ソフトウェア工学 98－4
（1994．5．25）

プログラム理解のための仕様記述の分析

関本 理佳　　海尻 賢二

信州大学 工学部 情報工学科
〒380 長野県長野市若里 500
E-mail: rika@cs.shinshu-u.ac.jp

あらまし　効果的な保守を行なうためには、まずそのソフトウェアを理解しなければならない。本研究では、対象プログラムから、形式的な仕様を抽出することによりプログラムの理解を支援するリバースエンジニアリングツールの開発を目指している。特に、ドメインを限定することにより、プログラムの構造的情報だけでなく、アプリケーション固有の知識等を利用する。このためには、ドメイン知識の分析が重要となってくる。そこで、UNIX のテキストユーティリティを対象にして、ソースプログラム、仕様記述の分析を行なった。本稿では、この分析に基づいて構築したプログラムプラン、仕様プランについて、さらに、これらの知識を利用した仕様記述生成の流れについて述べる。

和文キーワード
リバース・エンジニアリング, 形式的仕様記述言語, プログラム理解, ドメイン知識, プラン

An analysis of specification for program understanding

Rika SEKIMOTO　　Kenji KAIJIRI

Faculty of Engineering, Shinshu Univsity
500 wakasato nagano, 380 Japan

Abstract　The program understanding is a key aspect of software maintenance process. This paper describes a reverse engineering tool that extract a formal specification from source programs. This system uses not only the structural information of programs but also domain specific knowledge. For the semantic view point, the analysis of the domain knowledge becomes important. We restricted the target domain to the textutility of UNIX , and analyzed the programs and the specification that we wrote. In this paper, we discuss the knowledges for program understanding, the program plan and the specification plan. In addition, we will describe how to generate the specification using Z.

英文 key words
reverse engineering, formal specification, program understanding, domain specific knowledge, plan

1 はじめに

近年ソフトウェアのライフサイクルにおいて、保守の占める労力と時間の割合は増加の一途をたどり、ますます大きな問題となってきている。この主要な原因の一つに、そのシステムの開発保守以前に従事していたプログラマの意図を理解することが難しいということがあげられる [1]。つまり、ソフトウェア保守において何よりも最初に必要なことは、ソフトウェアプログラムを十分に理解することで、この理解を支援することにより効果的な保守が可能になると考えられる。我々はこのような観点から、対象プログラムから、そのソフトウェアが何をするのかという仕様レベルの情報を抽出することによりプログラム理解を支援する、リバースエンジニアリングツールについての研究を行なっている [11]。

従来のリバースエンジニアリングツールに関する研究では、構造的な観点に焦点をあてたものが多い。例えば、Cプログラムからクロスレファレンス情報を抽出し、構造化設計チャートなどに変換する CIA(C Information Abstraction System) [2] や、COBOL,PL/1 のソースプログラムから構造化分析／構造化設計の際に用いられる各種ダイアグラムを生成することにより、プログラムの理解を支援する RE/Cycle [3] などがある。また、COBOL プログラムを抽象化して Z で表現するという研究 [4] も報告されているが、ここでも意味的な抽象化は行なっておらず、構造的な情報のみを利用している。

本研究では、プログラマの意図の認識という観点からリバース・エンジニアリングを捉え、より上位の仕様のレベルに焦点をあてている。このような情報を抽出するためには、問題領域に依存した情報が必要となる。そこで、アプリケーションドメインを限定することにより、構造的情報だけでなく、アプリケーション固有の知識も利用してプログラムの認識を行なう。また、この認識結果をフォーマルな形式で抽出することにより、仕様の再利用や、さらにはフォワード過程への直結を目指す。

本稿では、対象ドメインを UNIX のテキストユーティリティとし、仕様記述言語には Z を用いた。以下 2 節で、リバース・エンジニアリングの問題について考察し、3 節で仕様記述言語 Z について説明する。次に 4 節では、テキストユーティリティの特徴およびその仕様記述について述べる。5 節では、ソースコード、仕様記述の分析から得られたプログラムプラン、仕様プランの分類とその知識表現を示し、6 節でこれらの知識を利用した仕様記述生成の方法について述べる。

