ソフトウェア開発作業系列の形式的定義と誘導システムの生成

飯田 元[†] 荻原剛志[†]
井上克郎[†] 鳥居宏次[††]

本論文では文脈自由文法でソフトウェア開発における様々な作業の手順を簡潔に定義し，手順に従った作業環境を生成する方法を提案する．この方法によると，比較的自由度の大きな手順を容易かつ簡明に書くことができる．また，記述を元にした処理系を生成する際，通常用いられるようなプログラミング言語等による方法と比べて単純な制御構造で，実現できる．処理系の実現例として，メニューを用いて次に行うべき作業を順次選択し，作業の誘導を行う開発支援環境が記述から生成できる．本論文では開発過程の形式化の手法を理解・比較するための記述対象として提案され広く用いられている共通問題を，記述例としてとりあげ，作業手順の記述およびその支援環境の生成の例を示す．

Formalizing Software Development Activities and Generating Navigation System

Hajimu Iida,[†] Takeshi Ogihara,[†] Katsuro Inoue[†] and Koji Torii[††]

This paper describes a way to define software development process formally. We assume that the development processes would be sequential sentences of activities, and we define the set of the sentences by the context free grammar. By using formal grammar, the sequences of development activities are defined formally so that characteristic and behavior of the development processes become clearer. This paper also describes a method to construct a menu oriented navigation system from the grammar. This system provides menus and guides the developers to perform activities in an appropriate order. The system works with the interpreter of process description language PDL.

1. はじめに

近年，ソフトウェア開発の手順や作業の内容（いわゆる開発過程）等を形式的に記述する試みが進められている[8),10),14),16)]．今まで説明が乏しく理解しにくかったり，あいまいで人によって認識が異なったりするような開発過程を，形式化することによって明確にし，正確に他人に伝えることができるようになることが期待される．また，その開発を支援するソフトウェア開発環境を，開発過程の形式的な記述に基づいて(半)自動的に生成する方法の研究も行われている[2),9),13)]．これらの多くの研究では，手続き型，関数型，論理型等のプログラミング言語を用いて開発過程の形式化を行っている．すなわち，ツールの起動や人間の行う作業などといった，開発過程を構成する基本的な作業を計算機言語における基本構成要素（例えば手続き型言語における演算や関数呼び出し，論理型言語における述語等）と対応させる．そしてそれらの言語が持つ様々な制御機構（if 文やパターンマッチ機構など）を用いて，実行時にこれらの基本要素の評価順序が期待する作業手順になるように記述を作成する．我々もまた同様の目的から，関数型の開発過程記述用言語 PDL (Process Description Language) を設計し，その処理系を作成するとともに[3),13)]，PDL による開発過程の記述を試みてきた[2),5)]．

しかし，現実の開発における複雑な作業手順を詳細に記述するのは容易ではない．また，例えば，代表的な作業手順のみを詳細に記述し，実際にその作業手順のみしか許さないような制約は，開発者の自由度を奪い，逆に開発効率が低下する恐れがある．さらに，これらのプログラミング言語を用いた記述では，基本要素と制御文が一般に混在しており，詳細なレベルの記述になるほど，全体の手順を直観的に把握するのが困

† 大阪大学基礎工学部情報工学科
Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University
†† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Advanced Institute of Science and Technology, Nara

難になろう.

本研究では許される作業の手順のすべてを，容易にかつ直観的に理解できる形で記述することをまず第一の目的とする．そして，それを実行するための細かい制御については記述を変更するのではなく，制約的な条件を付加する形で指示できるようにする．このようにすることで記述が簡潔になり，その開発過程の作業手順が持つ性質を比較的容易に知ることができる．また，他の開発過程との比較なども行うことができる．本研究の第二の目的は，その記述に基づいてソフトウェア開発環境を生成することである．この環境では，開発者自身が必要に応じて手で作業手順を制御でき，開発時に許される作業手順の中から比較的自由に好みの手順を選ぶことができるような支援系を容易に構築することができる.

