

属性文法による構造化分析法の形式的記述

Formal Descriptions of Structured Analysis with Attribute Grammars

馮 安^(*) 大野 浩史^(**) 井上 克郎^(*) 菊野 亨^(*) 鳥居 宏次^(*)

An Feng Koji Ohno Katsuro Inoue Tohru Kikuno Koji Torii

^(*) 大阪大学基礎工学部

Faculty of Engineering Science, Osaka University

^(**) 日本ユニシス株式会社

Nihon Unisys Limited

あらまし 本報告では、構造化分析法の属性文法を用いた形式的記述について述べる。構造化分析法はソフトウェア開発の要求分析と基本設計を行うための方法論で、広く利用されている。しかし、その方法論の記述に関してはガイドラインとして述べられている部分が多く、方法論を適用する上での困難が指摘されている。ここでは、ソフトウェア開発をプロセス（開発の過程）とプロダクト（開発の成果物）の2つを用いて表現する。更に、プロセスとプロダクトの記述に属性を導入することを提案する。形式的記述を求める手順は、まず構造化分析法をデータフロー図で表現し、プロセス、プロダクト、属性を決定した後、属性文法による記述に変換すること、とした。形式的記述を用いれば、要求分析と基本設計の具体的な作業内容がいまもなく指示される。従って、本報告は構造化分析法に基づく開発支援システムの構築の基礎を与えている。

Abstract Structured analysis(SA) is a well known method for requirement analysis. It is said, however, requirement analysis in SA difficult to apply because SA is explained through many rough guidelines. This paper proposes an approach to specify SA with attribute grammars. The specification consists of two parts which define software processes and products of SA respectively. Attributes are introduced both in definitions of processes and products. To obtain products, processes and their attributes, data flow diagrams are first introduced, based on which specifications with attribute grammars are generated.

1. まえがき

近年、ソフトウェアの生産性や信頼性を向上させるためには、ソフトウェア開発における環境の整備が必須であることが指摘されている⁽¹⁰⁾⁽¹²⁾。特に、厳密に定義された方法論に基づいたソフトウェア開発の重要性が明らかになってきた⁽²⁾⁽⁹⁾⁽¹⁵⁾。これまで、数多くのプロジェクトにおいて、幾つかのソフトウェア環境構築の試みが行われている⁽⁵⁾⁽⁶⁾⁽⁷⁾⁽¹³⁾。

ここでは、代表的な方法論の1つとして知られている構造化分析⁽²⁾(Structured Analysis, 以下SAと略す)法について議論する。SA法では、主として、開発すべきシステムの要求分析と基本設計を行う。

SA法は、歴史的には1970年後半Demarcoらによって提案されて以来盛んに利用されたが、80年代に入り殆ど注目されなくなっていた。その主な原因としては、①記述にガイドラインの要素が多いこと、②支援ツールが不備であること、等がある。最近、問題点②が解消されるに伴って、分析から保守までの一貫した支援機能を実現する方法論として構造化法⁽¹⁾⁽²⁾⁽¹⁵⁾が見直されてきている。

方法論自身の記述にガイドラインの要素が残っていると、その方法に従ってソフトウェアを開発するのは困難である。すなわち、方法論に基づいた開発環境の構築を行うためには、その方法論を形式的に記述して

おくことが必要となる。これまでも、ジャクソンシステム開発法JSD⁽⁸⁾に対する形式的記述⁽¹¹⁾、ジャクソン法JSPに対する形式的記述⁽⁸⁾、等が報告されている。

ソフトウェア開発法を形式的に記述する時、開発の過程(プロセス)と開発の成果物(プロダクト)を明確に区別することが行われている。ここで、プロダクトとはソフトウェア開発作業によって作成、または、利用されるプログラム、仕様書などを指す。プロセスとは、幾つかのプロダクトを利用して新しいプロダクトを生成する過程である。ここでは、プロセスとプロダクトの両者の記述において、属性(Attribute)の概念を新たに導入することを提案する。

本報告では、属性文法(Attribute Grammar)⁽⁸⁾を用いたSA法の形式的記述について述べる。属性文法はプログラミング言語の意味を記述するために導入された形式的な体系である。形式的記述はSA法に基づいたソフトウェア開発のプロセスとプロダクトの記述から構成されている。SA法の形式的記述は次のようにして求められる。先ず、データフロー図を用いてSA法を表現する。次に、データフロー図に基づいて、SA法の記述に必要となるプロダクト、プロセス、属性を求める。最後に、それらを利用して属性文法による記述に変換する。

属性文法を用いたシステム開発法の形式的記述としては、片山ら⁽⁹⁾のJSPに対するものがある。それと比べて、本報告での形式的記述はプロセス記述に入出力以外の属性が定義できる、プロセスとプロダクトの関係の関係を定義できる、等の特徴を持っている。

2. 形式的記述の試み

ソフトウェア開発工学の研究の進歩により、プログラミング段階の支援システムが多く開発されている。これに対し、要求分析と基本設計についてはその方法論すら厳密に記述されていない。ここではSA法の属性文法による形式的記述を試みる。

2.1 目的

構造化分析(SA)法を形式的に記述する目的は次の3つである。

- ①SA法に関する従来の記述⁽²⁾の中でガイドラインにすぎなかった部分を明示し、かつ、それを修正する。
- ②SA法を実際のシステム開発に適用する場合、作業内容(プロセスとプロダクト)を段階的に検証する。