2 リバース・エンジニアリング

リバース・エンジニアリング (reverse engineering)[5] の主な目的は、保守や再利用のために、システムの全体的な理解を支援することである。このために対象システムを分析して、以下のことを行なう。

- システムの構成要素とそれらの相互関係を明らかにする。
- システムを現在とは異なる形態もしくは、現在より抽象度の高いレベルで表現する。

リバース・エンジニアリングには、いくつかの技術的なレベルがある。比較的単純なものには、分析する元のソースコードと抽象的概念としては相対的に同じレベルの表現で、制御やデータの流れを視覚化する再文書化と呼ばれる技術がある。それよりも高いレベルの技術を要するものとして設計復元、関数抽象化、業務規則の抽出などがある。

本研究では、リバース・エンジニアリングを、単にドキュメント化などのソフトウェアをわかりやすく眺めるための視覚情報を作り出す過程とみなすのではなく、再利用可能なソフトウェア要素を抽出する過程または技術とみなす。このような考えのもとに、ソフトウェア開発におけるより上位のレベルである仕様に焦点をあて、既存のソフトウェアからその仕様を逆生成することを目指す。特に、このような意味的な抽象化を行なうために、アプリケーションドメインを限定し、リバースエンジニアリングへのドメイン知識の応用についても検討する。

3 仕様記述言語 Z

自然言語などによる仕様はわかりやすいが、曖昧さ、過不足、不整合などを伴いやすい。また、仕様の変更による保管・管理なども大変になる。形式的仕様 (formal specification) では、これらの問題を解決する有効な手段が与えられる。このような形式性に対する認識が高まるにつれて、さまざまな形式的仕様記述言語が開発されてきている。本研究では、現実世界のモデル化のために適した Z[6, 7] という言語に着目した。

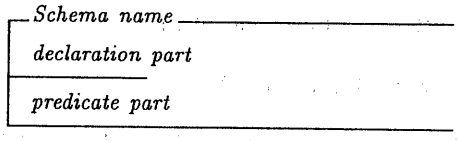
Z はオックスフォード大学で開発された汎用な形式的仕様記述言語である。Z を用いた仕様化では、仕様化対象のシステムの状態およびシステムに作用するオペレーションを、述語論理と集合論に基づいて定義していく。

状態とオペレーションはスキーマ (schema) と呼ばれる構造を用いて記述される。スキーマには、

- システムの可能な状態空間を表す状態スキーマ
- 対応するオペレーションの実行と、それによるシステムの状態変化を表すオペレーションスキーマ

とがある。状態スキーマは、システムの初期状態を表す初期状態スキーマを含む。これらのスキーマを用いてシステムをモデル化することが Z の特徴となっている。

スキーマは、以下のように記述される。



宣言部 (declaration part) では、< 変数 : 型 > のように変数の宣言を行なう。述語部 (predicate part) では、宣言部で宣言された変数に対する制約を規定する。変数は通常状態を表し、入力変数は変数名の最後に ? を、出力変数は変数名の最後に ! をつけて表す。また、オペレーションの実行によって状態変化した後の状態を表すのに ' が使われる。宣言部では他のスキーマ名を参照することにより、参照スキーマの値を継承できる。先頭に Δをつけてスキーマ名を宣言すると、そのスキーマで記述された状態の状態変化を表す。先頭に Ξ をつけた場合は、状態変化のないことを表す。

4 テキストユーティリティ

本研究では、実用規模のソフトウェアであること、コンパクトなプログラムの集まりであること、という二つの観点から UNIX のテキストユーティリティを対象ドメインとした。また、ここで扱ったプログラムは、C 言語で作成された GNU 版のものである。

4.1 テキストユーティリティの特徴

テキストユーティリティのパッケージは、ファイルの内容に対する種々の取り扱いについてのコマンドの集まりである。例えば、与えられたファイルの行数、単語数、文字数を表示する wc コマンドや、ファイルの連結と出力を行なう cat コマンド、ファイルの最初の数行を表示する head コマンドなどがある。

以下にその特徴を述べる。

- コマンド引数の利用

  各コマンドは多くのコマンド引数を持ち、これを利用して、ひとつのコマンドで複数の異なった処理が行なえるようになっている。つまり、コマンド引数の値、種類の組合せにより種々の機能の中から指定された特定の機能のみを実現するように設計されている。

