini というマクロ名をあてはめることにする. 出力 Gd_out1 が入力 Fd_ini と Hd_inj に渡されている場合には, それらはマクロを用いて同じオブジェクト obj に割り当てられる.

```
#let    Gd_out1 : obj ;
#let    Fd_ini : obj ;
#let    Hd_inj : obj ;
```

これらのマクロは, 以下のような記法で簡略化できる.

```
#let  sp_out1 :  SPd(requirement)
#let  pd_out1 :  PDd(sp_out1)
#let  cd_out1 :  CDd(pd_out1, sp_out1)
#let  cp_out1 :  CPd(cd_out1)
#let  db_out1 :  DBd(pd_out1, sp_out1, cp_out1, cd_out1)

Maind(requirement) == db_out1 ;
```

図4.8 マクロによるプロダクトフローの定義

```
#let  requirement sp_in1 :                    file1
#let  sp_out1 pd_in1 cd_in2 db_in2 :          file2
#let  pd_out1 cd_in1 db_in1 :                 file3
#let  cd_out1 cp_in1 db_in4 :                 file4
#let  cp_out1 db_in3 :                        file5
#let  db_out1 :                               file6
```

図4.9 入出力生成物のファイルへの割り当て

```
SP´, PD´,CP´ : string, string, state -> state ;
CD´   : string, string, string, state -> state ;
DB´   : string, string, string, string, state -> state ;

SP(S) == SP´(sp_in1, sp_out1, S) ;
PD(S) == PD´(pd_in1, pd_out1, S) ;
CD(S) == CD´(cd_in1, cd_in2, cd_out1, S) ;
CP(S) == CP´(cp_in1, cp_out1, S) ;
DB(S) == DB´(db_in1, db_in2, db_in3, db_in4, db_out1, S) ;
```

図4.10 結合された関数の定義

```
#let    Gd_out1  Fd_ini  Hd_inj  : obj ;
```

この例では，各オブジェクトはファイル名である．すべての入力，出力のファイルへの割り当ては図4.9に示す通りである．

## 2.6 ステップ3：プロセスフローとプロダクトフローの結合

ステップ1で得られた，開発過程の本体を表す各関数に入出力オブジェクト名を加え，各開発過程本体を表す新しい関数名を導入する．図4.10ではこれらの新しい関数名には“ ’ ”を付けて表す．これらの関数によって，プロセスフローとプロダクトフローが結合される．

## 2.7 ステップ4：抽象的なツール名の割り当て

我々は，各開発過程はソフトウェア開発者がツールの助けをかりながら実行するものであると仮定している．このステップでは，実際に使用されるツールは選択しないが，ツールの機能を表す抽象的なツール名を導入する．

ツールの起動は以下のように記述する．

```
tool_name( parameter_list, state)
```

ここで，tool_name は次のような写像を与える関数である．

```
tool_name :  string, state  ->  state ;
```

string はツールに対する引数の列であり，state はハードウェア，ソフトウェアの状態を表すシステム状態である．

これらの関数のより具体的な定義には，第3章で述べたようにツール起動のための組込関数 exec を利用すればよい．ただし，ここでは exec を用いた実現までは考えないこととする．

```
display, edit, graph, debug :
        string, state -> state;
compile, copy :
        string, string, state -> state;

SP'(in, out, S) == display(in, S) @ edit(out, S) ;
PD'(in, out, S) == display(in, S) @ graph(out, S) ;
CD'(in1, in2, out, S) ==
        display(in1, S)
        @ graph(in2, S)
        @ edit(out, S) ;
CP'(in, out, S) == compile(in, out, S) ;
DB'(in1, in2, in3, in4, out, S) ==
        graph(in1, S)
        @ display(in2, S)
        @ debug(in3, S)
        @ display(in4, S)
        @ copy(in3, out, S) ;
```

図4.11 抽象的なツール名の割り当て

```
less, vi, key3, dbx : string, state -> state;
cc, cp : string, string, state -> state;
display(x, S) ==  less(x, S) ;
edit(x, S)    ==  vi(x, S) ;
graph(x, S)   ==  key3(x, S) ;
                  /* key3 is a graphic editor. */
compile(x, y, S) ==  cc(x+" -o "+y+" -g ", S) ;
                  /* Generate debug inf. */
debug(x, S)   ==  dbx(x, S) ;
copy(x, y, S) ==  cp(x+" "+y, S) ;
                  /* copy x into y. */
```

図4.12 実際のツール名の割り当て

ひとつの開発過程で複数のツールを並列に使用するために，ツール起動の式を結合させる記法が用意されている．例えば，

```
tool1(para1, S)  @  tool2(para2, S)
```

という式は tool1 と tool2 が並列に起動されることを表している．この式全体は，tool1 と tool2 の２つの状態遷移の結果のシステム状態を返す．

この例では図4.11に示すような抽象的なツール名を用いた．display はテキストを表示するツール，edit はテキストエディタ，graph は図形エディタ，compile はコンパイラ，debug はデバッガ，copy はファイルの複製を作るツールをそれぞれ表している．

## 2.8 ステップ５：実際のツール名の割り当て

最後に，図4.12に示すように，抽象的なツール名のそれぞれに実際に用いるツールの名前を割り当てる．ここでは，display に less，edit に vi，graph に key3，debug に dbx，compile に cc，copy に cp というツールをそれぞれ選んだ．

## 2.9 具体的なスクリプト

最終的に得られたスクリプトは，上で述べたいくつかのプログラム片を集めたものになる．即ち，

(1) ステップ１で得られたプロセスフロー（図4.7），
(2) ステップ２で得られた，生成物をオブジェクトに割り当てるマクロ定義（図4.9），
(3) ステップ３で得られた，プロセスフローとプロダクトフローを結合させる関数の定義（図4.10），
(4) ステップ４で得られた抽象的なツール名の割り当て（図4.11），および

(5) ステップ5で得られた実際のツール名の割り当て（図4.12），

である．

これらのすべてのスクリプトを図4.13に示す．このスクリプトを実行すると，スクリプトで指定したツールが適切なファイルに対して自動的に起動される．実際のツールを起動する関数，例えばvi(x,s)はシステムに解釈され，シェルコマンド"vi x"に変換される．このコマンドは，現在のウィンドウで画面エディタviをファイルxに対して起動することを表している．

```
Main,
Before_sp, After_sp,
Before_pd, After_pd,
Before_cd, After_cd,
Before_cp, After_cp,
Before_db, After_db :  state -> state ;
precond_sp, postcond_sp,
precond_pd, postcond_pd,
Back_to_sp, Back_to_pd,
precond_cd, postcond_cd,
precond_cp, postcond_cp,
precond_db, postcond_db ;  state -> boolean ;

Main(S) == Before_sp(S) ;
Before_sp(S) ==
        if precond_sp(S) then After_sp(SP(S))
                         else Exception_sp(S) ;
After_sp(S) ==
        if postcond_sp(S) then Before_pd(S)
                          else Before_sp(S) ;
Before_pd(S) ==
        if precond_pd(S) then After_pd(PD(S))
                         else Before_sp(S) ;
After_pd(S) ==
        if postcond_pd(S) then Before_cd(S)
        else if Back_to_sp(S) then Before_sp(S)
        else if Back_to_pd(S) then Before_pd(S)
        else Exception_pd(S) ;
Before_cd(S) ==
        if precond_cd(S) then After_cd(CD(S))
                         else Before_pd(S) ;
After_cd(S) ==
        if postcond_cd(S) then Before_cp(S)
                          else Before_pd(S) ;
Before_cp(S) ==
        if precond_cp(S) then After_cp(CP(S))
                         else Before_cd(S) ;
After_cp(S) ==
        if postcond_cp(S) then Before_db(S)
                          else Before_cd(S) ;
Before_db(S) ==
        if precond_db(S)  then After_db(DB(S))
                          else Before_cp(S) ;
After_db(S) ==
        if postcond_db(S) then  S
                          else Before_cd(S) ;

#let  requirement  sp_in1 :           file1
#let  sp_out1 pd_in1 cd_in2 db_in2 :  file2
#let  pd_out1 cd_in1 db_in1 :         file3
#let  cd_out1 cp_in1 db_in4 :         file4
#let  cp_out1 db_in3 :                file5
#let  db_out1 :                       file6

SP', PD',CP' : string, string, state -> state ;
CD' : string, string, string, state -> state ;
DB' : string, string, string, string, state -> state ;

SP(S) == SP'(sp_in1, sp_out1, S) ;
PD(S) == PD'(pd_in1, pd_out1, S) ;
CD(S) == CD'(cd_in1, cd_in2, cd_out1, S) ;
CP(S) == CP'(cp_in1, cp_out1, S) ;
DB(S) == DB'(db_in1, db_in2, db_in3, db_in4, db_out1, S)


display, edit, graph, debug :
        string, state -> state;
compile, copy :
        string, string, state -> state ;

SP'(in, out, S) == display(in, S) @ edit(out, S) ;
PD'(in, out, S) == display(in, S) @ graph(out, S) ;
CD'(in1, in2, out, S) ==
        display(in1, S)
        @ graph(in2, S)
        @ edit(out, S);
CP'(in, out, S) == compile(in, out, S) ;
DB'(in1, in2, in3, in4, out, S) ==
        graph(in1, S)
        @ display(in2, S)
        @ debug(in3, S)
        @ display(in4, S)
        @ copy(in3, out, S) ;

less, vi, key3, dbx : string, state -> state;
cc, cp              : string, string, state -> state;
display(x, S) ==  less(x, S) ;
edit(x, S)    ==  vi(x, S) ;
graph(x, S)   ==  key3(x, S)  ;
                  /* key3 is a graphic editor. */
compile(x, y, S) == cc(x+" -o "+y+" -g ", S) ;
                    /* Generate debug inf. */
debug(x, S)   ==  dbx(x, S)       ;
copy(x, y, S) ==  cp(x+" "+y, S)  ;
                  /* copy x into y. */
```