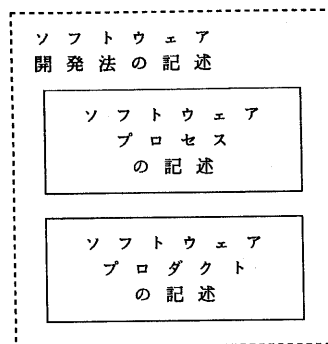


図1 形式的記述の方針

る。

- ③SAの作業を組み込んだ開発支援システムの構築の基礎を与える。

2.2 方針

本報告で提案する記述は、ソフトウェアプロセスの記述とソフトウェアプロダクトの記述から構成されている(図1参照)。ここで、ソフトウェアプロダクト(以下、プロダクトと呼ぶ)は、ソフトウェア開発における成果物を抽象化したものである。プロダクトの構文的な、あるいは、意味的な正しさを厳密に議論するため、プロダクトに対して属性を導入する。プロダクトの記述部分では、プロダクトの集合、プロダクトの属性、及びプロダクト相互の関係や属性相互の関係を定義する。

一方、ソフトウェアプロセス(以下、プロセスと呼ぶ)はプロダクトを生成する過程であり、通常その入出力の関係により定義される。複雑なプロセスの場合、それをより簡単なサブプロセスに分解して記述する。

プロセスの記述部分ではプロセスの分解方法、プロセスの入力プロダクト、出力プロダクトを定義する。記述能力の向上、及び記述の簡潔さのために属性を導入する。例えば、実行環境によって分解の方法が異なるプロセスに対しては、その実行環境を属性として定義する。属性の例としては実行時刻、必要な資源等がある。

最終的な記述言語として、ここでは属性文法(Attribute Grammar)を採用する。その主な理由は、①属性文法が、プロセスとプロダクトの記述に導入した属性を陽に取り扱うことができる、②属性文法では階層的構造が自然に、かつ、読解性良く記述できる、③属性文法による形式的な記述を実際のプログラムに変換することが比較的容易である⁽³⁾、等である。

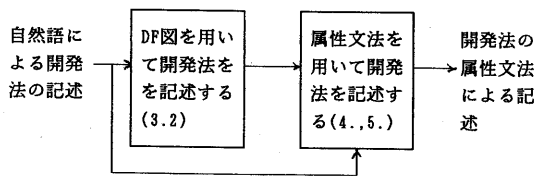


図2 形式的記述の手順

2.3 手順

SA法の形式的な記述を一回の作業で求めることは困難である。ここでは図2に示す様に2段階の方式をとる。まず、データフロー図⁽²⁾ (Data Flow Diagram, 以降DF図と呼ぶ)を用いてSA法を記述する。この時点での記述は未だ非形式的である。DF図はSA法自身の一つの構成要素になっており、プロセスやプロダクトを基本とする記述に適している。次に、DF図による記述から属性文法による形式的記述に変換する。この変換では、DF図に基づいて、使用するプロセス、プロダクト、属性を定める。引き続き、プロセスとプロダクト毎に、属性文法の構文規則、意味規則を定義する。

3. 構造化分析 (SA) 法

3.1 SA法の概要

SA法はシステム開発工程の要求分析と基本設計の工程に適用される。SA法では、トップダウンに分析、設計を進め、構造化モデルを作成する。

SA法で作成するモデルには、現行物理モデル、現行論理モデル、将来論理モデル、及び将来物理モデルの4つがある。分析、設計の作業はこの順番に各モデルを作成することによって実行される。

各モデルはデータフロー図(DF図)、データ辞書(DD)、ミニ仕様書(MS)から構成されている。ここで、DF図はデータの流れを図として表現したものである。DDではデータの定義を、MSでは最も基本的な機能に対する仕様書を定める。

DF図: DF図はモデル記述の中心であり、データフローとバブルから構成される。データフローはシステムで扱うデータの流れを、バブルはシステムを構成する機能単位を表す。図3に電報解析問題⁽⁴⁾の解に対するDF図を示す。DF図で丸印がバブル、有向枝がデータフローを表す。バブルに入る有向枝を入力データフロー、バブルから出る有向枝を出力データフローと呼ぶ。バブルとデータフローのラベルを名前と呼ぶ。

DF図は階層構造が許されている。一つのパブルはよ

り基本的な幾つかのパブルに機能的に分割される。この分割によって、トップダウンの分析が可能となる。図3(b)は図3(a)の電報処理を3つのパブルに分割した状態を描いている。図3(c)も同様である。図3(a), (b), (c)のそれぞれをDF図と呼び、各DF図には名前を付けるものとする。

DD: データ(DF図中のデータフローのラベル)を詳細に記述しているのがDDである。図4に電報解析問題のDF図(図3)中に現われている電報の定義を示す。データ電報は、一連の電文からなり、電文の間は"ZZZZ"で区切られており、2つの連続した"ZZZZ"で終了する。

MS: DF図上でこれ以上分割されないパブルに対する機能の記述をMSで行う。MSにおける記述には、順序、選択、繰り返しだけからなる構造化言語を用いる。図3中の処理結果の出力に対するMSの定義を図5に示す。

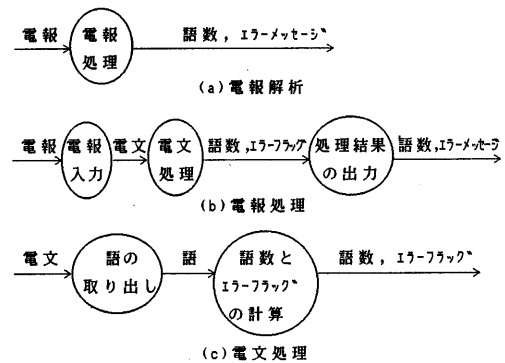


図3 DF図の記述例

電報 = 電文
+ {"ZZZZ" + 電文}
+ "ZZZZ" + "ZZZZ"