  例えば、wc コマンドではファイルの行数、単語数、文字数のみを表示する 3 つのオプションがある。また、head コマンドでは lineN オプションにより表示する行数を指定できる。

- 入力対象となるファイルの制約

  上述した wc, cat, head などのコマンドは、複数のファイルを引数としてとることができる。しかしパッケージの中には、2 つのソート済みファイルを相互参照して処理を行なう comm コマンド（2 つのファイルの同一行を表示）や、一つのファイルのみしか扱うことのできない split コマンド（ファイルの分割）等もある。

- ライブラリ関数の利用

  Cによるプログラムの特徴として、ライブラリ関数の利用があげられる。テキストユーティリティにおいて使用されるライブラリ関数の情報は、プログラムを抽象化するのに必要である。

4.2 Zによる仕様記述

既存のコンピュータシステムの仕様記述を、Z言語で記述してみるという試みがいくつも行なわれてきている。例えば、ファイルの open, close などを行なう Filing System [8] がある。我々は、テキストユーティリティにおける仕様に関する知識を得るために、オンライン・マニュアルを参照して、各コマンドのZ言語による仕様記述を作成した。

コマンドの対象はバイトストリームであるので、基本型を [BYTE] とする。FILE は基本型 BYTE の sequence で表すことができる。以下に、wc コマンドのバイト数のカウント処理についての仕様記述例を示す。

WcBytes

$input? : FILE$

$byte\_count! : \mathbb{N}$

$byte\_count! = \#input?$

5 ドメイン知識の分析

実用規模のプログラムに対して、構造的観点からのみではなく、意味的観点を考慮したプログラム理解の支援を行なうためには、アプリケーション固有の知識が重要となる。そこで我々は、このような知識を獲得するために、アプリケーションドメインを GNU のテキストユーティリティに限定し、そのソースコード, 仕様記述の分析を行なった。

本節では、この分析に基づき体系化したプログラムプラン、仕様プランについて、それぞれの分類とその知識表現について述べる。

5.1 プログラムプラン

ここでのプログラムプランとは『プログラムにおいて頻繁に使われる標準的な記法』である [10]。非常に基本的なもの（例えば、2変数の交換）もあれば、データ構造に密接に関連したもの（例えば、バッファの再割当）、さらにはアルゴリズム的なもの（例えば、分類）もある。我々はプログラムプランを、制御プラン、基本プラン、複合プラン、メタプランの4種類に分類した。熟練したプログラムマは言語の基本命令だけでなく、このようなプランの知識をたくさんもっており、これを組み合わせてプログラムを作成していると考えられる。

そこで、プログラムを理解するためには、そのプログラムがどの様なプログラムプランの組み合わせにより構成されているかを認識する必要がある。本研究では、このような考え方に基づき、まずプログラムをプログラムプランの複合体として認識する。そして、これをプログラムプラン木と呼ぶ木構造により表現する。

5.1.1 プログラムプランの分類

プログラムプランは、以下の4種類のプランに分類できる。

- 制御プラン (Control plan)
- 基本プラン (Basic plan)
- 複合プラン (Compound plan)
- メタプラン (Meta plan)

制御プランとは、プログラム中に現れる基本的な制御構造のパターンで、selection plan, subprogram plan, loop plan, assignment plan 等から構成されている。制御プランはプログラム断片、および他のプランの構成子としての役割を持つ。

認識のベースをなるのが基本プランである。基本プランには、動作プラン (Action plan) とデータプラン (Data plan) がある。動作プランは、制御プラン、テンプレートから構成され、プログラム中の基本動作を表すプランである。データプランは、特定のアプリケー

ションにおいて特別に意味づけされたデータ構造を表現している。

成分として他のプログラムプランを含むプランを、複合プランと呼ぶ。

メタプランは、いくつかプランに共通な親プランである。それ自身は本体を持たず、プログラムの処理やアルゴリズムのバリエーションを記述するために利用する。

5.1.2 プログラムプランの表現

標準入力からデータを読み込むためのプログラムプラン (data-read-plan) の例を以下に示す。

```
data-read-plan(?data)
   CLASS read-plan
   DESCRIPTION data read from
               standard input media
   BODY {  DESCRIPTION none
           BODY procedure-call-plan
                name:template(scanf)
                parameter(&?data)
           BODY-CONSTRAINTS none }
   PARAMETER-CONSTRAINTS variable(?data)
```