図4.13 得られたスクリプト

# 第3節　適合可能なソフトウェア開発環境

## 3.1 ソフトウェア開発環境の適合

ＳＤＥ（ソフトウェア開発環境）とは，ソフトウェア開発の各段階を含むすべての行動を高め，あるいは自動化する環境のことである[Dart 87]．これは計算機やネットワークのようなハードウェア，オペレーティングシステムやツールのようなソフトウェアなど，ソフトウェア開発者の行動に関係するすべてのものを含んでいる．ソフトウェア開発者は，特定されたひとつのＳＤＥ，あるいはいくつかの異なるＳＤＥのもとでソフトウェア生成物を作成する．技術の進展とユーザの期待の増大に伴い，多数のＳＤＥが提案され，また実際に実現されている[Dart 87][Kishida 88]．

これらのＳＤＥは，特定のプログラミング言語，プログラム構造，ツール，あるいは開発技法に直接依存しており[Dart 87]，変更や改編の可能性は一般には限定されている．しかし，ソフトウェア開発者はさまざまな理由から，自分たちの使用しているＳＤＥを変更したいと考えるであろう．

例えば，開発作業が異なれば異なった支援ツール集合が必要であり，開発者の習慣が異なれば異なったツールの機能を求めるであろう．このような差異は非常に広範囲にわたるため，すべてのツールやすべての習慣をそれだけで支援できるようなただひとつのシステムを構築しようとするのは非現実的である．それよりも，特定の開発者と特定の開発作業に応じて，開発者の習慣に合うようなカスタマイズや，効率向上のための調整，新しいツールを加えるような拡張，ハードウェアやシステムソフトウェアの変更への対応が可能なシステムを提供する方がより効果的である[Riddle 87]．

このような，あるシステムから特定のシステムを作り出す操作を適合(adaptation)と呼ぶ．適合は，ＳＤＥからＳＤＥへの写像として形式的に定義される．適合の操作対象となるＳＤＥを適合可能な

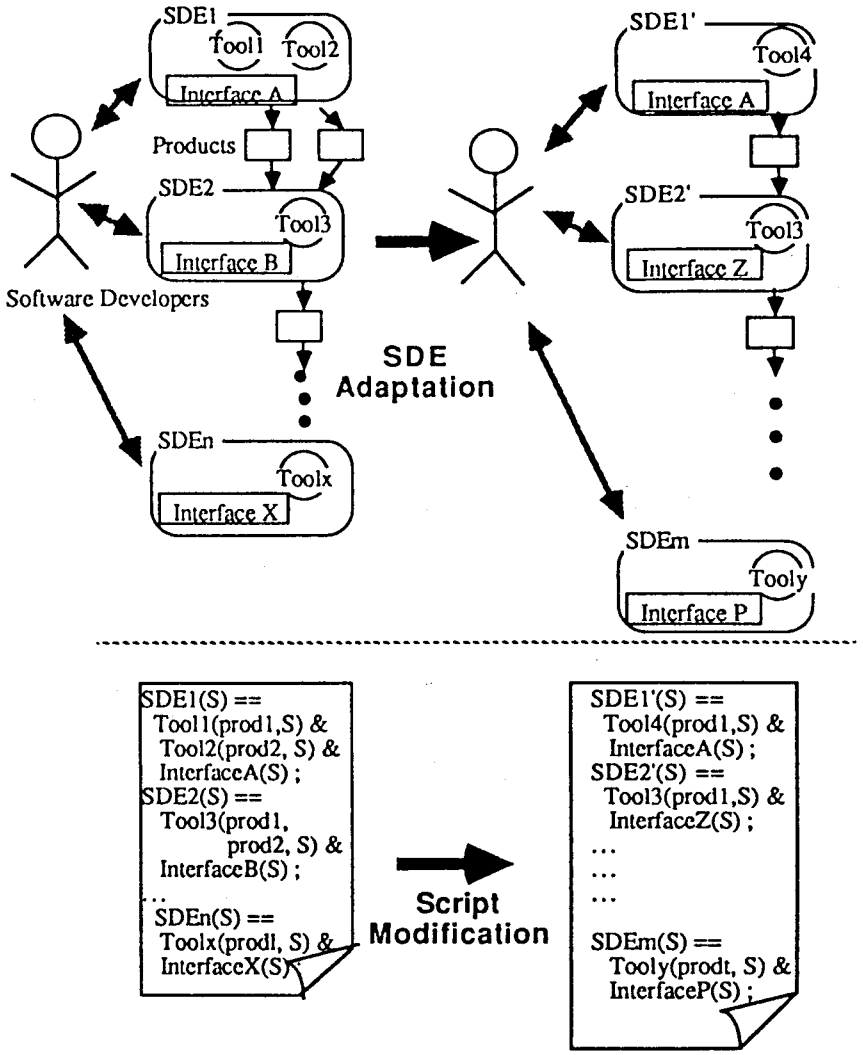



図4.14 スクリプトの変更とＳＤＥの適合

ＳＤＥ，適合の結果として得られるＳＤＥを特定化ＳＤＥと呼ぶ．

適合可能なＳＤＥシステムは，上で述べたようなさまざまな適合を支援し，開発者に特定化したＳＤＥを提供する．適合可能なＳＤＥシステムを実現するために，ＳＤＥは何らかの方法で定義されていなければならない．直接的な方法のひとつは，形式的な言語を用いて，特定化ＳＤＥをその言語のスクリプト（プログラム）で表現することである．スクリプトを変更することにより，異なったＳＤＥを得ることができる．図4.14に示すように，ＳＤＥの適合はこのようなスクリプトの変更に対応している．

## 3.2 適合可能なソフトウェア開発環境のアーキテクチャ

この節では，ＰＤＬによって実現される開発環境の拡張である，適合可能なＳＤＥシステムのアーキテクチャ設計について論じる．

適合可能なＳＤＥは2つの機能を提供する．第一はスクリプト実行と呼ばれ，ＳＤＥを表現したスクリプトを実行し，特定化ＳＤＥを開発者のために実現することである．もうひとつはスクリプト適合と呼ばれ，新しいスクリプトの生成，すでに存在するスクリプトの変更，および型紙となるスクリプトから新しいスクリプトを作成することである．

さまざまな種類のアーキテクチャを考えることができるが，ＳＤＥの機能の直接的な実現のひとつを図4.15に示す．このアーキテクチャの個々の構成要素について説明する．

(1) スクリプト実行機能のために，スクリプト実行マネージャを用意した．これは特定化ＳＤＥの定義の中の実行可能なスクリプトを解釈し，ソフトウェア開発者のために適合可能なＳＤＥを実現する．スクリプト実行機能に加え，関数やマクロの定義の変更や，実行のトレース，未定義関数への警告など，デバッグ用機能も提供している．この警告機能の拡張として，ソフトウェア開発者は実行中に未定義関数の値を決定することができる．この機能によ