図4 DDの記述例(電報の定義)

各電文に対して次のメッセージを出力する:
1. 語数
2. もしフラッグ == 1なら,
 メッセージ"長すぎる語有り"
 そうでなければ,
 メッセージ"長すぎる語なし"

図5 MSの記述例(処理結果の出力の仕様)

3.2 DF図によるSA法の記述

ここではDF図を用いたSA法の記述を示す。この記述は次のR1～R3に従って行っている。

R1:SA法におけるプロセスをバブルに、プロダクトをデータフローに対応させる。

R2:バブルの分割を利用して、プロセスの機能分割を記述する。

R3:DF図中の名前に関して次の制限を置く。

- ① バブルの名前は高々1つの修飾語、丁度1つの他動詞と丁度1つの目的語からなる。(直観的には、“他動詞+目的語”はプロセスの名前に、修飾語はプロセスの属性値になっている。)
- ② データフローの名前は高々1つの修飾語、丁度1つの目的語からなる。(目的語はプロダクトの名前に、修飾語はプロダクトの属性値になっている。)

③ DF図の名前は丁度1つの他動詞と丁度1つの目的語からなる。(DF図の名前はプロセスの名前が対応している。)

3.1で述べたように、SA法による要求分析(Requirement Analysis)は、基本的に既存環境の調査(現行物理モデルの作成)、現行論理モデルの作成、将来論理モデルの作成、将来物理モデルの作成の4つから構成されている。これらの各ステップでの出力を見ると、いずれも、モデルを作成している。そのため、これらをモデルの作成(Create model)というプロセスに抽象化できる。つまり、SA法による要求分析はモデルの作成と言うサブプロセスの4回の実行と見なせる。これを図6(a)に示す。そこで、データフロー(1),(4)の名前はPhys. modelで、(2),(3)の名前はLog. modelである。修飾語Phys.とLog.は目的語Modelの性質に関する制限を表す。更に、Create modelの分解方法には3つ

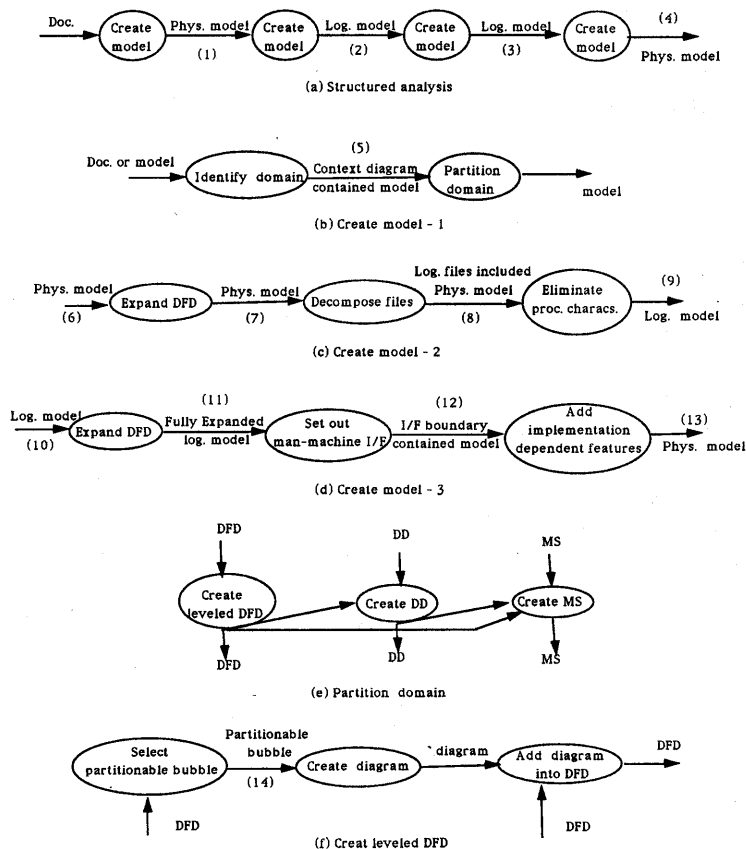


図6 DF図によるSA法の記述

あり、それぞれ図6(b),(c),(d)に示す。どれを選択するかは、入力データフローの目的語 (Model か Doc.) とその性質 (Phys.かLog.) によって決める。

これらの議論に基づいて、DF図を用いてSA法を記述すると図6が得られる。ただし、紙面の都合上、図6の記述は完全なものではなく、バブル分割を最初の5回実行したところまでとなっている。その様子を図7の白丸部分に示す。

4. 属性文法

4.1 属性文法の概要

属性文法は、D.E.Knuth⁽⁸⁾によってプログラミング言語の意味記述のために導入された形式的体系である。属性文法は4項組 $G = (G_0, A, C, R)$ として定義される。ここで G_0 は文脈自由文法、 A は属性の集合、 C は適用条件、 R は意味規則の集合である。

① 文脈自由文法 $G_0 = (N, T, P, S) \dots N$ と T は、それぞれ、非終端記号と終端記号の集合である。 P は構文規則の集合で、 $S \in N$ は開始記号である。各構文規則は次の形とする。

$$X_0 ::= X_1 X_2 \dots X_n$$

但し、 $X_0 \in N$, $X_i \in NUT$ ($1 \leq i \leq n$)。以下では、 X_0 を左辺記号、 X_i を右辺記号と呼ぶ。