プログラムプランの本体 (BODY) は、テンプレートおよびプランの組合せとして表現される。ここでの基本は順次的結合であるが、制御プランで指定することにより、任意の表現ができる。また、パラメータもしくはプラン要素に対するデータ型の指定や、プラン要素の相互関係が指定されている場合、その種々の制約を BODY-CONSTRAINS に記述する。

各プログラムプランは BODY の記述の他に、プランのカテゴリとしての階層を表す CLASS、自然言語による内容を記述する DESCRIPTION、プランパラメータに対する制約を記述する PARAMETER-CONSTRAINS、により表現される。

5.2 仕様プラン

テキストユーティリティの Z 言語による仕様記述を分析して、共通に現れるスキーマやパターンを仕様プランとして知識ベース化した。

5.2.1 仕様プランの分類

仕様プランは、以下の 2 種類から構成される。

- 基本仕様プラン (Basic specification plan)
- 汎用仕様プラン (Generic specification plan)

基本的な動作に関するスキーマ知識を表現したものを、基本仕様プランと呼ぶ。基本仕様プランは、それ自身のみで完成したスキーマを生成することができる。

汎用仕様プランとは、テキスト・ユーティリティの仕様記述に現れる共通なパターンを一般化したものである。汎用仕様プランは、スキーマを生成するための宣言部、述語部に関する知識に、属性のみが決められているプレースホルダ部を含む。そして、このプレースホルダに制約条件を満たす仕様の断片が挿入されて始めて、スキーマを生成することができる。例えば、ファイル単位で入力して何らかの処理を行なう each-file-process-spec や、行単位で入力して何らかの処理を行なう line-by-line-process-spec などがある。

5.2.2 仕様プランの表現

あるシーケンスが入力された時に、その要素数を求めるための仕様プラン (count-spec) の例を以下に示す。

```
count-spec
   DESCRIPTION count number of a
               input sequence
   BODY {  SCHEMA-NAME Count
           DECLARATION
                 input? : seq X
                 output! : N
           PREDICATE
                 output! = # input? }
```

仕様プランの本体 (BODY) には、スキーマを生成するためのテンプレートが記述される。SCHEMA-NAME においてスキーマ名を示し、宣言部、述語部の情報を、それぞれ DECLARATION、 PREDICATE に記述する。汎用仕様プランでは、プレースホルダ部に対する制約が BODY-CONSTRAINS に記述される。

6 プログラム認識に基づく仕様化

プログラム認識による仕様記述生成の流れを図 1 に示す。

本研究ではプログラム認識を、具象プログラムから抽象プログラム、具象仕様、そして抽象仕様を生成す

るという3つのプロセスに分割して考えている。ここでの具象プログラムとは、C言語による実際のプログラムのことである。まず、具象プログラムにおけるプログラムプランを認識する。この認識に基づき、プログラムプラン木により抽象プログラムを表現し、これからZ言語を用いた仕様記述を生成する。

ここでは、抽象プログラム化プロセスによるプログラムプラン木の構築と、具象仕様を生成する仕様化プロセスについて述べる。

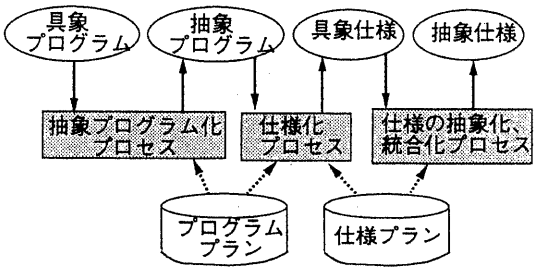


図 1: 仕様記述生成の流れ

6.1 プログラムプラン木の構築

6.1.1 フローグラフへの変換

現在C言語のプログラムを対象としているため、コンパイルマクロ、インクルード (# include)、副作用のある種々の表現といった認識の障害となる問題がある。そこでまず、プログラムプランの認識を行なう前に、構文解析およびデータフロー解析により対象プログラムの基本的な制御構造を認識する。これにより、プログラムにおける制御構造の標準化および副作用のある表現の簡単化を行ない、制御プランをノードとする属性付きフローグラフへ変換する。

