



Title	ジャクソン開発法の形式的記述の詳細化とその実行
Author(s)	稲田, 良造; 荻原, 剛志; 井上, 克郎 他
Citation	情報処理学会研究報告プログラミング (PR0) . 1988, 1988(94), p. 99-108
Version Type	VoR
URL	https://hdl.handle.net/11094/51054
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

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

The University of Osaka

ジャクソン開発法の形式的記述の詳細化とその実行

Refinement and Execution of a Formal Description of the Jackson System Development Method

稲田 良造 萩原 剛志 井上 克郎 菊野 亨 鳥居 宏次
Ryozo INADA Takeshi OGIHARA Katsuro INOUE Tohru KIKUNO Koji TORII

大阪大学基礎工学部
Osaka University

あらまし 近年、ソフトウェア開発過程を形式的に記述する方法についての研究が行なわれている。我々は、既に、ジャクソンシステム開発法 JSD (Jackson System Development) を自然語で表わした後、それを代数的言語によって記述した。今回、この一般的な JSD の定義から JSD の支援環境を生成した。まず、代数的記述をソフトウェア開発過程記述言語 PDL による記述に変換した。PDL は関数型言語で、そのインタプリタが作成されている。代数的記述から得られた PDL 記述には、多くの未定義関数が存在し、また、具体的なユーティリティプログラム(ツールと呼ぶ)名やファイル名の情報が不足している。そこで、必要な情報を段階的に付加し、記述の詳細化を行なって、実行可能な PDL 記述を得た。この記述を実行することによって、JSD に従った開発を行なうために必要なエディタ等のツールが、順次自動的に起動される。この JSD の記述の詳細化の手順を示し、実行した例について述べる。

Abstract We had written a description of JSD(Jackson System Development) in a natural language, and transformed into a specification in an algebraic language. Based on this specification, we have formally generated a support system of the JSD. First, the algebraic specification is translated into a script(program) of Process Description Language PDL. PDL is a functional language and its interpreter is available. The translated script is refined stepwise with additional information such as utility programs(tools) and files to use. By executing the final script obtained, tools such as editors are activated and conditions for each step is examined according to the specification of the JSD.

1. まえがき

ソフトウェアの生産性、信頼性を向上させる目的で、種々のソフトウェア開発法が提案されている。ソフトウェア開発法は、通常、ソフトウェアを生産する全過程をいくつかのプロセスと呼ばれる工程に分かれる。各プロセスでは、仕様書や構造図などのいくつかのソフトウェア生成物(プロダクトと呼ぶ)を入力とし、ある定まった行為を行なうことによって、いくつかのプロダクトを出力する。このような各プロセスを順次実行していき、最後に目的とするプロダクトが得られる。ただし、途中で矛盾が生じたり、先に進めなくなったときには、再度以前のプロセスからやり直す

(バックトラックと呼ぶ)。

しかし、一般にそのようなソフトウェア開発法は、抽象的な概念を使ったり、いろいろな例を用いて説明されることが多い。従って、初心者がこれらの開発法を使用できるようになるためには、多くの時間をかけて例題を理解したり、経験者から教えてもらったりすることが必須である。また、方法論が指定する種々の制約の範囲が不明確で、人によって同一の方法論と誤っているものが、全然異なる行為を行なったりする恐れもある。

近年、ソフトウェア開発過程を理解しやすい形で正確に記述する試みがなされている。代表的なものとして、Osterweil による Process Programming¹⁾ や、

Williams の Behavioral Approach²⁾ などがある。

Process Programming では、ソフトウェアの開発過程を PASCAL 的な構文を用いて、手続き的に記述することを提案している。これに対し、Behavioral Approach では、開発過程の各プロセスの入力プロダクト、出力プロダクトの条件を記述することによって全体を記述することを提案している。

これらの方法では、開発法を比較的厳密に記述しようとしているが、意味定義があいまいであり、それらを用いて開発過程を支援する環境を直接生成することはできない。我々は、意味定義が簡明な方法で方法論をできる限り正確に記述しようとした。また、その記述を順次詳細化していき、実行可能なプログラムを作成し、そのプログラムを直接実行することによって、開発過程の支援環境を生成することを試みた。

我々は、まず、階層的開発モデルと呼ばれる Behavioral モデルに基づきジャクソン開発法 JSD^{3) 9)} (Jackson System Development) の記述を自然語で行ない、それを代数的言語 ASL/1¹³⁾ で記述した^{15) 16)}。本稿では、この ASL/1 による JSD の記述を“JSD の定義”と呼ぶ。Behavioral モデルとは、開発過程の各プロセスの入力プロダクト、出力プロダクトの性質を記述して、開発過程全体を表わそうとしたものである。

ASL/1 の記述は、一般に抽象度が高く、そのままの形で実行するのは困難である。従って、まず、ASL/1 の記述を、ASL/1 の部分言語である関数型言語 PDL の記述に変換する。得られた記述には、(1)プロセスの逐次的な流れ、(2)プロダクトの流れ、(3)各プロセスの前提条件、完了条件が書かれているが、多くの未定義関数が含まれており、また、具体的なバックトラックやツール（エディタやコンパイラ等のユーティリティプログラム）名の情報が含まれていない。従って、次のような4段階の詳細化を行ない、未定義関数をなくし、また、具体的な情報を付加して、実行可能な JSD の記述を得る。