② 属性の集合 $A \dots$ 各 $X \in NUT$ には属性の集合 $A(X)$ が割り当てられている。 X の属性 $attr \in A(X)$ を、通常、 $X.attr$ と表す。 $A(X)$ は相続属性の集合と合成属性の集合に分けられる。全ての $A(X)$ の和集合を A と表す。

③ 適用条件 $C \dots$ 各構文規則 $p \in P$ には適用条件が指定されており、 p を適用するための必要条件が属性相互の関係を表す述語の形で与えられている。実際の記述では次のように表す。

where 適用条件

④ 意味規則の集合 $R \dots$ 属性の値を定義するために、各構文規則 p に対し意味規則が指定されている。左辺記号の合成属性と右辺記号の相続属性を定義しなければならない。

属性文法では階層的構造を自然に、読解性高く、しかも簡潔に記述することができる。そのため属性文法の応用範囲は、コンパイラや構造エディタの記述だけでなく、階層的データベース、CAD、階層的プログラミング、等に広まってきている。

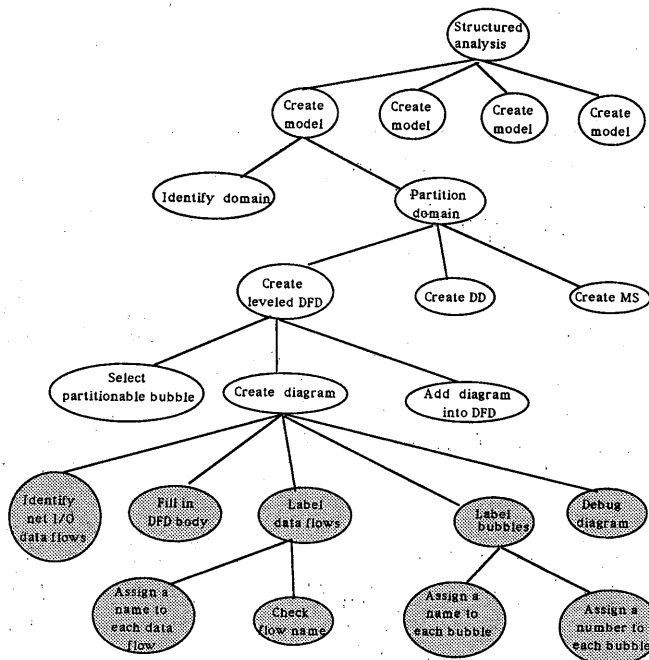


図7 プロセスの階層構造図

4.2 属性文法の書式

属性文法による SA 法の記述はプロダクトの記述部分とプロセスの記述部分から構成されている。図8に属性文法による記述の書式を示す。

プロダクトの記述部分は、属性宣言部と構成規則の記述部からなる。属性宣言部では属性名と属性のデータタイプを定義する。データタイプとしては、“INT”（整数），“STRING”（文字列），“SET”（集合）などがある。一方、構成規則記述部では各プロダクトの分割方法と属性の評価方法を定義する。各構文規則により、左辺のプロダクトが右辺の要素から構成されることを表す。分割条件は構文規則の適用条件である。

（注1）ソフトウェアのプロダクトには2次元的な形状のもの（例えば、図、表など）が多い。このようなプロダクトの記述を簡潔に行うために構成子を導入し、構文規則を次のように拡張する。

プロダクト名 ::= 構成子(要素₁, ..., 要素_n)
または、

プロダクト名 ::= 要素_i
ここで、要素_i (1 ≤ i ≤ n) はプロダクト名である。構成子としては次の4つを考える。

Follows($X_1, X_2, \dots, X_n$) ... 要素 $X_1, X_2, \dots, X_n$ が系列として、この順番に並ぶ。

Contains(X, Y) ... 要素 Y が X の枠の中に存在する。

Connects(Y, X_1, X_2) ... 系列 X_1, X_2 が要素 Y により連結される。

Set($X_1, X_2, \dots, X_n$) ... 要素 $X_1, X_2, \dots, X_n$ は1つの集合である。

プロセスの記述部分はプロセスの属性宣言部と分解規則の記述部からなる。属性宣言部では、属性名と属性のデータタイプを定義する。属性としてはプロセスの入力、出力（プロダクト名）や動的性質（環境、時刻、必要な資源）などがある。一方、分解規則記述部では各プロセスの分解方法と属性の評価方法を定義する。各構文規則により、左辺のプロセスが右辺のサブプロセスに分解されることを表す。分解条件は構文規則の適用条件である。

属性文法によるプロダクトとプロセスの記述例を図9に示す。図9中のプロダクト記述では、DFDの構成と意味を定めている。属性 $Diag_no$ と Bub_no は、それぞれ、DF図の番号の集合とプロセス番号の集合を表す。Diagramsの分割条件で、各DF図が必ず異なる番号

プロダクトの記述

プロダクトの属性宣言部

属性名のリスト : 属性のタイプ ;

プロダクトの構成規則の記述部

プロダクト名 ::= 構成子(要素, ..., 要素)
Where 分割条件
{ 属性値を定義する規則
}

プロセスの記述

プロセスの属性宣言部

属性名のリスト : 属性のタイプ ;