6.1.2 機能分割

テキストユーティリティにおける各コマンドは、たくさんのコマンド引数を持つため、ひとつのコマンドを利用して複数の異なった処理が行なえるようになっている。つまり、コマンド引数の値、種類の組合せにより種々の機能の中から指定された特定の機能のみを実現するように設計されている。プログラム理解の支援を考える時、この指定された機能のプログラムを見るという視点も重要になる。そこで本研究では、各コマンドのソースコード全てを対象とするのではなく、まずこのコマンド引数を指定した機能単位のプログラムについての仕様生成を考えている。

この機能分割を実現するために、パラメータスライスという前向きスライスの一種を考える [9, 12]。パラメータスライスとは、コマンドパラメータで与えられた引数の値、型、値域によって実行され得る実行パスであり、これを求めることにより各コマンドの単一機能のプログラムを取り出すことができる。

6.1.3 プログラムプランの認識

機能分割されたプログラムに対し、関数ごとにプログラムプランの認識を行なっていく。各関数はプログラムプランの複合体として認識され、これにより以下のようなプログラムプラン木が作られる。

```
*** head -lines100 head.c ***
   main:
     command-arg-plan
     each-file-process-plan
       process:procedure-call-plan
          name:template(head-file)
          parameter:template(argv[optind],number)
     program-exit-plan

   head-file(filename,number):
     file-open-plan
       descriptor:template(fd)
       file:template(filename)
     procedure-call-plan
       name:template(head)
       parameter:template(filename,fd,number)
     file-close-plan
       descriptor:template(fd)

   head(filename,fd,number):
     return-plan
        value:function-call-plan
           name:template(head-lines)
           parameter:template(filename,fd,number)

   head-lines(filename,fd,lines_to_write):
      conditional-loop-plan
           condition:?length
           body:bytes-read-plan
                out-length-set-plan
                bytes-write-plan
```

これは、指定されたファイルの最初の数行を表示するheadコマンドの、出力行数を指定したlineNオプションに対する認識結果である。

6.2 仕様化プロセス

仕様化プロセスでは、プログラムプラン木を入力として受り、Z言語による仕様を生成する。

まず仕様化プロセスでは、入力されたプログラムプラン木の各プログラムプランおよびプログラムプランの組合せから、それに対応する仕様プランを検索する。そして次に、各仕様プランの知識からスキーマを生成し、それらを結合する。

6.2.1 仕様プランの検索

入力されたプログラムプラン木におけるプランの組合せに対して、その制約を満たす仕様プランが存在するか調べる。次に、まだ認識されていないプログラムプランに対して、DISCRIPTIONの情報を用いてそれに対応した仕様プランを見つける。

またここで、スキーマの結合知識となるプログラムプラン、仕様生成の際に抽象化されてしまうプログラムプランの認識も行なう。

6.2.2 スキーマの生成

認識された各仕様プランのSCHEMA-NAME、DECLARATION、PREDICATEの情報からスキーマを生成する。汎用仕様プランについては、その仕様プラン自身の知識と、それに続く仕様プランの知識、またはスキーマの結合知識を用いてスキーマを生成する。

6.2.3 スキーマの結合

仕様プランの情報から生成した各スキーマを結合し、入力されたプログラムプラン木に対する仕様を生成する。

スキーマ間の結合は、プログラムプラン木におけるprocedure–call–planやfunction–call–planなどの制御プランの情報によるSchema conjunction ($\wedge$)と、汎用仕様プランの制約条件によるSequential composition ($\fatsemi$)を用いて記述される。

以下にheadコマンドlineNオプションのプログラムプラン木から生成される仕様記述例を示す。

$$HeadCommand \mathrel{\widehat=} GetFilename \fatsemi (Open \wedge HeadLine \wedge Close)$$

GetFilename

$$argv : \mathrm{seq}(\mathrm{seq}\, BYTE)$$
$$argc : \mathbb{N}$$
$$name! : NAME$$
$$\exists i : 2 .. argc \bullet name! = argv[i]$$