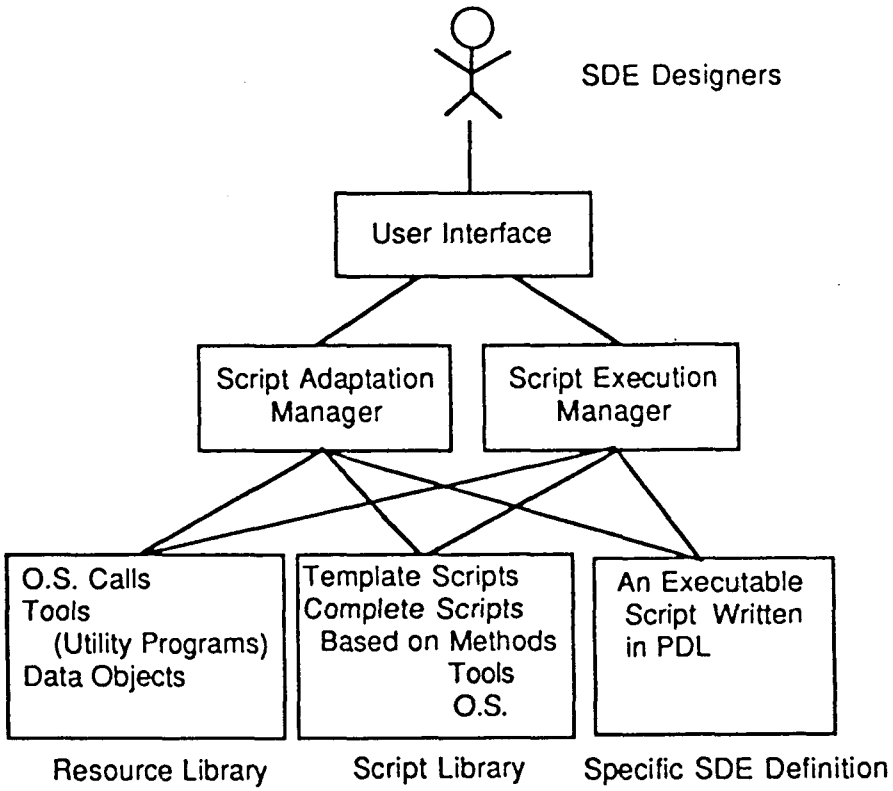



図4.15 適合可能なＳＤＥのアーキテクチャ

って，すべての関数の具体的な定義が与えられていないスクリプトであっても実行を行うことができ，大変有用である．

(2) スクリプト適合機能のために，スクリプト適合マネージャを用意した．これはスクリプトライブラリの中のさまざまな種類のスクリプトから，必要なスクリプトを探し出す．また，段階的詳細化によって実行可能なスクリプトを容易に得られるように，それらの編集や結合を効果的に支援する．得られたスクリプトは特定化ＳＤＥ定義の中に保存される．

(3) リソースライブラリ

ハードウェア，ソフトウェア機能の利用を進めるために，リソースライブラリを用意する．ＯＳコール，ツール，およびデータオブジェクト（ファイル）が操作される．このライブラリへの要求は一般にシステムコールやコマンドとして実現される．

(4) スクリプトライブラリは，ＰＤＬで記述されたさまざまなスクリプトの集合である．ライブラリには，関数あるいはマクロ定義の付加によって実行可能なスクリプトとなる，型紙にあたる不完全な記述も含まれていてよい．ライブラリはまた，特定の開発技法やツール，あるいはＯＳに基づいたさまざまな実行可能スクリプトを含む．このライブラリ内のスクリプトには，スクリプト適合マネージャによって実体化，結合，あるいは変更されて実行可能なスクリプトとして構成されるものも含まれている．

(5) 特定化ＳＤＥ定義は，スクリプト実行マネージャによって実行することができる，ＰＤＬの完全なスクリプトである．ソフトウェア開発者は，この特定化ＳＤＥ定義の中のスクリプトによって決定されるそれぞれのＳＤＥのもとで作業を行う．

(6) ユーザインタフェースは，システムとＳＤＥ設計者の間に視覚的なインタフェースを提供する．このシステムのユーザとして，２種類のユーザを考えることができる．一方のユーザはＳＤＥ設計者（あるいは適合可能ＳＤＥのユーザ）である．彼らはスクリプト適合マネージャを使用してしてＳＤＥを設計する．もう一方のユーザは，このインタフェースを使用せずに，特定化ＳＤＥ定義のスクリプトによって定義されたインタフェースを用いるソフトウェア開発者である．

## 第4節　結言

抽象度の高いスクリプトから，プロセスフローとプロダクトフローを記述することによって実行可能なスクリプトへ詳細化する方法について論じた．この詳細化の方法は大変簡潔かつ直接的であるため，さまざまな開発過程のさまざまなスクリプトに適用可能である．この段階的詳細化法は実行可能なＰＤＬスクリプトを得るための方法のひとつである．ほかにもさまざまな詳細化技法を考えることができるが，この方法は開発過程記述の対象として仮定した環境のもとで，大変直接的に行うことができる．段階的な詳細化による情報の付加は，基本的にはすでに存在しているスクリプトに変更を加えることなく，単に新しい関数定義やマクロ定義を加えてゆくだけで行うことができる．このため，この方法はＰＤＬのような関数型の言語に適している．

これに対し，開発過程記述のために手続き的言語を選択した場合には，抽象度の高いスクリプトの意味がより複雑になり，また，新たな情報を付加するために，すでに存在するスクリプトを変更しなければならなくなるなど，いくつかの困難な問題が生じる．

ある開発過程のために記述したＰＤＬスクリプトは，そのスクリプトを実行することによって正しさを示すことができ，また，開発過程の進行に伴ってツールやメッセージが適切に起動されるかを見ることができる．スクリプトの形式的な検証が必要な場合は，代数的仕様に対するさまざまな検証手法[Ehrig 85]をそのまま適用することができる．

# 第５章　開発過程の記述による支援システムの作成

## 第１節　序言

ソフトウェア開発過程全体の構成方法，各開発過程で行なう作業，生成物の種類や記法などは，作るべきソフトウェアの種類や規模，開発者の数や質によって大きく異なる．あるソフトウェアを正しく作成できた開発過程に従って，別のソフトウェアを開発できるとは限らない．また，人によっては一つの開発過程として捉える範囲や行なう作業，生成物の記法などが異なるであろう．

このような開発過程をできるだけ定型化して開発の効率を上げたり，作られるソフトウェアの品質を保つ試みがされている．例えば，ＪＳＤ（Jackson System Development）[Jackson 82][Cameron 86]と呼ばれるソフトウェア開発技法では，対象とする開発の範囲を要求分析，設計書作成として，開発過程の構成や各作業を比較的具体的に指定している．

しかしこれらのソフトウェア開発技法は，作業の進め方や考え方の方針を与えてはいるものの，どの開発過程でどのようなツールを用いるべきか，どのようになれば次の開発過程に進めばよいか，先に進めなくなったときにどの開発過程に戻り，やり直せばよいか，など，実際に開発を進める上での具体的な情報は示していないことが多い．

そこで，ソフトウェア開発過程記述とその実行支援系によって，使用すべきツールを自動的に起動したり，作業を進める上での判断基準を表示したりして，ソフトウェア開発技法に従った開発を効率よく行なえる支援システムを作成することを試みた．

本章では，開発過程の一例としてＪＳＤを選び，ＪＳＤの支援システムを作成する．このシステムはＪＳＤの各開発過程の作業に適したツールを自動的に起動したり，各生成物がＪＳＤの各作業段階

の求める条件を満たしているかどうかの判定基準を与えたりする．ただし，一人の開発者が一台のワークステーション上で全ての開発作業を行なうことを前提とする．

作成の方法としては，第４章で述べた段階的詳細化の手法を用いた．この方法はＪＳＤに対しての固有のものではなく，抽象的な説明や例を用いた開発過程の説明から，その開発過程の支援システムを作成するための一般的な方法であり，ＪＳＤ以外の開発支援システムの作成にも広く用いることができる．

支援システムの作成は以下のような手順で進めた．

まず，ＪＳＤを理解しやすい形に整理するために，各開発過程で参照，変更，作成する生成物が満たすべき条件（性質）を自然語で記述（性質記述と呼ぶ）する．次にそれを厳密に記述するために，開発過程記述言語ＰＤＬによる記述に変換した．得られたＰＤＬによる形式的な記述は，ＪＳＤに従った開発過程を厳密に定義しているが，使用するツールや，途中で矛盾が生じたりして先に進めなくなったときの対処方法などの情報は含んでいない．従って，ＰＤＬによる記述を順次詳細化し，含まれていなかった情報を付加してＪＳＤの支援システムのスクリプト（プログラム）を完成させた．この詳細化は，元の記述の機械的な置き換えや関数定義の追加だけで容易に行なえた．

この支援システムのスクリプトを実行すると，各開発過程の作業に適したツールが自動的に起動される．さらに，各生成物が満たすべき条件を機械的に判定するか，または開発者が判定を行ないやすいように判定基準を提示して開発者の判断の入力を受けることによって，次に行なうべき作業に自動的に移る．この支援システムを用いることによって，開発者はＪＳＤに従ったソフトウェア開発作業を比較的簡単に行なうことができる．