プロセスの分解規則の記述部

プロセス名 ::= サブプロセス名 ... サブプロセス名
Where 分解条件
{ 属性値を定義する規則
}

図8 属性文法の書式の説明図

をもつこと、各DF図と同じ番号のバブルがDF図中に必ず存在すること、が指定されている。

図9中のプロセスの記述ではプロセス Create leveled DFD(階層化DFDの作成)の分解を定めている。分解条件としては、選ばれたバブルは未展開であること、また、作成されたDF図は必ずバブルと同じ番号をもっていること、が指定されている。

（注2）プロセスの定義において、その入出力となるプロダクトに関する記述を参照する必要がある。プロセス X の入出力($X.attr$ と表す)はそのプロダクトの名前と属性で定まる。プロダクト名が一意に決まる(Nameとする)場合は、データタイプとして $X.attr$: Nameと書く。一方、プロダクト名が $\{Name_1, \dots, Name_n\}$ のいずれかである場合は、 $Is(X.attr, Name_i)$ のように明示的に記述する。また、プロダクトNameの属性 $Name.prd_attr$ の値を $prd_attr(X.attr)$ と表す。

```

/* Product Attributes */
DFD.Diag_no, Diagrams.Diag_no, DFD.Bub_no, Diagrams.Bub_no : INT;
Diagram.Diag_no, Title.no, Diagram.Bub_no, Bubbles.no : INT;
Bubble.no, Bubble_identifier.no, NUMBER.img : INT;

/* Product Description */
DFD := Diagrams
{
  DFD.Diag_no := Diagrams.Diag_no;
  DFD.Bub_no := Diagrams.Bub_no;
}
Diagrams := Set(Diagram, Diagrams)
  where ((Diagram.Bub_no  $\supset$  Diagrams[2].Diag_no)
    && (Diagram.Diag_no  $\notin$  Diagrams[2].Diag_no))
{
  Diagrams[1].Diag_no := (Diagram.Diag_no)  $\cup$  Diagrams[2].Diag_no;
  Diagrams[1].Bub_no := Diagram.Bub_no  $\cup$  Diagrams[2].Bub_no;
}
Diagrams :=  $\epsilon$ 
{
  Diagrams.Diag_no := {};
  Diagrams.Bub_no := {};
}
Diagram := Set(Title, Bubbles, Flows)
{
  Diagram.Diag_no := Title.no;
  Diagram.Bub_no := Bubbles.no;
}
Bubbles := Set(Bubble, Bubbles)
{
  Bubbles[1].no := (Bubbles.no)  $\cup$  Bubbles[2].no;
}
Bubbles :=  $\epsilon$ 
{
  Bubbles[1].no := {};
}
Bubble := Contains(CIRCLE, Bubble_identifier)
{
  Bubble.no := Bubble_identifier.no;
}
Bubble_identifier := Follows(STRING, NUMBER)
{
  Bubble_identifier.no := NUMBER.img;
}
Title := Follows(NUMBER, STRING)
{
  Title.no := NUMBER.img;
}

/* Process Attributes */
Develop_DFD.input, Develop_DFD.output, Select_a_bubble.input : DFD;
Select_a_bubble.output, Create_diagram.input : Bubble;
Create_diagram.output.output : Diagram;

/* Process Description */
Develop_DFD := Select_a_bubble Create_diagram
  Where (Bub_no(Select_a_bubble.output)
     $\notin$  Diag_no(Develop_DFD.input)
    && Diag_no(Create_diagram.output)
    == Bub_no(Select_a_bubble.output))
{
  Select_a_bubble.input := Develop_DFD.input;
  Create_diagram.input := Select_a_bubble.output;
  Develop_DFD.output :=
    Develop_DFD.input  $\cup$  {Create_diagram.output}
}

```

図9 属性文法による記述例

5. 属性文法によるSA法の形式的記述

ここでは、DF図による記述に基づいて属性文法によるSA法の形式的記述を与える。

5.1 属性文法への変換手順

DF図によるSA法の記述は、厳密には、未だ非形式的である。例えば、バブル間のインタフェースとなっているデータ自身の定義は記述されていない。

属性文法によるSA法の形式的記述への変換は基本的に次の4つのステップから構成される。

- S1: DF図のデータフローに基づいて、使用するプロダクトとそれらの属性を決定する。
- S2: 全てのプロダクトを属性文法で記述する。
- S3: DF図のバブルに基づいて、使用するプロセスと

その属性を決める。

S4: 全てのプロセスを属性文法で記述する。

各ステップについて、順次、詳しく説明する。先ずステップS1について述べる。DF図中のデータフロー名は修飾語と目的語から構成されている。目的語をプロダクト名とする。次に、その修飾語の意味を厳密に記述するために、プロダクトに属性を導入する。属性としてはプロダクトの構成情報(例えば、大きさ)、作成情報(バージョン、作成時刻、など)、管理情報(データベースへの保存を行うか否か)がある。

ステップS2でのプロダクト X の属性文法による記述は次のように求める。先ず、プロダクトの構成に注目し、構成要素の集合 $\{X_1, X_2, \dots, X_n\}$ と要素間の関係を定める構成子 Y を決める。次に構文規則 $X ::= Y(X_1, X_2, \dots, X_n)$ を定義する。もし X_i が、文字列、整数などの基本要素でなければ、更に X_i を左辺記号とする構文規則を定義する。最後に、 X の各属性を定義するための意味規則を定義する。

ステップS3について説明する。DF図中のバブル名は“修飾語+他動詞+目的語”となっている。他動詞と目的語を用いてプロセス名とする。バブルの各入出力を正しく定義し、修飾語の意味を厳密に定義するため属性を導入する。