- ①バックトラック付加、
- ②プロダクトのファイルへの割り当て、
- ③作業の複数の仕事への分割、
- ④仕事の具体的なツールへの割り当て。

ここで、“仕事”はツールの起動とそれに付随する組み込み関数のまとまり、“作業”は仕事の集合体であり、開発過程の各プロセス中の前提条件判定、プロセス本体、完了条件判定のいずれかである。

①～④の段階的な詳細化によって生成された PDL による JSD の記述は、PDL の処理系である PDL インタプリタ¹⁴⁾によって解釈、実行される。

このように、PDL による JSD を PDL インタ

リタによって実行することを、“JSD を実行する”という。

JSD の実行によって、JSD の各プロセスに必要なツールが自動的に起動される。また、各開発過程の前提条件、完了条件を機械的に判定し、またはユーザが各条件を満たしているかを入力することにより判定し、次に行なうべき作業に移る。このように、条件判定が容易になり、また、条件判定の結果によって、次に行なうべき作業の方向付けが行なえる。これらより、信頼性の高いソフトウェア作成に役立つと思われる。

2. ジャクソン開発法 JSD とその問題点

ジャクソン開発法 JSD^{3) 9)} (Jackson System Development) は、仕様要求分析、ソフトウェア設計などを含むソフトウェア開発手法である。本章において、JSD とその問題点について述べる。

2.1 ジャクソン開発法 JSD

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を現実起こる行動、出来事について抽象化した“モデル”を作成し、その後に詳細な機能を付加していく。そして最後に実現を行なう。

JSD は大きく、モデル化の段階、要求機能の仕様化の段階、仕様の実現化の段階の3つの段階に分けられる。

モデル化の段階では、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、出来事に着目し、それらの行動、出来事について抽象化した“モデル”を作成する。

この段階は“モデル”の作成の程度によって、さらに、エンティティアクションステップ、エンティティ構造ステップ、初期モデルステップの3つのステップに分割される。

要求機能の仕様化の段階では、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。この段階を機能ステップと呼ぶ。

仕様の実現化の段階では、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。この段階を実現化ステップと呼ぶ。

図1にこの5つのステップの流れを示す。

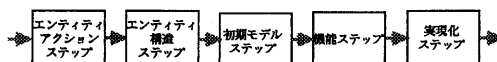


図1. JSDの開発過程

このように JSD では、開発過程を比較的明確に分割、順序付けを行なっている。また、各過程で作成すべきプロダクトの記法についても、比較的詳細に定めている。従って、JSD は他の開発法に比べて、“定義”を与える対象としてふさわしいと思われた。

2.2 JSD の問題点

2.1 のように、JSD は開発過程の分割、順序付けや各プロセスで作成すべきプロダクトの記法等を比較的细节に定めている。しかし、以下のような問題がある。

1) 説明記述が例示的である。

例を用いた説明から、一般的な手法を類推させるような記述が多い。

2) 説明記述の抽象度が一定していない。

ある手法については、詳しく、具体的に説明しているが、ある手法については、それほど詳しい説明が与えられていない。

3) 開発過程や手法が変化している。

開発過程の分割法が Jackson²⁹⁾ と Cameron³⁰⁾ により変わっている。また、データの扱い方等に関しても年ごとに新たなものとなっている。

これらの問題により、初心者にとっては JSD は理解し難く、利用し難いものとなっている。

3. JSD の形式化、及び詳細化の方法とその特徴

JSD には、2.2 で述べたように、説明記述について 1) 例示的、2) 抽象度が様々、3) 変更点がある、といった問題点がある。このことが、JSD を理解、利用するのを困難なものとしている。

我々は、JSD の解説を行なっている論文や本を理解し、また、JSD の経験者から調査を行ない、JSD を“直接的に”記述した²⁵⁾。ここでいう“直接的な”記述とは、例を用いずに、具体的に各プロセスの入出力プロダクト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ものである。この記述を行なう際に、できる限り不要な情報を削ったり、例から推測できる情報を追加して、記述の抽象レベルを統一した。また、この記述の対象とする JSD の文献は (8) とした。

この自然語の記述をもとにして、代数的言語 ASL / 1²³⁾ で JSD の記述を行なった²⁵⁾。ASL / 1 は、その意味が簡明、かつ厳密に定義されており、自然語のようなあいまいさが無い。本稿では、この ASL / 1 による JSD の記述を“JSD の定義”と呼ぶ。

この JSD の定義、及び、自然語による JSD の記述は、後述する階層的開発モデルに基づいている。

JSD の定義には、その詳細が記述されていない、いわゆる未定義関数が多数存在する（例、図 2 関数“システムが対象とする世界に関する記述がある？”）。未定義関数の意味はその関数の表現式自身であると定めているが、より具体的な「値」等は得られない。我々は、これらの未定義関数をより具体的に定義していき、また、その他必要な情報を追加して、最終的に機械的に実行可能な記述を得るために、“JSD の記述”をもとにして、5. で述べる手順で段階的に詳細化を行なった。