# 第2節　ＪＳＤの性質記述

## 2.1 ＪＳＤ（Jackson System Development）の概要

ＪＳＤ[Jackson 82][Cameron 86]は，要求分析，設計などを対象とするソフトウェア開発の方法である．

ＪＳＤでは，まず，作成しようとするシステムが対象とする実世界で起こる行動，出来事について抽象化した“モデル”を作成する．モデルは３つの開発過程に分けて作成し，それぞれ順に，実体行動ステップ(Entity/Action Step)，実体構造ステップ(Entity Structure Step)，初期モデルステップ(Initial Model Step)という．次に，システムに要求される出力のための機能をモデルに付加し，仕様を完成する（機能ステップ(Function Step)）．最後に，仕様をシステムの稼働する環境に合わせて変換し，実行可能なシステムを作成する（実現化ステップ(Implementation Step)）．図5.1にこの５つの開発過程の流れを示す．

ＪＳＤの文献には記述が例示的であるという問題点がある．文献[Jackson 82]では「倉庫会社問題」など３つの例題を通して，文献[Cameron 86]では「図書館問題」を通して説明を行なっており，これらの例から一般的な行為を類推させるような記述がほとんどである．また，開発過程の分割法が[Jackson 82]と[Cameron 86]では異なっている．例えば，[Jackson 82]では上で述べたように開発過程を分割しているが，[Cameron 86]ではモデルを作成する開発過程を一つの開発過程として，全体を３つの開発過程に分割している．本章では，基本として[Jackson 82]に基づいて理解したＪＳＤの記述を行なっている．

## 2.2 ＪＳＤの支援システム作成の手順

以下の手順に基づいてＪＳＤの記述を行ない，支援システムを作成した[稲田88]．

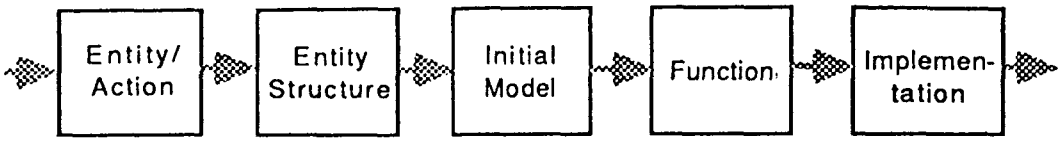



図5.1　ＪＳＤの開発過程

表5.1 自然語によるＪＳＤの性質記述

---

＜開発過程の順序関係＞
<<ＪＳＤ>>
・ＪＳＤは５つの開発過程（実体行動ステップ，実体構造ステップ，初期モデルステップ，機能ステップ，実現化ステップ）を逐次的に実行する．
＜前提条件＞
<<実体行動ステップ>>
・要求仕様書にシステムが対象とする世界に関する記述がある．
<<実体構造ステップ>>
・・・
＜完了条件＞
<<実体行動ステップ>>
・実体リストが正しく作成されている．
・行動リストが正しく作成されている．
・・・
＜生成物の受け渡し関係＞
・ＪＳＤへの入力は，実体行動ステップへの入力である．
・ＪＳＤの出力は，実現化ステップの出力である．
・実体行動ステップの出力は，実体構造ステップへの入力である．
・・・

---

① 文献の例示的な記述より理解されたＪＳＤを整理するために，各生成物の満たすべき条件等の記述（性質記述と呼ぶ）を自然語で行なう．

② この記述をＪＳＤに従った開発の支援システムの仕様とみなした．ただし，支援の対象を一人の開発者によるワークステーション上での開発とした．支援システムを作成するために，自然語による記述を関数型言語ＰＤＬによる記述に変換する．

③ 得られた記述には，多くの「未定義関数」（その記述レベルで実行可能な定義の与えられていない関数）が含まれている．また，矛盾が生じたりして先に進めなくなったときの適切なプロセスへの後戻り（以降単に後戻りという）や具体的なツール名などの情報が含まれていない．従って，3.2 で述べる４段階の詳細化によって情報を付加し，関数定義やマクロ定義を行ない，支援システムのスクリプトを得る．

④ 得られたスクリプトを，ＰＤＬの処理系であるＰＤＬインタプリタ[荻原88]によって解釈，実行する．これによって，ＪＳＤの支援システムが作成される．

## 2.3 自然語による性質記述

本章での性質記述とは，ソフトウェア開発の各開発過程で参照，変更，作成される生成物の性質についての記述であり，具体的には各開発過程に対して以下の４項目について行なった記述である．

① 開発過程の順序関係．さらにいくつかに分かれた開発過程の逐次的な実行順序．

② 前提条件．開発過程での作業で参照，変更される生成物（入力生成物と呼ぶ）が満たしていなければならない条件．

③ 完了条件．開発過程での作業で作成，変更された生成物（出力生成物と呼ぶ）が満たしていなければならない条件．

④ 生成物の受け渡し関係．いくつかに分かれた開発過程間での入

出力生成物の受け渡し関係.

ここでは，例を用いずに，具体的に各開発過程の入出力生成物がどのような条件を満たしていなければならないかを自然語で記述した．このとき，必要以上に詳しい情報を削ったり，例から推測できる情報を追加して，記述の抽象度をそろえた.

自然語による性質記述の一部を表5.1に示す．例えば，＜開発過程の順序関係＞においては，ＪＳＤは５つの開発過程を逐次的に実行することを記述している.

ＪＳＤの開発過程の一つである実体行動ステップの＜前提条件＞は「システムが対象とする世界に関する記述がある」が満たされていなければならないことを示す．＜完了条件＞についても同様である.

＜生成物の受け渡し関係＞では，各開発過程の入力生成物がどの開発過程からの出力生成物であるかを記述する.

## 第3節　ＰＤＬによる支援システムの作成と実行

### 3.1 ＰＤＬ記述への変換

第２節で得られたＪＳＤの性質記述を，ＪＳＤの支援システムの仕様とみなして，支援システムを作成した．まず，自然語による性質記述をＰＤＬによる記述（図5.2）に変換した．変換方法としては，

①＜開発過程＞の各開発過程（例 P1）を前提条件の判定部（PreP1）と実際に作業を行なう本体（P1body），完了条件の判定部（PostP1）に分け，前提条件の判定部，本体，完了条件の判定部を順に実行するように記述する．例えば，P1 を行なった後に P2 を行なう関数Ｆは，

```
F(S) == PreP1(S);
PreP1(S)  == PostP1(P1body(S));
PostP1(S) == PreP2(S);
PreP2(S)  == ・・・;
・・・
```

のように記述する．

②＜前提条件＞，＜完了条件＞の各条件を，引数をシステム状態とする述語として記述する．述語とは，真偽値を返す関数である．システム状態は Ｓ で表すことにする．
例えば，自然語による開発過程P1の前提条件の記述が「生成物１が正しい」と「生成物２が正しい」であればＰＤＬでは，

```
P1PreCond(S) == 生成物１が正しい(S)
                & 生成物２が正しい(S);
```

```
JSD(S)              == PreEA(S);
PreEA(S)            == PostEA(EAbody(S));          ①
PostEA(S)           == PreES(S);
   ...
EAPreCond(S)        == システムが対象とする世界に
                       関する記述がある？(S);      ②
   ...
#let EAin            : ESin2;
#let EAout           : ESin1;                      ③
   ...
```

図5.2　自然語記述から変換した記述

のように記述する.

③＜生成物の受け渡し関係＞において，同一である出力生成物と入力生成物をマクロ定義で記述する．例えば，自然語による記述が「P2 への入力は，P1 からの出力である」であればＰＤＬでは，

```
#let P2in : P1out;
```

のように記述する.

### 3.2 スクリプトの段階的詳細化

自然語による性質記述から変換されたＰＤＬ記述には，

(1) 開発過程の逐次的な流れ，
(2) 生成物の流れ，
(3) 各前提条件，完了条件の情報.

がそれぞれ独立した関数定義として含まれている．例えば，後戻り（前提条件，完了条件が成り立たないときに起こる）や実際に使用すべきツールの情報などは含まれていない．これらの情報が含まれないために，ＰＤＬの記述には未定義関数が数多く存在する.

ここでは，以下に述べる手順（図5.3に示す）に従って情報を付加し，段階的に詳細化を行なった.

① 開発過程の後戻り付加.
② 生成物のファイルへの割り当て.
③ 開発過程本体の基本機能への分割.
④ 基本機能のツールへの割り当て.