ステップS4では各DF図に対応して1つの構文規則を定義する。DF図の名前が構文規則の左辺記号となり、バブルの名前が右辺記号となる。プロセスに対して導入した属性の評価方法を意味規則として定義する。

5.2 属性文法による記述例

変換手順S1～S4に従って、図6のDF図から属性文法によるSA法の形式的記述を作成する。最終的に求まる記述を図10に示す。

ステップS1…図6のDF図より、目的語 Model, DFD, Diagram, Bubble を求め、これらをプロダクトと定める。目的語 Model の修飾語には Phys. (物理的な)と Log. (論理的な)があり、モデルの種類を区別している。それらを表すために属性 Kind を導入し、属性 Kind の値は Phys. または Log. とする。こうして表1に示すプロダクトと属性の一覧表を作成する。

図10の最初の部分でプロダクトの属性を宣言している。また、このプロダクトと属性を用いて各データフローを記述し直す(表2参照)。

ステップS2…表1中の各プロダクトの構成に基づいて、プロダクトに対する構文規則を決める。例えば、プロダクト Model に対し図10の①に示す構文規則を

表1 プロダクトと属性

Product Name	Attribute	
	Name	Type
Model	Contained	2{"Context_diagram", "Log_file", "I/F_boundary"}
	Kind	{"Phys.", "Log."}
	Fully_expanded	{TRUE, FALSE}
DFD		
Diagram		
Bubble	Partitionable	{TRUE, FALSE}
ID		
MS		
Doc.		

表2 データフローとプロダクトの対応

Data Flow No.	Product Name	Semantic Restrictions
(5)	Model	"Context_diagram" ∈ contained
(1)(4)(6)(7)(13)	Model	kind == "Phys."
(8)(9)	Model	kind == "Log." && "Log_files" ∈ contained
(2)(3)(10)	Model	kind == "Log."
(11)	Model	kind == "Log." && Fully_expanded == TRUE
(12)	Model	"I/F_boundary" ∈ contained
(14)	Bubble	partitionable == TRUE

定義する。これに対応し、3つの属性 contained, kind, fully_expanded の評価法を図10の②に示す意味規則として定義する。

ステップS3…図6のDF図の場合、各データフロー名には修飾語がないので、データフロー名はそのままプロセス名となる。入出力を表すために、属性Input, Output, Input1, Input2, Input3を導入する。属性のタイプはデータフローと対応するプロダクトとする。表2に基づいて、表3に示すプロセスと属性の一覧表を作成する。これらの属性は図10の Process Attributesの部分で宣言されている。

ステップS4…各DF図を1つの構文規則へ変換する。例えば、図6(a)のDF図は図10の③に示す構文規則となっている。バブルの入出力関係は意味規則(図10の⑤)として定義する。適用条件(図10の④)では表2で示した各データフローの意味上の制限を表している。

6. 検証の試み

ここでは検証の問題をSA法の属性文法による記述そのものの正当性(6.1)と、その記述が正しいと仮定してSA法を適用する時の正当性(6.2)に分けて議論する。

6.1 SA法自身の正当性

属性文法による形式的記述の正当性として、次の2

表3 プロセスと属性

Process Name	Attribute	
	Name	Type
Structured analysis, Create model	Input	Doc. or Model
	Output	Model
Partition domain, Expand DFD, Decompose files, Eliminate proc. characs., Set out man-machine I/F, Add implementation dependent features	Input	Model
	Output	Model
Identify Domain	Input	Doc.
	Output	Model
Create leveled DFD	Input	DFD
	Output	DFD
Create DD	Input1	ID
	Input2	DFD
	Output	ID
Create MS	Input1	MS
	Input2	DFD
	Input3	ID
	Output	MS
Select partitionable bubble	Input	DFD
	Output	Bubble
Create diagram	Input	Bubble
	Output	Diagram
Add diagram into DFD	Input1	DFD
	Input2	Diagram
	Output	DFD

つが考えられる。

①SA法の従来の記述の中でガイドラインと解釈された部分(あいまいな記述, 脱落していた記述)が正しく修正されただか。

②属性文法による記述を求める過程で、不注意による誤りが入り込んでいないか。

具体的に①、②の正しさを調べる1つの方法は、構文規則の不備(非終端記号が構文規則の左辺に1度も現われていない)が生じていないことを確認することである。別の方法としては、属性値の評価が行えるかどうかを調べることである。①に関しては、従来の記述でDDとMSに関する記述が脱落していることが多く確認された。