この詳細化の最終段階の記述に現われる関数は、UNIX の C シェル上で稼働する“ツール（コマンド）”を起動させる関数や、ウィンドウを操作する関数になっている。

4. 階層的開発モデルに基づく JSD の形式的記述

4.1 階層的開発モデルの提案

ソフトウェア開発過程のプロセスとは、一つ以上の入力プロダクトを用いて一つ以上の出力プロダクトを生成する開発過程の単位をいう。

各プロセスには、そのプロセスの本体の作業を始めるために満たしていなければならない条件（前提条件）と、その作業を終了するために満たしていなければならない条件（完了条件）が存在する。

また、プロセスは階層的に捉えることができ、一つのプロセスをさらに抽象度の低いいくつかのプロセス（サブプロセス）に分割することができる。ただし、どのように分割しても、各プロセスは必ず入力プロダクトと出力プロダクトを持つ。また、各プロセスの出力プロダクトは、必ず、そのプロセス以降に行なわれるプロセスの入力プロダクトであるか、または、開発過程全体の最終出力プロダクトである。

階層的に捉えることのできる開発過程を記述するための抽象的なモデルとして階層的開発モデルを提案した。これは①プロセスの集合、②入力プロダクトが満たすべき前提条件の集合、③出力プロダクトが満たすべき完了条件の集合、④同一レベルでのプロセス間の入出力プロダクトの受け渡し関係の集合、⑤異なるレベル間でのプロセスの入出力プロダクトの対応関係の集合の 5 つの組で表現される Behavioral モデルである。

異なるレベル間でのプロセスの入出力プロダクトの対応関係を記述することによって、プロセスの階層的な記述が行なえる。

4.2 J S D の形式的記述

4.1で述べた階層的開発モデルに基づいて、J S D を(1)自然語で記述し、さらに(2)代数的言語による記述に変換する、という段階を経て、形式的な記述が得られている。以下、これらの段階について述べる。

(1)自然語による J S D の記述

J S D の過程の自然語による記述の一部を表1に示す。表1では、J S Dのプロセスと各プロセスの入力と出力が属する言語、及び前提条件、完了条件、プロダクトの受け渡し関係、対応関係を定義している。

例えば、<プロセス>においては、J S Dは、エンティティアクションステップ、エンティティ構造ステップ、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ことを記述している。

J S Dの開発段階の一つであるエンティティアクションステップの<入力>“ソフトウェアシステム要求仕様書”は“仕様書記述言語”(一般には自然語と同等)に属し、<前提条件>“システムが対象とする世界に関する記述がある”などが満たされていなければならないことを示す。

表1. J S D の自然語による記述

<<J S D>>	
<プロセス>	
J S Dは5つのサブプロセス	
・エンティティアクションステップ	
・エンティティ構造ステップ	
・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。	
<<エンティティアクションステップ>>	
<入力>	
・ソフトウェア要求仕様書：仕様書記述言語	
<出力>	
・エンティティリスト：list of エンティティ	
・アクションリスト：list of アクション	
<前提条件>	
・ソフトウェアシステム要求仕様書が存在し、システムが対象とする世界に関する記述がある。	
<完了条件>	
・エンティティリストが正しく作成されている。	
・アクションリストが正しく作成されている。	
...	
<<エンティティ構造ステップ>>	
<入力>	
...	
<プロダクトの受け渡し関係>	
・エンティティ構造ステップへの入力は、エンティティアクションステップへの入力と出力である。	
・初期モデルステップへの入力は、エンティティアクションステップへの入力とエンティティ構造ステップの出力である。	
...	
<プロダクトの対応関係>	
・J S Dへの入力は、エンティティアクションステップへの入力である。	
・J S Dの出力は、実現化ステップの出力である。	

なお、この表で用いられている“仕様書記述言語”，“エンティティ”，“アクション”はいずれもJ S D特有のプロダクトを記述するための言語であり、各プロダクトはこれらの言語に属するものとする。また、これらの言語については別に定義されているものとする。

<プロダクトの受け渡し関係>では、各プロセスの入力プロダクトが、どのプロセスの出力プロダクトであるかを記述する。

<プロダクトの対応関係>では、プロセスとそのサブプロセス間で同じである入力プロダクトどうし、同じである出力プロダクトどうしを記述する。

(2)代数的言語による J S D の記述

代数的言語 A S L / 1 を用いて行なった J S D の定義を図2に示す。A S L / 1 は、代数的仕様記述言語 A S L / *¹⁾²⁾¹²⁾において、文法及び公理の記述法を具体的に定めた部分言語である。

一般に、代数的言語は、意味の定義が簡明である、抽象的に高いものから低いものまで同じ枠組みの中で記述できる、検証を形式的に行なうことができる、といった特徴を持つ。従って、さまざまな抽象度の記述を厳密に行なえ、ソフトウェア開発過程を記述するのに適していると考えられる。