ここで，開発過程本体は複数の基本機能から成る．基本機能とは，例えば編集やコンパイルなどをいい，一つ，または複数のツールの

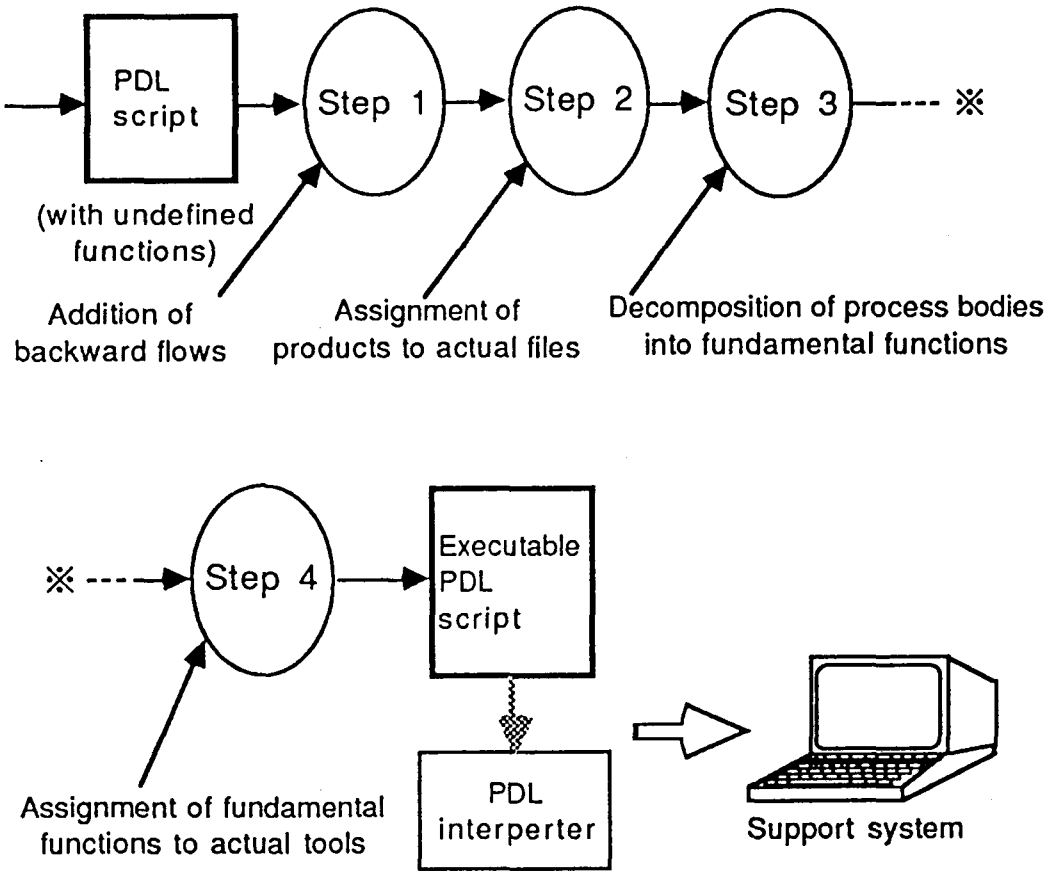

PDL
script
Step 1
Step 2
Step 3
※
(with undefined
functions)
Addition of
backward flows
Assignment of
products to actual files
Decomposition of process bodies
into fundamental functions
※
Step 4
Executable
PDL
script
PDL
interperter
Support system
Assignment of fundamental
functions to actual tools

図5.3 詳細化の手順

起動で置き換えられる．ツールの起動は，UNIX のコマンドを実行するＰＤＬの組み込み関数 exec で行なわれ，例えばエディタ vi などが起動される．

本章で示す詳細化では，自然語からの変換で得られた記述の機械的な置き換えや関数定義の追加だけで実行可能な記述を得ることができる．

次に，各手順の詳細について述べる．

① 開発過程の後戻り付加

まず，各開発過程での後戻りに関する情報を，最初のＰＤＬの記述に含まれる情報“開発過程の逐次的な流れ”に付加する．例えば P1 の前提条件判定部ならば，

```
PreP1(S)   ==   PostP1(P1body(S));
```

という記述を

```
PreP1(S)   ==   if P1PreCond(S)
                then PostP1(P1body(S))
                else PrePx(S);
```

に変換する（完了条件も同様）．ここで PrePx は条件が成り立たない場合に実行する開発過程 Px の入口である前提条件判定部である．この段階での情報付加後の記述を図5.4に示す．以降，各手順で新たに変更，追加された記述は，図において太字で表わす．

② 生成物のファイルへの割り当て

ここでは，最初のＰＤＬ記述で得られた生成物の流れを表わすマクロ定義を生成物間の同値関係とみなし，その最小の同値類分割を考える．そして，各同値類に対して適切な１つのファイルを割

```
JSD(S)              == PreEA(S);
PreEA(S)            == if EAPreCond(S)
                       then EAPost(EAbody(S))
                       else Exception(S);
PostEA(S)           == if EAPostCond(S)
                       ...;
   ...
#let EAin           : ESin2;
   ...
```

図5.4 後戻りを付加した開発過程の記述

```
#let  EAin ESin2 IMin2 FUin3 IPin4 : "Spec"
#let  EAout ESin1 : "EntAct"
   ...
JSD(S)              == PreEA(S);
PreEA(S)            == if EAPreCond(S)
                       then EAPost(EAbody(S))
                       else Exception(S);
   ...
```

図5.5 生成物のファイルへの割り当て

り当てる．この割り当ては，マクロ文

```
#let 生成物1・・・生成物n : "ファイル名"
```

として行なう（図5.5）．ここで生成物1，・・・，生成物nは同一の同値類に属する生成物である．また，このマクロ文はn個のマクロ文

```
#let 生成物1 : "ファイル名"
  ...
#let 生成物n : "ファイル名"
```

の略記法である．

③ 開発過程本体の基本機能への分割

ここでは，各開発過程の本体（関数 EAbody 等）をいくつかの基本機能に分割する．すなわち，各本体を編集やコンパイル等の，ツールを用いて行なう仕事に分割する．開発過程 Px の本体 Pxbody は一般に，

```
Pxbody(S) == 基本機能1(S)
             @ 基本機能2(S)
             @ ...
             @ 基本機能n(S);
```

と表わす（図5.6）．“@”は前後の関数が並列に評価されることを表わす[荻原89]．

④ 基本機能のツールへの割り当て

ここでは，③で分割した基本機能，述語にツールを割り当てる．

```
#let  EAin ESin2 IMin2 FUin3 IPin4 : "Spec"
  ...
JSD(S)              == PreEA(S);
PreEA(S)            == if EAPreCond(S)
                       ...;
  ...
EAPreCond(S)        == Exist(EAin,S) &
                       SatisfyEASpec(EAin,S);
EAbody(S)           == Dispaly(EAin,S) @ Editor(EAout,S);
EAPostCond(S)       == ...;
  ...
```

図5.6 基本機能への分割

```
#let  EAin ESin2 IMin2 FUin3 IPin4 : "Spec"
  ...
JSD(S)              == PreEA(S);
PreEA(S)            == if EAPreCond(S)
                       ...;
EAPreCond(S)        == Exist(EAin,S) &
                       SatisfyEASpec(EAin,S);
EAbody(S)           == Dispaly(EAin,S) @ Editor(EAout,S);
  ...
Display(file,S)     == wclose(exec("less "+file,wopen(S)));
Editor(file,S)      == wclose(exec("vi "+file,wopen(S)));
Exist(file,S)       == exec("test -f "+file,S);
SatisfyEASpec(file,S)
                    == if read(write(Massage,S)) = "yes"
                       then TRUE
                       else FALSE;
```

図5.7 基本機能のツールへの割り当て

すなわち，具体的なツールを起動する組み込み関数や，その他の組み込み関数を割り当てて，未定義関数の存在しない記述を完成する（図5.7）．記述の形は，ツールの起動については，

```
基本機能 i (S) ==
    wclose(
        exec("ツール i1", exec("ツール i2",
            ・・・wopen(S)・・・)));
```

のように表わされる．ここで，wopen，wclose，execはそれぞれウィンドウを開く，閉じる，またツールを起動する組み込み関数である．述語については，ツールを用いることにより自動的に判定できる（例，図5.7 関数 Exist）ものもあるが，そのような述語ばかりではない．自動的に判定できないときには，人間が条件判定を行ない，述語の値を入力するように記述（例，図5.7 関数 SatisfyEASpec）する．

以上の詳細化によって，未定義関数が存在しない記述を完成した．この記述の一部を図5.8に示す．

この記述を基にして，条件判定部において人間が判定を行なうときに判定のよりどころとなる生成物を表示し，ユーザはその内容を参照しながら判定を行なうことができるように変更した．この変更によってさらに使いやすい支援システムが作成された．