6.2 SA法の適用の正当性

ここでは、属性文法によるSA法の記述(図10)に従って、電報解析問題⁽⁴⁾を解いてみる。そのDF図を作成すると、詳細な手順は省くが、図3が得られる。

SA法を実際のソフトウェア開発に適用する場合、次の2つの正当性が問題となる。

```

/* Product Attributes */
*contained : SET;
*kind : {"Log","Phys."};
*partitionable, *fully_expanded, *marked : BOOL;
*level : INT;

/* Product Description */
Doc ::= STRING
Model ::= Set(DFD,DD,MS)
{
  Model.contained := DFD.contained;
  Model.kind := DFD.kind;
  Model.fully_expanded := DFD.fully_expanded;
}
DFD ::= Diagrams
{
  DFD.contained := Diagrams.contained;
  DFD.kind := Diagrams.kind;
  DFD.fully_expanded :=
    ((Diagrams.level == 1) ? TRUE : FALSE);
}
Diagrams ::= Set(Diagram, Diagrams)
{
  If ("Context_diagram" ∈ Diagram.contained)
    or ("Context_diagram" ∈ Diagrams[2].contained))
    Insert("Context_diagram",Diagrams[1].contained);
  If ("Log_file" ∈ Diagram.contained)
    && ("Log_file" ∈ Diagrams[2].contained))
    Insert("Log_file",Diagrams[1].contained);
  If ("I/F_boundary" ∈ Diagram.contained)
    or ("I/F_boundary" ∈ Diagrams[2].contained))
    Insert("I/F_boundary",Diagrams[1].contained);
  Diagrams[1].kind := (((Diagram.kind == "Log.") &&
    (Diagrams[2].kind == "Log.") ? "Log." : "phys.");
  Diagrams[1].level := MAX(Diagram.level, Diagrams[2].level);
}
Diagrams ::= ε
{
  Insert("Log_file" ∈ Diagrams.contained);
  Diagrams.kind := "Log.";
  Diagrams.level := 0;
}
Diagram ::= Set(Bubbles,Files,Flows)
{
  If (Bubbles.level == 0)
    Insert("Context_diagram",Diagram.contained);
  If (Files.kind == "Log.")
    Insert("Log_file",Diagram.contained);
  If (Bubbles.marked == TRUE)
    Insert("I/F_boundary",Diagram.contained);
  Diagrams.kind := ((Bubble.kind == "Log.") &&
    (Files.kind == "Log.") ? "Log." : "phys.");
  Diagrams.level := Bubbles.level;
}
Bubbles ::= Set(Bubble,Bubbles)
{
  Bubbles[1].level := Bubble.level;
  Bubbles[1].marked := Bubble.marked or Bubble[2].marked;
  Bubbles[1].kind := Bubble.kind and Bubble[2].kind;
}
Bubbles ::= Bubble
{
  Bubbles.level := Bubble.level;
  Bubbles.marked := Bubble.marked;
  Bubbles.kind := Bubble.kind;
}
Bubble ::= Contains(Circle,Bubble_identifier)
{
  Bubble.level := Bubble_identifier.level;
  Bubble.kind := Bubble_identifier.kind;
  Assign(Bubble.marked);
  Assign(Bubble.partitionable);
}
Bubble_identifier ::= Follows(Bubble_name,Bubble_no)
{
  Bubble_identifier.level := Bubble_no.level;
  Bubble_identifier.kind := Bubble_name.kind;
}
Bubble_no ::= Follows(INT,Int_list)
{
  Bubble_no.level := 1 + Int_list.level;
}
Bubble_no ::= ε
{
  Bubble_no.level := 0;
}
Int_list ::= Follows(" ",INT,Int_list)
{
  Int_list[1].level := 1 + Int_list[2].level;
}
Int_list ::= ε
{
  Int_list.level := 0;
}

/* Process Attributes */
Structured_analysis.input : Doc;
Create_model.input, Identify_domain.input : {Doc, Model};
Create_model.output, Identify_domain.output : Model;
Expand_DFD.input, Expand_DFD.output : Model;
Decompose_files.input, Decompose_files.output : Model;

Eliminate_pro_Characs.input : Model;
Eliminate_pro_Characs.output : Model;
Set_out_man_machine_I/F.input : Model;
Set_out_man_machine_I/F.output : Model;
Add_implementation_dependent_features.input : Model;
Add_implementation_dependent_features.output : Model;

Create_leveled_DFD.input, Create_leveled_DFD.output : DFD;
Create_DD.input1, Create_DD.output, Create_MS.input3 : DD;
Create_MS.input2, Create_DD.input2 : DFD;
Create_MS.input1, Create_MS.output : MS;
Select_partitionable_bubble.input : DFD;
Add_diagram_into_DFD.input1 : DFD;
Select_partitionable_bubble.output, Create_diagram.input : Bubble;
Create_diagram.output, Add_diagram_into_DFD.input2 : Diagram;

/* Process Description */
Structured_analysis ::=
  Create_model Create_model Create_model Create_model
  where (Kind(Create_model[1].output) == "Phys."
    && Kind(Create_model[2].output) == "Log."
    && Kind(Create_model[3].output) == "Log."
    && Kind(Create_model[4].output) == "Phys.")
  {
    Create_model[1].input := Structured_analysis.input;
    Create_model[2].input := Create_model[1].output;
    Create_model[3].input := Create_model[2].output;
    Create_model[4].input := Create_model[3].output;
    Structured_analysis.input := Create_model[4].output;
  }
Create_model ::= Identify_domain Partition_domain
  where (Is(Create_model.input,Doc)
    && ("Context_diagram" ∈
      Contained(Identify_domain.output)))
  {
    Identify_domain.input := Create_model.input;
    Partition_domain.input := Identify_domain.output;
    Create_model.output := Partition_domain.output;
  }
Create_model ::= Expand_DFD Set_out_man_machine_I/F
  Add_implementation_dependent_features
  where (Is(Create_model.input,Model)
    && (Kind(Create_model.input) == "Log.")
    && (Kind(Expand_DFD.output) == "Log.")
    && (Fully_expanded(Expand_DFD.output) == TRUE)
    && ("I/F boundary" ∈ Contained(Decompose_files.output))
    && (Kind(Add_implementation_dependent_features.output)
      == "Phys."))
  {
    Expand_DFD.input := Create_model.input;
    Set_out_man_machine_I/F.input := Expand_DFD.output;
    Add_implementation_dependent_features.input :=
      Set_out_man_machine_I/F.output;
    Create_model.output :=
      Add_implementation_dependent_features.output;
  }
Create_model ::=
  Expand_DFD Decompose_files Eliminate_pro_Characs
  where (Is(Create_model.input,Model)
    && (Kind(Create_model.input) == "Phys.")
    && (Kind(Expand_DFD.output) == "Phys.")
    && ("Log_files" ∈ Contained(Decompose_files.output))
    && (Kind(Decompose_files.output) == "Phys.")
    && (Kind(Eliminate_pro_Characs.output) == "Phys."))
  {
    Expand_DFD.input := Create_model.input;
    Decompose_files.input := Expand_DFD.output;
    Eliminate_pro_Characs.input := Decompose_files.output;
    Create_model.output := Eliminate_pro_Characs.output;
  }
Partition_domain ::= Create_leveled_DFD Create_DD Create_MS
  {
    Create_leveled_DFD.input :=
      DFD_part(Partition_domain.input);
    Create_DD.input1 := DD_part(Partition_domain.input);
    Create_DD.input2 := Create_leveled_DFD.output;
    Create_MS.input1 := MS_part(Partition_domain.input);
    Create_MS.input2 := Create_leveled_DFD.output;
    Create_MS.input3 := Create_DD.output;
    Partition_domain.output :=
      Set_of(Create_leveled_DFD.output,
        Create_DD.output,Create_MS.output);
  }
Create_leveled_DFD ::= Select_partitionable_bubble
  Create_diagram Add_diagram_into_DFD
  where (partitionable_bubble.output
    == TRUE)
  {
    Select_partitionable_bubble.input :=
      Create_leveled_DFD.input;
    Create_diagram.input := Select_partitionable_bubble.output;
    Add_diagram_into_DFD.input1 := Create_leveled_DFD.input;
    Add_diagram_into_DFD.input2 := Create_diagram.output;
    Create_leveled_DFD.output := Add_diagram_into_DFD.output;
  }