A S L / 1 では、各関数の引数の型と関数値の型を「関数名：第一引数の型、第二引数の型、...、第n引数の型 → 関数値の型」の形で指定する。また、項の合同関係を「項1 == 項2」で表わす。

```

TEXT J S D プロセス
# J S D
# エンティティアクションステップ
エンティティアクションステップ：
  仕様書記述言語 →
  [list of エンティティ, list of アクション];
# 前提条件
エンティティアクションステップ前提条件：
  仕様書記述言語 → boolean;
エンティティアクションステップ前提条件(spec)
  == システムが対象とする世界に関する記述がある?(spec);
# 完了条件
エンティティアクションステップ完了条件(ent,act)
  == ...;
...
# プロダクトの受け渡し関係
エンティティアクションout1 == エンティティ構造in1;
エンティティアクションout2 == エンティティ構造in2;
エンティティアクションin == エンティティ構造in3;
エンティティ構造out1 == 初期モデルin1;
...
# プロダクトの対応関係
J S D in == エンティティアクションin;
J S D out1 == 実現化out1;
J S D out2 == 実現化out2;
J S D out3 == 実現化out3;

```

図2. J S D の代数的言語による記述

基底データ型としては、論理型、文字列型、集合型、リスト型などを用いる。さらに、JSDのプロダクトを記述するための言語としての“仕様書記述言語”、“エンティティ”、“アクション”などの型は別に定義されているものとする。

“#前提条件”の関数“エンティティアクションステップ前提条件”は、自然語での記述の<入力>（ここでは“spec”として表わしている）を引数として、“システムが対象とする世界に関する記述がある？”と同値である、という述語で定義される。“システムが対象とする世界に関する記述がある？”は自然語での記述<前提条件>が満たされていれば値「真」をとる述語関数である。“#完了条件”についても同様である。

“#プロダクトの受け渡し関係”、および、“#プロダクトの対応関係”では、「エンティティアクションステップout1 == エンティティ構造ステップin1」などのように、一致関係にある入出力プロダクトの同値関係を定義する。

5. 形式的記述の詳細化

4. で記述したJSDの代数的記述からは、次の3つの情報が抽出できる。(1)JSDのプロセスの逐次的な流れ、(2)プロダクトの流れ、さらに、(3)各プロセスの前提条件、完了条件、である。

ここではまず、これらの情報を表わす形式的な記述を、関数型言語PDL (Process Description Language)を用いて記述する。PDLはソフトウェア開発過程を記述するために、今回開発した関数型言語である。PDLについては、5.1で述べる。

4. で記述したJSDの代数的記述からPDLによる記述を作成し、さらに段階的に詳細化する。

JSDの代数的記述から得られたPDLの記述には、現実の開発を行なうために必要な情報が含まれていないために、未定義のままの関数が多く含まれている。

そのため、5.2で述べる詳細化を行ない、順次必要な情報を付加し、未定義関数のないPDLの記述を得る。

5.1 ソフトウェア開発過程記述言語PDL

ソフトウェア開発過程は、計算機上で稼働することのできるソフトウェア資源を、人間が意味のある順序で利用し、目的のソフトウェア（プロダクト）を開発することであるといえる。ここでは、計算機上で利用することのできるソフトウェア資源を“ツール”と呼ぶ。

従って、計算機上で利用することのできるツールを、ある決まった順序で起動するということが記述でき、その記述に従ってツールが利用可能なものとなることによって次の利点が考えられる。ソフトウェア開発過程の中で使うべきツールを知らない一般の開発者は、ツールの起動には全く関与する必要がなくなり、そのツールを使って行なうべき事だけに集中することができる。

今回、我々は、ツールの起動の順序を記述できる言語を開発した。この言語を、ソフトウェア開発過程記述言語PDL (Process Description Language)と呼ぶ。

PDLの特徴には次のようなものがある。

1)ASL/F¹³⁾に基づく関数型言語である。ASL/Fは関数型言語で、ASL/1の部分言語である。従って、意味の定義が簡明に行なえる、形式的な検証が容易である、等の特徴を持つ。

2)基底データ型として、整数型、文字列型、論理型の他に、システムの状態を表わすState型がある。PDLでは、目的の行為を行なうために、主にStateを参照、変更するような記述を行なう。

3)並列実行を記述できる。ソフトウェア開発過程には、並列に仕事を行なうことが多くある。例えば、あるテキストファイルを参照しながら、新しくテキストファイルや図形ファイルを作成するなどである。このように並列な仕事を行なえるように、PDLには並列に関数を評価する記述法が用意されている。

4)組み込み関数として、ツールを起動する関数や、ウィンドウを操作する関数が用意されている。これらによって、自由にウィンドウを開き、その上でツールを起動することができる。