### 3.3 スクリプトの実行

3.2の手続きによって得られたスクリプトをＰＤＬインタプリタで実行することによって，ＪＳＤの支援システムが稼働される．

この支援システムを用いて，「倉庫会社問題」および「図書購入業務問題」[梶谷84]を解いた．ここでは，前者の実行例の一部を示す．まず実体行動ステップ（EA）の前提条件判定部（PreEA）が実行

```
//JSD Script
//Definition of Products(Macro)
#let  EAin ESin2 IMin2 FUin3 IPin4 : "Spec"
#let  EAout ESin1 : "EntAct"
  ...
//JSD(Jackson System Development)
JSD(S)              == PreEA(S);
//EA(Entity/Action Step)
PreEA(S)            == if EAPreCond(S)
                       then PostEA(EAbody(S))
                       else Exception(S);
PostEA(S)           == if EAPostCond(S)
                       then PreES(S)
                       else PreEA(S);
//ES(Entity Structure Step)
   ...
//Process Body
EAbody(S)           == Display(EAin,S) @ Editor(EAout,S);
   ...
//EAPreCond & EAPostCond
EAPreCond(S)        == Exist(EAin,S) &
                       SatisfyEASpec(EAin,S);
EAPostCond(S)       == EAEntAct(EAin,EAout,S);
//ESPreCond & ESPostCond
   ...
//Tool Assignment
Display(file,S)     == wcolse(exec("less "+file,wopen(S)));
Editor(file,S)      == wclose(exec("vi "+file,wopen(S)));
Exist(file,S)       == status(exec("test -f "+file,S));
SatisfyEASpec(file,S)
                    == if read(write(Message,S)) = "yes"
                       then TRUE
                       else FALSE;
   ...
```

図5.8　ＰＤＬによるＪＳＤの実行可能な記述（一部）

される．ここでは，ツールによって，仕様書（ファイル Spec）が存在するかを自動的に判定（図5.8 関数 Exist）し，さらに仕様書に記述されるべき事が記述されているかを開発者に尋ね，開発者から真か偽かの値を受ける（図5.8 関数 SatisfyEASpec）．仕様書が存在し，正しく記述されていれば，EA の本体（EAbody）が実行される．仕様書が存在しないか，存在するが正しく記述されていなければ，例外処理（Exception）として終了する．

EAbody では，２つのウィンドウを開く．開発者は，一方のウィンドウで起動された UNIX のツール less で表示された仕様書を参照しながら，もう一方のウィンドウで起動されるツール vi を用いて，仕様書中のキーワードを表として記述する実体行動リスト（ファイル EntAct）を編集する．

EAbodyが終了すると，EAの完了条件判定部(PostEA)が実行される（図5.9に示す）．PostEA では，作成した実体行動リストを表示しそれが正しいかの判定を開発者に尋ねる．開発者が正しいと判定すれば，次の開発過程である実体構造ステップ（ES）の前提条件判定部（PreES）を評価し，さらに ES の本体（ESbody）が実行される．正しく作成されていなければ，PreEA に戻り，再び EA を行なう．

開発者は，以上で述べたような対話的な作業を繰り返し，目的とするソフトウェアを作成する．

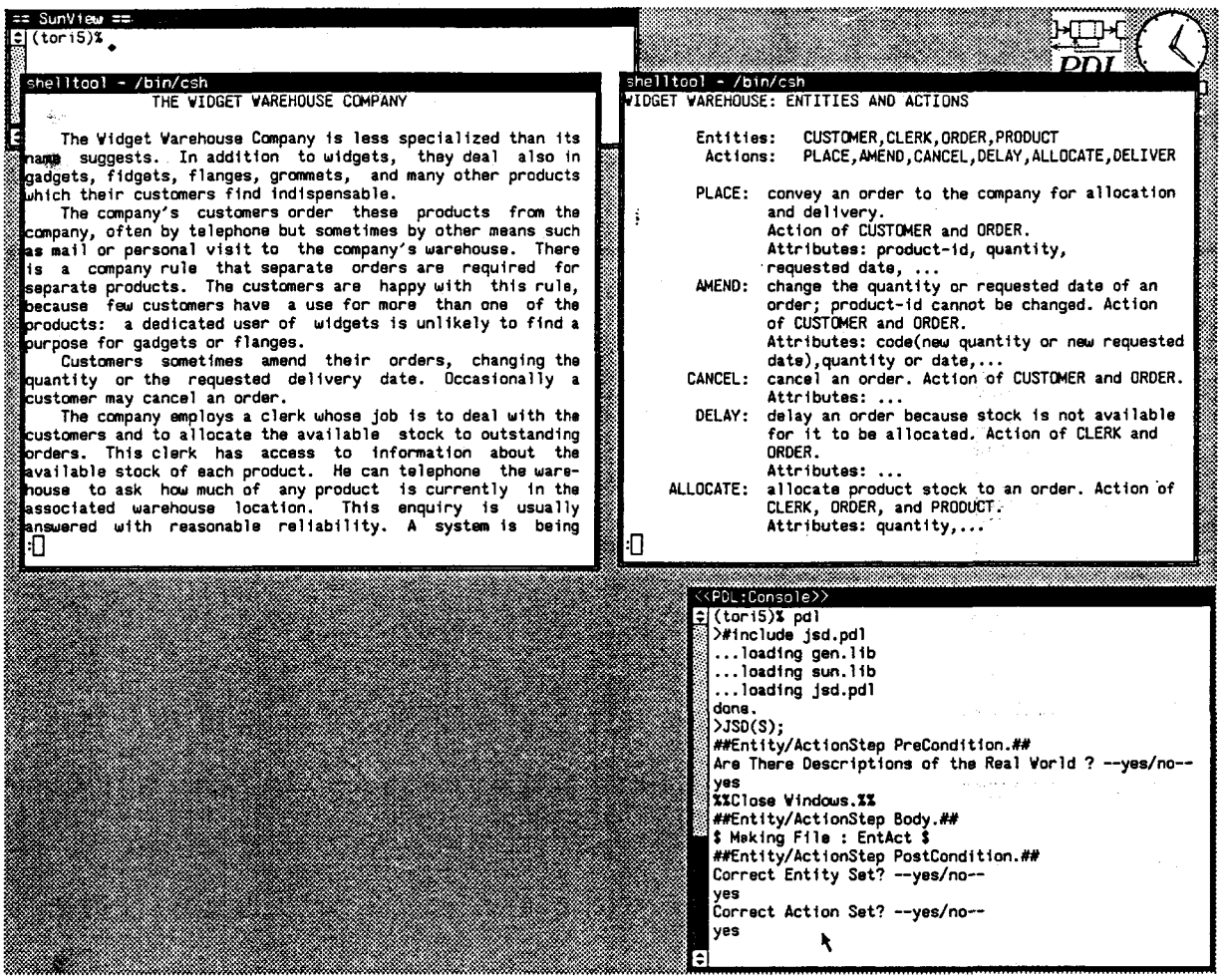



図5.9 得られたスクリプトの実行

## 第4節　検討

(1) 性質記述について

スクリプトの開発の最初の段階で行なった性質記述は，開発過程の動作的記述モデル[Wiliam 88]に基づいている．性質記述を行なうことによって，ＪＳＤの各開発過程で作成する生成物の性質や各開発過程の入出力生成物の関係を明らかにすることができた．ただし，ＪＳＤの文献の例示的な記述から一般的な性質記述を得るのは，かなり困難な作業であった[野村88][井上88].

また，自然語による性質記述は，比較的容易にＰＤＬの関数の記述に変換することができた.

(2) 段階的詳細化について

自然語による性質記述を変換して得られたＰＤＬの記述に対し，詳細化の各段階では，すでにある記述を機械的に置き換えるか，新たな関数定義を追加するだけで完全なＰＤＬスクリプトを得ることができた.

本章で行なった段階的詳細化は，開発過程の形式的な記述から実行可能なスクリプトを得るための有効な一手法となる．さらに性質記述が行なえる他の開発過程についても，今回示した方法によって支援システムを作成することができるであろう.

自然語記述から変換したＰＤＬによる記述のサイズは約50行であったが，段階的詳細化を行なうことによって得られた記述のサイズは約460行であった.

(3) 支援システムについて

作成したシステムを用いることによって，紙などのドキュメントを用いてＪＳＤを手作業で行なう場合に比べ，生成物の修正や再利用等の管理が容易になった.