Open

$$FS$$
$$FS'$$
$$name? : NAME$$
$$cid! : CID$$
$$fid, fid' : FID$$
$$name? \in \mathrm{dom}\, nstore$$
$$fid = fid' = nstore(name?)$$
$$fstore' = fstore$$
$$(\exists\, CHAIN' \bullet$$
$$\quad posn' = 0$$
$$\quad fid' = fid$$
$$\quad cstore' = cstore \oplus \{cid! \mapsto \theta CHAIN'\}$$
$$nstore' = nstore$$
$$cid! \notin \mathrm{dom}\, cstore$$

HeadLine

$$fid? : FID$$
$$length? : \mathbb{N}$$
$$output! : FILE$$
$$(let\ infile == (fstore(fid?)\ \underline{cut}\ nl);$$
$$\quad outfile == (output!\ \underline{cut}\ nl) \bullet$$
$$\quad\quad outfile = (1 .. length?) \upharpoonright infile)$$

Close

FS

FS'

$cid? : CID$

$cid? \in \mathrm{dom}\ cstore$

$cstore' = \{cid?\} \lhd cstore$

$fstore' = fstore$

$nstore' = nstore$

7 おわりに

大規模な実用のデータ処理システムの開発、保守を考えた場合、特定の分野での永年の開発経験を持つ、熟練プログラマの専門知識がきわめて有用である。このことからもわかるように、実プログラムを対象とする場合は、ドメイン知識の分析が重要となる。そこで本研究では、対象ドメインをUNIXのテキストユーティリティとし、そのソースコード、仕様記述の分析を行なった。これによりプログラム理解のために必要な知識の体系化を試みた。本稿では、プログラムから仕様を生成するための知識として、プログラムプラン、仕様プランの提案をした。さらに、これらの知識を利用した仕様記述生成の方法について検討した。

これにより、構造的な観点からのみのアプローチでは不可能であった上位の仕様に対するリバースエンジニアリングが可能となる。

しかし、現段階はプログラムから仕様を生成するという全体の流れを検討したにすぎず、仕様プランについては、まだテキストユーティリイの一部のものしか取り扱っていない。そのため、仕様プランの検索知識についても不十分なものになってしまっている。今後は、この検索知識を洗練するとともに、仕様プランを階層的に体系化することにより、抽象度の高い仕様の生成について検討していく予定である。また、各機能に分割した仕様記述をどのような方法で統合するかについても、今後検討していかなければならない。

参考文献

[1] Carma McClure著, ベストCASE研究グループ訳 :ソフトウェア開発と保守の戦略, 共立出版, (1993)

[2] 四野見 秀明他:構造化分析／構造化設計の方法論に基づいたプログラム理解支援ツール, 情報処理学会ソフトウェア工学研究会資料, SE-87-1 (1992)

[3] Y. Chem, M. Y. Nishimoto, and C. V. Ramamoorthy :*The C Information System*, IEEE Trans. on S. E., Vol.SE-16, No.3 (1990)

[4] H. J. van Zuylen :*THE REDO COMPENDIUM*, John Wiley & Sons (1993)

[5] Elliot J.Chikofsky and James H. Cross :*Reverse Engineering and Design Recovery : A Taxonomy*, IEEE Software, Vol.17, No.1 (1990)

[6] J. M. Spivey :*The Z Notation - A Reference Manual*, Prentice Hall (1989)

[7] J. B. Wordsworth: *Software Development with Z*, Addison-Wesley (1992)

[8] C. Morgan and B. Sufrin :*Specification of the UNIX Filing System*, IEEE Trans. on S. E., Vol.SE-10, No.2 (1984)

[9] S. Horwitz, T. Reps, and D. Binkly, :*Interprocedural slicing using dependence graphs*, ACM TOPLAS 12(1) (1990)

[10] 海尻賢二:プログラム認識／理解とその応用, 電子情報通信学会知能情報工学研究会資料,KBSE93-11(1993)

[11] 関本理佳他:*Z*言語によるテキスト・ユーティリティの仕様記述について, 情報処理学会第48回全国大会講演論文集 (分冊5), 3H-6 (1994)

[12] Kenji Kaijiri:*Support of plan library construction*, JCKBSE'94 : Japan-CIS Symposium on Knowledge-Based Software Enginnering (1994)