5)マクロ機能が充実している。引数無しマクロ、引数付きマクロ、また、その関数定義でのみ有効なローカルマクロ等の機能が用意されている。

PDLの処理系として、PDLインタプリタ¹⁴⁾が作成されており、PDLの記述を実行することができる。現在はUNIXのウィンドウシステム上で稼働している。

PDLによる簡単な記述例を図3に示す。この記述は、C言語によるプログラム“samp.c”の作成過程（関数Work）を表わしている。Workではウィンドウを開き（関数wopen）、エディット（関数Edit）を行なった後、コンパイル（関数Comp）をおこなう。Editでは、UNIXのツールviでファイル“samp.c”をエディット（関数execによってツールを起動）する。Compでは、ツールccで行なうコンパイルが成功すれば終了する（Comp then 節）が、コンパイルが失敗すればさらにエディット、コンパイルを繰り返

す (Comp else 節)。ここで、S はシステムの状態 State 型の変数であり、S1 は exec(CC,S) を行なった後の状態を表わす。また、「#let」文はマクロ定義、関数 status は C シェルのステータスコードを返す組み込み関数である。

```
#let VI: "vi samp.c"
#let CC: "cc -o samp samp.c"
Work(S) == Comp(Edit(wopen(S)));
Edit(S) == exec(VI,S)
Comp(S) == If status(exec(CC,S):S1) = 0
then S1
else Comp(Edit(S1));
```

図3. PDLによる簡単な開発過程の記述例

5.2 JSDの段階的詳細化

4. で得られた JSD の代数的記述を、PDL による記述 (図6. a) に変換する。このとき PDL の記述には、(1) プロセスの逐次的な流れ (図5. a) と (2) プロダクトの流れ (図5. b), (3) 各前提条件, 完了条件の情報が含まれている。しかし、この記述には未定義関数が数多く存在する。ここで未定義関数とは、関数の意味はその表現式自身であると定めているが、具体的な「値」等は得られない関数をいう。未定義関数が含まれる理由には次のことが考えられる。

1) JSD の定義に情報が含まれていない。例えば、各プロセスの前提条件, 完了条件が成り立たないときの戻すべきプロセスの情報や各プロセスで実際に使用すべき「ツール」の情報などである。

2) 実現不可能な条件判定がある。各前提条件, 完了条件の中には条件判定が知的で人間が判定を行なう必要のあるものが少なくない。これに関しては、既存の「ツール」を利用できない。

未定義関数の存在しない記述へ変換するための詳細化とは、次のことをいう。上記の1)に対しては欠けている情報を付加し、2)に対しては、ツールなどを用いて機械的に判定できない条件判定はユーザに尋ねるようにする。このような、詳細化するために、以下に述べる手順 (図4 に示す) を提案し、この手順に従って JSD を詳細化した。

- ① バックトラック付加。
- ② プロダクトのファイルへの割り当て。
- ③ 作業の複数の仕事への分割。
- ④ 仕事の具体的なツールへの割り当て。

ここで、「作業」は複数の「仕事」から成り、「仕事」は一つ、または複数の「ツール」で置き換えられる。具体的には、「ツール」とは UNIX 上で稼働するソフトウェア、すなわち UNIX のコマンドをいう。例

えば、UNIX のエディタ vi などである。実際には、コマンドを実行する PDL の組み込み関数 exec である。また、「仕事」とは、一つのツール、もしくは切り離すことのできないいくつかのツール群を使用して

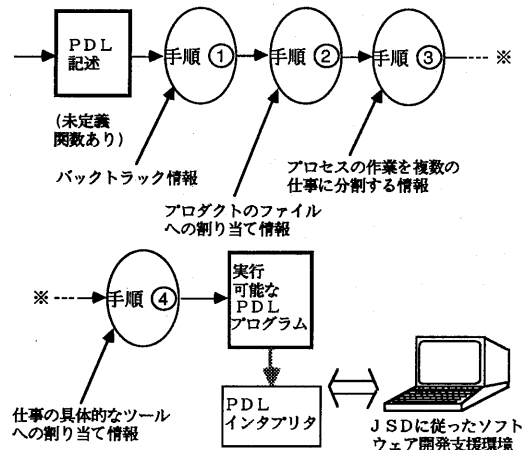
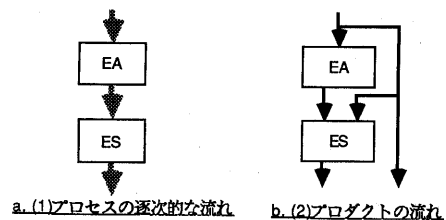
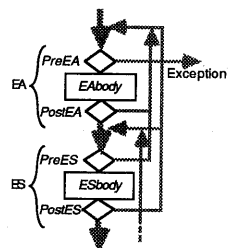