ＪＳＤの各作業を計算機上のいろいろなツールを独立に使って行

なう場合に比べ，本システムを用いる場合は，必要なツールの起動方法や，参照，変更，作成すべき生成物（ファイル名）を知らなくても，適切なツールを使い，適切な生成物を参照，変更，作成することができる．従って，ＪＳＤ本来の作業である「エンティティ構造図を作る」等の作業に専念できるようになった．また，各開発過程で作成した生成物に対して，ツールなどの利用により機械的に判定できるもの（例えば，ファイルの存在）に関しては，開発者は判定に関与する必要はなくなった．一方，機械的に判定できないもの（例えば，作成した生成物の内容の正しさ）に関しては，条件判定の指針が与えられ，その判定のよりどころとなる生成物が表示されるため，判定作業が容易になった．さらに，前提条件，完了条件を判定した結果，次に行なうべき作業の方向付けを行なう，というような開発過程管理を，開発者が行なう必要がなくなった．

この方法で作成したシステムの機能や使いやすさ，生成物の品質等については，用いるいろいろなツールに大きく依存している．本システムは，汎用的なテキストエディタや図形エディタ等のツールしか用いることができなかったため，作業の多くを人間に依存する必要があり，また，各ツールのユーザインタフェースの違いから効率よく使用しにくかった．しかし，例えば「仕様書から動詞を抜き出す」，「エンティティ構造図からテキストへ変換する」等の強力なツールが利用でき，またそれらのツールのユーザインタフェースの統一がとられているならば，作業効率や生成物の品質を向上させることができよう．

(4) 記述の妥当性について

実際にＪＳＤの支援システムを使用して問題を解くにあたり，文献によって理解されたＪＳＤに従った開発作業が行なえた．従って，ＪＳＤの自然語による記述，ＰＤＬによる記述はそれぞれ妥当なものと考えられる．

ＰＤＬでは，その意味が形式的に定義されているため，詳細化の

正しさや記述が満たすべき性質を形式的に検証できる．これらの検証のためには，代数的仕様の検証のために提案されているいろいろな手法[東野88]を用いることができる．

## 第5節　結言

自然語によるＪＳＤの性質記述を行なうことによって，ＪＳＤの各開発過程の入出力生成物の関係や性質を記述することができた．さらに，この記述を開発支援システムの仕様とみなし，ＰＤＬによる形式的な記述に変換した後，段階的に詳細化することによって実行可能なＰＤＬスクリプトを得た．この詳細化では，抽象度の高い記述を機械的に置き換えるか，関数定義を追加することで完全なスクリプトを得ることができた．

このスクリプトを実行することにより，各作業に適したツールが自動的に起動される．各開発過程の条件判定を容易に行なうことができ，条件判定の結果によって次に行なうべき作業に自動的に移る．矛盾があって先に進めないときには適切な開発過程へ後戻りする．このような支援を行うことによって，ＪＳＤに従った開発を容易に進めることができる．

ＪＳＤの他に，ＪＳＰ（Jackson Structured Programming）[Jackson 75]と呼ばれる開発方法についても，同様な手法でＪＳＰの自然語による性質記述からＰＤＬによる形式的な記述を得て，それを詳細化，実行し，支援システムを作成した．スクリプトのサイズは，コメント等も含めて約530行であった．

# 第6章　結論

本論文では，ソフトウェア開発過程を記述するための動作的記述モデルを提案し，記述用言語に必要な機能について論じた．この議論に基づいて，開発過程記述用言語ＰＤＬを設計し，さらにそのインタプリタを作成した．次に，段階的詳細化を用いてソフトウェア開発過程を記述する手法を提案した．これを用いて，ソフトウェア開発技法のひとつであるＪＳＤの開発過程を記述し，その記述に基づいて実際にソフトウェア開発が行えることを示した．

今後は，提案した開発過程の記述方法，およびこの記述に基づく開発支援システムが実際のソフトウェア開発に有効な手段であることを，さまざまな開発過程の記述と支援の実績を通じて示す必要がある．

ここでは，本研究に関する今後の問題点について論じる．

(1) 記述の形式的な性質の利用

ＰＤＬによる記述は代数的言語の枠組みによって意味定義が与えられている．第３章で述べたように，代数的言語の持つ性質から，ＰＤＬは段階的な詳細化によってスクリプトの開発を行うのに適している．しかし現状では，この代数的言語による形式的な意味定義の利点を十分に利用しているとは言えない．例えば，段階的詳細化の前後で記述が一貫しているか，あるいは部品化された記述が利用できるかどうかの検証など，記述の形式的意味を利用することによって開発過程の記述を効果的に進めることができると考えられる．

ＰＤＬでは，計算機資源および開発者を抽象的に表現するためにシステム状態を導入しているが，現在のＰＤＬではシステム状態がどのような構造によって定義されるのか，また関数の適用によってその状態がどのように変化するのかなどについては定義していない．

システム状態の定義は開発過程の記述方法自体に大きく影響する．システム状態を詳細に定義することによって，開発過程をより明確に記述できる可能性があると考えられる．

(2) チームによる開発への拡張

本論文では，開発者が一人で行う作業だけを記述の対象とした．しかし，一般のソフトウェア開発は複数人からなるチームで行うことが多い．また，開発過程の記述に基づいた支援は，個人による開発よりも，むしろ複数人が共同して行う開発作業に対して求められていると考えられる．このため，支援システムの実用性を高める上で，チームによる開発への拡張は不可欠である．

しかし，記述すべき対象を複数の開発者が複数の計算機上で行う開発作業に拡張した場合，各作業者の間の情報のやりとりや，分散した資源や生成物の管理をどのように行うかということが問題となる．さらに，各作業者をどの作業に配置するのか，全体の作業の管理・運営はどのように進めるのか，などといった人的側面にも注目しなければならない．

(3) スクリプトの動的な変更

現在，開発作業を支援するためのスクリプトは，開発作業が始まる時点で既に存在しており，開発作業を通じてその内容は変わらないものと考えている．

これに対して，作業の途中で他の開発過程のスクリプトに切り替えたい，あるいは，いくつかのスクリプトの中から適切なスクリプトを作業中に動的に選択し，それらを組み合わせて開発作業を行いたい，などの要求も考えられる．

実現の面から検討すると，開発者がスクリプトを自由に変更できるようにしてしまうと，異なるスクリプト間で作業に関する情報を利用できなかったり，あるいは作業自体を適切に進められなくなるなどの問題が生じることが考えられる．

しかし，開発者自身で開発環境を整備し，より使いやすくして行くために，このような機能の導入についても検討する必要があるであろう．

(4) 生成物の動的な管理

ソフトウェア開発の支援には，開発過程管理と生成物管理の2つの面からの支援が考えられる．本論文で述べた動作的モデルは開発過程の管理に重点を置いたモデルである．従って，開発過程の進行に伴う生成物の流れは記述できるが，それぞれの生成物の内容に関する情報は十分には記述できないということがわかってきた．

一般にプログラムは複数のモジュールから成り，いくつものソースファイル，ヘッダファイルなどから構成されている．このような複数の生成物から構成されるソフトウェアの開発のためには，それぞれの生成物の生成，修正，あるいはバージョンの管理などの作業を支援できなければならない．例えば，あるソースプログラムを修正する場合，同時に修正しなければならないソースプログラムは何か，作り直さなければならないオブジェクトプログラムは何か，などの情報を利用して必要な部分への修正を効率的に行いたいという要求がある．

しかし，ソフトウェアをどのようなモジュールから構成し，どのようにファイルに分割するかということは開発を開始してから決まる事項であり，また，後から変更されることもある．このため，開発作業中に現れるすべての生成物についての作業や情報をあらかじめ記述しておくことはできない．

このような複数の生成物の管理を開発作業中に動的に行うために，開発過程の記述，実行の方法と生成物の関係について，さらに検討を加える必要がある．

(5) プロダクトサーバの導入

ＰＤＬシステムでは，生成物に対する具体的な操作はＵＮＩＸの

ツールを起動することによって行うという方法をとっている．このような記述を行うためには，ＵＮＩＸとそのツールに関する多少の知識が必要である．また，引数の文字列を作成する部分の記述など，ツールとのインタフェースが複雑なものになる場合がある．

この点については，よく使用される機能や特定のツールに関する記述をライブラリとして蓄積しておき，開発過程の記述はこのようなライブラリを前提として記述するという方法が有効である．第５章で述べたＪＳＤに対するＰＤＬスクリプトもこの方法を利用して記述され，実際に記述の労力を減らすことができた．

これに対し，生成物に対する基本的な操作についてはシステムがある程度の機能を提供し，記述はこれらの機能を組み合せるだけで定義できるようにするという方法が考えられる．システム内で生成物の操作管理を行う部分をプロダクトサーバと呼ぶ．プロダクトサーバの考え方を導入することにより，生成物の操作を容易に記述できると期待される．

しかし，プロダクトサーバ自体の必要性も含め，どのようなレベルのどのような機能を提供すべきかについて，十分な検討はなされていない．今後，さまざまな開発過程を記述して経験を蓄積する必要がある．

(6) 効果的なユーザインタフェースの開発

ソフトウェア開発作業を支援するシステムは，開発過程の実行の状況や，次に行うべき作業，作業の対象となる生成物についての情報を整理された形で提供し，開発者の要求に適切に応えなければならない．このためには，使いやすいインタフェースを持った実行処理系が欠かせないものであると考えられる．

開発作業支援のためにどのような機能がインタフェースに求められているのかという点も，今後検討すべき課題である．

# 謝　辞

本研究に関して，理解ある御指導を賜り，つねに励ましていただいた鳥居宏次教授に心から深謝致します．

種々の適切な御指導，御教示をいただいた情報工学科の嵩忠雄教授，都倉信樹教授，谷口健一教授，首藤勝教授に深謝致します．

大学院において御指導をいただいた櫻田榮一教授，橋本昭洋教授，宮原秀夫教授，脇田壽教授，豊田順一教授，北橋忠宏教授，藤井護教授，田村進一教授に深謝致します．

さまざまな面で御助言，御援助をいただいた鳥居研究室の菊野亨助教授，井上克郎講師，工藤英男助手，松本健一助手に心から感謝致します．

本研究をすすめる上で，ＪＳＤの解釈に関して御協力をいただいた日本ユニシス株式会社加藤潤三氏に感謝致します．

本研究に関して御討論いただいた鳥居研究室の方々，特に，ＪＳＤの開発過程記述に御協力いただいた野村研仁氏（現ＮＴＴ），稲田良造氏（現ダイキン工業），ＰＤＬ処理系の作成に御協力いただいた新田稔研究員（ＳＲＡ），院生飯田元氏，西村好洋氏に感謝致します．

# 文　献

[Cameron 86]

J.R.Cameron: "An Overview of JSD", IEEE Transactions on Software Engineering, Vol.SE-12, No.2, pp.222-240(1986).