```

図10 属性文法によるSA法(図6)の記述

① 水平方向の正当性…各DF図内で、各バブルに対する入力データフローと出力データフロー(すなわち、各プロセスの入力プロダクトと出力プロダクト)が矛盾を起こしていないか。

② 垂直方向の正当性…DF図間で(正確には、分解されたバブルとDF図の間で)機能的に矛盾を起こしていないか。

図3のDF図に関しては、この記述のレベルでは矛盾を起こしていない。厳密に言えば、属性の評価まで実行して正当性を確認する必要がある。

7. むすび

本報告では、SA法を属性文法によって形式的に記述する試みについて述べた。形式的記述はSA法に基づいたソフトウェア開発におけるプロセスとプロダクトの記述から構成されている。作成手順としては、DF図を用いてSA法を記述しておいて、そのDF図から属性文法による記述を生成するという方法をとった。

現在のところ、DF図によるSA法の記述を完成し、属性文法による形式的記述を作成中である。記述の中では82個のプロセスと35個のプロダクトを用いている。82個のプロセスの中で、46個が自動的に実行可能となっている。

今後の重要な課題としては形式的検証についての考察がある。SA法の形式的記述に基づいて実際のシステム開発を試みることによって、記述内容そのものの妥当性について考察することも重要である。これらの考察に基づいて、SA法を一貫して支援する開発環境を作成する予定である。

謝辞 本研究を進めるにあたり、日頃御討論頂く大阪大学基礎工学部情報工学科鳥居研究室の諸氏に感謝致します。更に、本報告の作成において御協力頂いた同研究室の松本健一氏、楠本真二氏に感謝致します。

参考文献

- [1] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R.: "Structured Programming", Academic Press(1972).
- [2] Demarco, T.: "Structured Analysis and System Specification", Yourdon Press(1979).
- [3] 馮, 杉山, 藤井, 鳥居: "共通属性をもつ属性文法とそのPROLOGによる処理系", 信学論, Vol. J70-D, No. 7, pp. 1311-1319(1987).
- [4] Henderson, P. and Snowdon, R.: "An experiment in structured programming", BIT, Vol. 12, pp. 38-53 (1972).
- [5] Kaiser, G.E. and Feiler, P.H.: "An architecture for intelligent assistance in software development", Proc. of 9th ICSE, pp. 180-188 (1987).
- [6] 片山, 生田: "階層的・関数的アプローチによるソフトウェア設計プロセスの記述", 日本ソフトウェア科学会第5回大会論文集(1988).
- [7] Kishida, K. et al.: "SDA: A novel approach to software environment design and construction", Proc. of 10th ICSE, pp. 69-79(1988).
- [8] Knuth, D.E.: "Semantics of context-free languages", Math. Syst. Theory, Vol. 2, No. 2, pp. 127-145(1968).
- [9] Jackson, M.A.: "System Development", Prentice-Hall(1983).
- [10] 松本 吉弘: "ソフトウェアに対する要求の形成", 情報処理学会誌, Vol. 28, No. 7, pp. 853-861 (1987).
- [11] 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資料 58-1 (1988).
- [12] 大野 豊: "ソフトウェア工学の背景と展望", 情報処理学会誌, Vol. 28, No. 7, pp. 845-852 (1987).
- [13] Osterweil, L.: "Software processes are software too", Proc. of 9th ICSE, pp. 2-13(1987).
- [14] Williams, L.G.: "Software process modeling: a behavioral approach", Proc. of 10th ICSE, pp. 174-186(1988).
- [15] Yourdon, E. and Constantine: "Structured Design", Yourdon Press(1978).