図4. JSD記述の段階的詳細化の手順

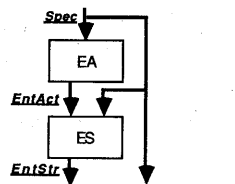


a. (1)プロセスの逐次的な流れ

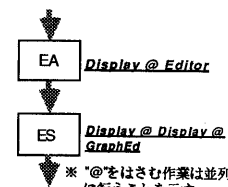
b. (2)プロダクトの流れ



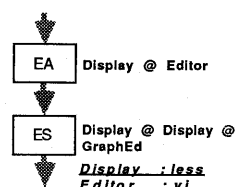
c. バックトラック情報



d. プロダクトのファイルへの割り当て



e. 作業の複数の仕事への分割



f. 仕事のツールへの割り当て

図5. 各手順で付加する情報

行なう行為をいう。例えば、ツール vi を用いて行なうエディットなどである。“作業”とは、いくつかの仕事の集合体で、開発過程のプロセスにとって意味のある単位をいう。例えば、プロセスの本体や条件判定部などであり、それらはいくつかの仕事の集合で定義される。

次に、各手順の詳細について述べる。但し、ここでいうプロセスは前提条件判定部、プロセス本体、及び、完了条件判定部に分かれるものとする。

① “バックトラック付加”では、各プロセスの入力、出力が満たすべき条件（前提条件、完了条件）が成り立たない場合の、バックトラックに関する情報を、“(1)プロセスの逐次的な流れ”に付加する（図5. c）。記述の形式は、

```
「前提（完了）条件判定部(S) ==
  if 前提（完了）条件(S)
  then 正規の流れ(S)
  else 後戻りの流れ(S);」
```

である。この段階での情報付加後の記述を図6. b に示す。新たに変更、追加された関数定義は、図6において太字で表わす。

② “プロダクトのファイルへの割り当て”では、“(2)プロダクトの流れ”に関する情報から、プロダクトの同値関係を導き、同値関係にあるプロダクトを同一の具体的なファイルに割り当てる（図5. d）。記述の形式は、「プロダクト1 プロダクト2 ……: “ファイル名”」（マクロ定義）である。この段階での情報付加後の記述を図6. c に示す。

③ “作業の複数の仕事への分割”では、各プロセス内での作業をいくつかのツールや組み込み関数の実行で置き換えられる仕事単位に分割する（図5. e）。ここで記述の詳細化を行なうのは、各プロセスの本体、及び、前提条件、完了条件である。プロセス本体の記述の形式は一般に、「プロセス本体(S) == 仕事1(S) @ 仕事2(S) @ …… @ 仕事n(S);」, 前提条件、完了条件の記述の形式は、「前提（完了）条件(S) == 述語1(S) & 述語2(S) & …… & 述語m(S);」のように表わされる。ここで、“@”をはさむ関数は並列に評価されることを表わす。なお、前提条件、完了条件の記述は、(3)の“前提条件、完了条件”から変換する。この段階での情報付加後の記述を図6. d に示す。

④ “仕事の具体的なツールへの割り当て”では、③で分割した仕事、述語にツールを割り当てる（図5. f）。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、未定義関数の存在しない記述を完成する。記述の形式は、ツールの起動については、「仕事(S) == wclose(exec("ツ

```
JSD(S) == IP(FU(IM(ES(EA(S)))));
EAIn == ESin2;
EAout == ESin1;
EAPreCond(EAIn,S) == システムが対象とする世界に関する
記述がある?(EAIn,S);
```

a. 階層的開発モデルに基づいた記述

```
JSD(S) == PreEA(S);
PreEA(S) == if EAPreCond(S)
  then EAPost(EAbody(S))
  else Exception(S);
EAbody(S) == EAwork(EAIn,EAout,S);
PostEA(S) == ……;
```

b. バックトラック付加後の記述

```
#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);
EAbody(S) == EAwork(EAIn,EAout,S);
PostEA(S) == ……;
```

c. プロダクトのファイルへの割り当て後の記述

```
#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);
EAbody(S) == EAwork(EAIn,EAout,S);
PostEA(S) == ……;
EAPreCond(S) == Exist(EAIn,S) &
  SatisfyEASpec(EAIn,S);
EAwork(In,out,S) == Dispaly(In,S) @
  Editor(out,S);
EAPostCond(S) == ……;
```

d. 作業の複数の仕事への分割後の記述

```
#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);
EAbody(S) == EAwork(EAIn,EAout,S);
PostEA(S) == ……;
EAPreCond(S) == Exist(EAIn,S) &
  SatisfyEASpec(EAIn,S);
EAwork(In,out,S) == Dispaly(In,S) @
  Editor(out,S);
EAPostCond(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(Message,S)) = "yes"
  then TRUE
  else FALSE;
```

e. 仕事の具体的なツールへの割り当て後の記述

図6. 各段階の情報付加による記述の変化

ール 1 ",exec("ツール 2 ",...,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 6.e 関数 Exist) ものものがあるが、そのようなものばかりではない。このようなときには、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 6.e 関数 SatisfyEASpec) する。この段階での情報付加後の記述を図 6. e に示す。