[Dart 87]

S.A.Dart, R.J.Ellison, P.H.Feiler, and N.Habermann: "Software Development Environments", IEEE Computer, Vol.20, No.11, pp.18-28 (1987).

[Ehrig 85]

H.Ehrig and B.Mahr: "Fundamentals of Algebraic Specification 1", Springer-Verlag (1985).

[Garlan 89] D.Garlan: "The Role of Formalized Domain-Specific Software Frameworks", Proceedings of the 5th International Software Process Workshop: Experience with Software Process Models, (to appear) (1989).

[東野88]

東野, 関, 谷口: “代数的仕様から関数型プログラムの導出とその実行”, 情報処理, Vol.29, No.8, pp.881-896 (1988).

[Harel 88]

D.Harel, H.Lachover, A.Naamad, A.Pnueli, M.Politi, R.Sherman, and A.Shtul-Trauring: "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", Proceedings of the 10th International Conference on Software Engineering, pp.396-406 (1988).

[Humphrey 89]

W.S.Humphrey and M.I.Kellner: "Software Process Modeling: Principles of Entity Process Models", Proceedings of the 11th International Conference on

Software Engineering, pp.331-342 (1989).

[稲田88]

稲田, 荻原, 井上, 菊野, 鳥居: “ジャクソン開発法の形式的記述の詳細化とその実行”, 電子情報通信学会技術報告, SS88-36 (1988).

[稲田89]

稲田良造, 荻原剛志, 井上克郎, 鳥居宏次: “ソフトウェア開発過程の形式化とその詳細化による支援システムの作成 －ＪＳＤを例として－”, 電子情報通信学会論文誌(D-I), Vol.J72-D-I, No.12, pp.874-882 (1989).

[井上84]

井上, 関, 谷口, 嵩: “関数型言語ＡＳＬ／Ｆとその最適化コンパイラ”, 電子通信学会論文誌(D), Vol.J67-D, No.4, pp.458-465 (1984).

[Inoue 86]

K.Inoue, H.Seki, K.Taniguchi, and T.Kasami: "Compiling and Optimizing Methods for the Functional Language ASL/F", Science of Computer Programming, Vol.7, No.11, pp.297-312 (1986).

[井上88]

井上, 野村, 稲田, 菊野, 鳥居: “階層的プロセスモデルの提案とそのＪＳＤへの適用”, ソフトウェア・シンポジウム'88, pp.315-324 (1988).

[Inoue 89]

K.Inoue, T.Ogihara, T.Kikuno, and K.Torii : "A Formal Adaptation Method for Process Descriptions", Proceedings of the 11th International Conference on Software Engineering, pp.145-153 (1989).

[Jackson 80]

M.A.Jackson: "Principles of Program Design", Academic

Press (1975), 鳥居宏次(訳): “構造的プログラム設計の原理”, 日本コンピュータ協会 (1980).

[Jackson 82]

M.A.Jackson: "System Development", Prentice-Hall (1982).

[Kaiser 87]

G.E.Kaiser and P.H.Feiler: "An Architecture for Intelligent Assistance in Software Development", Proceedings of the 9th International Conference on Software Engineering, pp.180-188 (1987).

[Kaiser 88]

G.E.Kaiser, P.H.Feiler, and S.S.Popovich: "Intelligent Assistance for Software Development and Maintenance", IEEE Software, Vol.5, No.3, pp.40-49 (1988).

[Kaiser 89]

G.E.Kaiser: "Experience with Marvel", Proceedings of the 5th International Software Process Workshop: Experience with Software Process Models, (to appear) (1989).

[梶谷86]

梶谷, 伊東, 松浦, 谷口, 嵩: “オフィスワークの代数的記述と検証 －図書購入業務の記述－”, 電子通信学会技術報告, AL84-38 (1984).

[嵩86a]

嵩, 谷口, 杉山 : “代数的言語の設計と処理系”, 榎本編, ソフトウェア工学ハンドブック, オーム社, pp.1066-1073 (1986).

[嵩86b]

嵩, 谷口, 杉山, 関: “代数的言語ＡＳＬ／＊ －意味定義を中心に－”, 電子通信学会論文誌(D), Vol.J69-D, No.7, pp.1066-1073 (1986).

[Katayama 89]

T.Katayama: "A Hierarchical and Functional Software Process Description and its Enaction", Proceedings of the 11th International Conference on Software Engineering, pp.343-352 (1989).

[Kellner 88]

M.I.Kellner: "Representation Formalisms for Software Process Modeling", Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process, pp.43-46 (1988).

[Kishida 88]

K.Kishida, T.Katayama, M.Matsuo, I.Miyamoto, K.Ochimizu, N.Saito, J.H.Sayler, K.Torii, and L.G.Williams, "SDA : A Novel Approach to Software Environment Design and Construction", Proceedings of the 10th International Conference on Software Engineering, pp.69-79 (1988).

[野村88]

野村, 井上, 鳥居: “システム開発法ＪＳＤの定義付けの試み”, 情報処理学会ソフトウェア工学研究会 58-1 (1988).

[荻原88]

荻原, 飯田, 新田, 井上, 鳥居: “ソフトウェア開発環境記述用関数型言語の設計と処理系の試作”, 電子情報通信学会技術報告, SS88-35 (1988).

[荻原89]

荻原, 井上, 鳥居 : “ソフトウェア開発を支援するツール起動自動制御システム”, 電子情報通信学会論文誌(D-I), Vol.J72-D-I, No.10, pp.742-749 (1989).

[Ohki 88]

A.Ohki and K.Ochimizu: "Process Programming with Prolog", Proceedings of the 4th International Software

Process Workshop: Representing and Enacting the Software Process, pp.118-121 (1988).

[Osterweil 87]

L.Osterweil: "Software Processes are Software Too", Proceedings of the 9th International Conference on Software Engineering, pp.2-13 (1987).

[Penedo 85]

M.H.Penedo and E.D.Stuckle: "PMDB - A Project Master Database for Software Engineering Environments", Proceedings of the 8th International Conference on Software Engineering, pp.150-157 (1985).

[Penedo 88]

M.H.Penedo and W.E.Riddle: "Software Engineering Environment Architectures", IEEE Software, Vol.14, No.6, pp.689-696 (1988).

[Penedo 89]

M.H.Penedo: "Acquiring Experiences with Executable Process Models", Proceedings of the 5th International Software Process Workshop: Experience with Software Process Models, (to appear) (1989).

[Riddle 87]

W.E.Riddle: "Software Designer's Associates: A Preliminary Description", Proceedings of the 20th Hawaii International Conference on System Sciences, pp.371-381 (1987).

[Roberts 88]

C.Roberts: "Describing and Acting Process Models with PML", Proceedings of the 4th International Software Process Workshop: Representing and Enacting the Software Process, pp.136-141 (1988).

[Taylor 88]

R.N.Taylor, F.C.Belz, L.A.Clarke, L.Osterweil, R.W.Selby, J.C.Wileden, A.L.Wolf, and M.Young: "Foundations for the Arcadia Environment Architecture", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (SIGSOFT Software Engineering Notes, Vol.13, No.5), pp.1-13 (1988).

[Torii 84]

K.Torii, Y.Morisawa, Y.Sugiyama, and T.Kasami: "Functional Programming and Logic Programming for the Telegram Analysis Problem", Proceedings of the 7th International Conference on Software Engineering, pp.57-64 (1984).

[Williams 88]

L.G.Williams: "Software Process Modeling: A Behavioral Approach", Proceedings of the 10th International Conference on Software Engineering, pp.174-186 (1988).