以上の詳細化の手順によって、未定義関数が存在しない、PDL による JSD の実行可能な記述が完成した。完成した PDL による JSD の記述の一部を図 7 に示す。

```
//JSD Script
//Definition of Products(Macro)
#let EAIn ESIn2 IMIn2 FUIn3 IPIn4 : "Spec"
#let EAout ESIn1 : "EntAct"
#let ESout IMIn1 : "Entstr.kfig"

//JSD(Jackson System Development)
JSD(S) == PreEA(S);
//EA(Entity/Action step)
PreEA(S) == if EAPreCond(S)
    then PostEA(EAbody(S))
    else Exception(S);
EAbody(S) == EAwork(EAIn,EAout,S);
PostEA(S) == if EAPostCond(S)
    then PreES(S)
    else PreEA(S);
//ES(Entity Structure step)
PreES(S) == if ESPreCond(S)
    then PostES(ESbody(S))
    else PreEA(S);
....

//Process Body
EAwork(In,out,S) == Display(In,S) @
    Editor(out,S);
ESwork(In1,in2,out,S)
    == Display(In1,S) @
    Display(on2,S) @
    GraphEd(out,S);
....

//PreConditions & PostConditions
//EAPreCond & EAPostCond
EAPreCond(S) == Exist(EAIn,S) &
    SatisfyEASpec(EAIn,S);
EAPostCond(S) == EAEntAct(EAIn,EAout,S);
//ESPreCond & ESPostCond

//Tool Allocation
Display(file,S) == wclose(exec("less "+file,wopen(S)));
Editor(file,S) == wclose(exec("vi "+file,wopen(S)));
GraphEd(file,S) == wclose(exec("key3 "+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 7. PDL による JSD の記述 (実行可能な記述)

6. 形式的記述 JSD の実行

5. の形式的記述の詳細化によって得られた PDL による JSD の記述は、未定義関数が存在せず、PDL の処理系である PDL インタプリタ^{2,4)}によって実行可能である。PDL による JSD の記述を PDL インタプリタを通して実行することを、“JSD を実行する”という。

JSD の実行の様子は、次のようになる。JSD を実行すると、まず最初のプロセス Entity/Action Step(EA) の前提条件判定部を実行する。EA の前提条件判定部では、ツールによって、ファイル“Spec”が存在するかを自動的に判定し、さらに、ファイル中に記述されるべき事が記述されているかをユーザに尋ね、「値」を受ける。ファイルが存在し、正しく記述されていれば、EA の本体を実行する。ファイルが存在しないか、もしくは、存在するが正しく記述されていなければ、例外処理として終了する(仕様書を作成するのは JSD の作業ではない)。

EA の本体では、2つのウィンドウを開き、一方ではファイル“Spec”を UNIX のツール less で表示し、もう一方では、新しいファイル“EntAct”を編集するためのツール vi を起動する。

EA の本体が終了すると、EA の完了条件判定部を実行する。EA の完了条件判定部では、本体で作成したファイルが正しいかの評価をユーザに尋ねる。正しく作成されていれば、次のプロセスの前提条件判定部を評価し、正しく作成されていなければ、EA の前提条件判定部に戻り、再び EA の作業を行なう。ただし、ファイルが存在しなければ(すなわち、本体でなにも行わずに終了すれば)、自動的に EA の前提条件判定部へ戻る。

以上のようなことを、各プロセスについて繰り返す。JSD の実行例を図 8 に示す。JSD の PDL による記述のサイズは、コメント等を含めて 371 行、マクロ定義と関数定義を合わせた数は、216 個である。

JSD を実行することによる利点は以下のようなものである。

JSD の各プロセスの前提条件、完了条件について、ツールなどの利用により機械的に判定できるものに関しては、自動的に判定を行ない、機械的に判定できないものに関しては、ユーザに条件判定の指針を与えることができる。

また、前提条件、完了条件を判定して、その結果、次に行なうべき作業の方向付けを行なう、というようなプロセス管理が行なえる。

これらのことにより、初心者であっても JSD に従った作業が行なえ、より信頼性の高いソフトウェアの

作成に役立つと思われる。

7. あとがき

既に提案されている階層的開発モデルに基づいて記述された“JSDの定義”を詳細化し、実行可能なPDLによる記述を得た。

“JSDを実行”することにより、各プロセスの条件判定部において、条件判定を容易に行なうことができる。さらに、条件判定の結果、次に行なうべき作業の方向付けができる。このことより、初心者でもJSDに従った開発行為が行なえ、より信頼性の高いソフトウェアが作成できる。

ソフトウェアの開発過程において、一般に、初心者にとっては、次に使用するべきツールがわからない（ツールを起動するコマンドを知らない）ことが多い。しかし、開発過程の専門家が、PDLを用いて開発過程

を記述すれば、初心者がその記述を実行することにより、開発過程に従った作業が行なえると思われる。

今後、今回述べた詳細化とは別に、より実際の開発過程に有効なJSDの記述が必要であると思われる。例えば、開発過程の途中で中断したときに、中断したところから再実行できるような記述や、プロダクトに対する管理が行なえるような記述等が有効である。

ジャクソン開発法の一つであるJSP¹⁰⁾ (Jackson Structured Programming) についても、同様な方法でPDLの記述を詳細化し、“実行”した。サイズは、コメント等も含めて528行、マクロ定義と関数定義を合わせた数は334個である。

【謝辞】

JSDに関する種々の御助言をいただいた日本ユニシス(株)加藤潤三氏に深謝します。

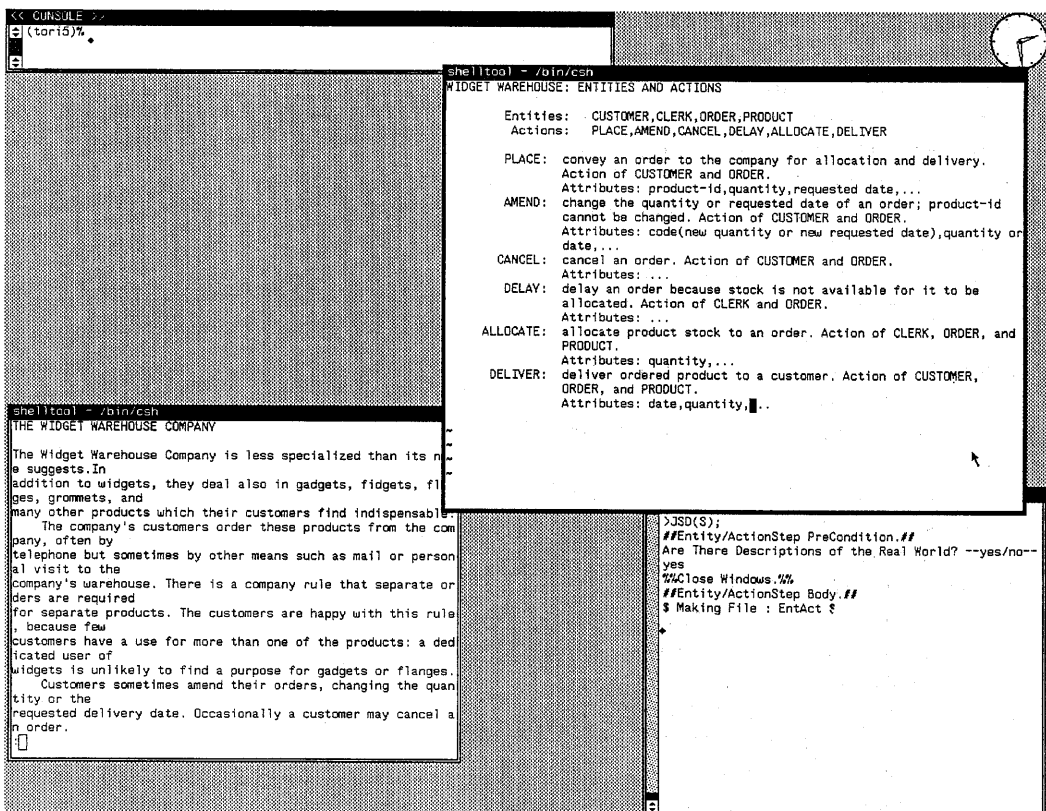


図8. 実行例 (エンティティアクションステップ EA の本体)

【参考文献】

- (1) Osterwell, L. : "Software processes are Software too", Proc. of 9th ICSE, pp.2-13 (1987).
- (2) Williams, L.G. : "Software process modeling: A behavioral approach", Proc. of 10th ICSE, pp.174-186 (1988).
- (3) Kaiser, G.E. and Feiler, P.H. : "An Architecture for intelligent assistance in software development", Proc. of 9th ICSE, pp.180-188 (1987).
- (4) Barghouti, N.S. and Kaiser, G.E. : "Implementation of a knowledge-based programming environment", Proc. of 21st HICSS, pp.54-63 (1988).
- (5) Riddle, W.E. : "Software designer's associates: A preliminary description", Proc. of 20th HICSS, pp.371-381 (1987).
- (6) Bisiani, R., Lecouat, F. and Ambriola, V. : "A planner for the automation of Programming Environment Tasks" Proc. of 21th HICSS, pp.64-72 (1988).
- (7) Ramanathan, J. and Sarkar, S. : "Providing customized assistance for software lifecycle approaches", IEEE Trans. Software Eng., Vol. SE-14, No.6, pp.749-757 (1988).
- (8) Jackson, M.A. : "System development", Prentice-Hall (1982).
- (9) Cameron, J.R. : "An overview of JSD", IEEE Trans. Software Eng., Vol. SE-12, No.2, pp.222-240 (1986).
- (10) Jackson, M.A. : "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).
- (11) 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.93-123 (1986).
- (12) 嵩, 谷口, 杉山, 関 : "代数的言語 A S L / * - 意味定義を中心に -", 信学論 (D), Vol.J69-D, No.7, pp.1066-1073 (1986).
- (13) 井上, 関, 谷口, 嵩 : "関数型言語 A S L / F とその最適化コンパイラ", 信学論 (D), Vol.J67-D, No.4, pp.458-465 (1984).
- (14) 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形の作成", 信学技報, SS 発表予定 (1988).
- (15) 野村, 井上, 鳥居 : "システム開発法 J S D 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).
- (16) 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 J S D への適用", ソフトウェア・シンポジウム '88, pp.315-324 (1988).