ここでは，作業手順を定義する手段として文脈自由文法を用い，再帰を含むような開発過程の作業手順を表すことを可能にした．また，後で述べるように，不当な作業手順の導出を省くための簡単な制約規則を各生成規則に付加できるようにした．さらに，この文法表現を基に作業手順の導出を行うような開発支援系を構築する方法についても述べ，階層化されたメニューによって開発者を適切な作業へと誘導する支援システムの構築例を示す．支援システムは文法表現を開発過程記述言語 PDL に変換することによって得られる．このシステムは，文法の開始記号を構文規則に従って展開していくことで作業手順の誘導を行うものであり，構文解析的な手法を伴わないため，すべてのクラスの文脈自由文法について適用が可能である.

この方法によって実際に，ある典型的な，設計／コード変更プロセスの例題[10]について作業の文法表現を記述し，その支援システムを試作した．この例題は開発過程の形式化の共通問題として国際プロセスワークショップにおいて提案され，他にも多くのモデルを用いた記述がなされている[10].

本文の以下では，2章で関連研究についてその問題点を述べ，3章で作業系列の文法による定義方法について，4章ではこの文法を用いた開発支援方法についてそれぞれ述べる．さらに，5章では PDL 処理系を用いた開発支援システムを文法からの生成の例について紹介し，6章でまとめを行う.

2. 関連研究

ソフトウェア開発過程のモデル化の手法としてこれまでに様々な方法が提案されている.

Osterweil らは Ada の拡張言語 Appl/A を用いて開発過程を記述しようと試みている[14),15]．Appl/A は Ada の持つ豊富な記述能力の上に，プロダクトや資源の間の関係を記述できるような拡張が加えられているため，開発過程を記述する上での様々な要求に応えることができる．Appl/A によって，作業の手順やプロダクトの操作が手続き的に表現される．しかし，完全にプログラムの形で記述された開発過程は，その構造や作業の系列の持つ性質がわかりにくい．また，記述から逸脱した作業は行うことができず，すべての例外的事象を考慮した記述を行う必要がある.

Kaiser らは，あらかじめ記述したルールに従って自動的にツールを起動したり，プロダクトに対する操作を行うことができる開発環境 Marvel を設計した[8]．Marvel はプロダクトやツールを管理するためのオブジェクト群と，作業の手順を表現する拡張可能なルール群から構成されている．ルールは前提条件，完了条件，および開発作業からなり，あるルールの条件が満たされたとき，そこに記述された開発作業を表すツールが起動される．したがって，手続き的な記述とは異なり，例外的記述を容易に行え，作業を柔軟に実行できるといった長所がある．しかし，その一方で，作業が具体的にどのように進むのかを記述から予測することは困難である．そのため，開発過程全体の構造が不明確になるといった問題がある.

Appl/A のように純粋に手続き的なプロセス記述では，開発過程の構造を比較的明確に記述できるが，一方で，作業系列の束縛が大きくなる恐れがある．また，Marvel のようなルールベースの記述では記述の容易さ，実行の柔軟さなどが期待できる半面，開発過程の構造や作業の実行系列などを把握しにくいという問題がある．しかし，ソフトウェアの開発において，作業の進捗状況を把握したり，作業に関するデータの収集を行う等といった，いわゆるプロセス・マネジメントを効果的に行うためには，このような作業の流れを明らかにし，制御する必要があろう.

一方，作業の系列を正規表現等で定義する試みはこれまでにも Williams によってなされている[16]．Williams は正規表現に並列動作を表す表記（シャッフルオペレータ）を追加した constrained expression[1]と

呼ばれる表現を用いて作業系列を定義している．しかし，基本プログラムを段階的に拡張してゆく差分開発(incremental development)や，本稿で例として取り上げたトップダウン開発など，再帰的な系列を持つ開発過程については，正規表現よりも文脈自由言語による方が自然に定義可能である．また文献16では具体的な開発支援系の構築については言及されていない．

さらに，類似の試みとして，片山らによる属性文法による開発過程の記述がある[9]．片山らはHFSPという階層的に定義されたプロセスモデルを提案し，その枠組を記述するために属性文法を用いている．ここでは属性文法は生成物の自動生成や管理手順などといった実際の計算メカニズムとして用いられている．

これら，ソフトウェア開発過程をモデル化しようとする従来からの試みにおける問題の一つは，開発過程の持つ様々な要素（作業手順，生成物の管理，資源の割り当てなど）を一つのモデルでは表し切れないということであると考えられる．また，MarvelやHFSPでは記述言語とモデルとが一体化しており，記述が自然に行える一方で，モデルの拡張や変更といったことは考慮していない．一方で，我々の用いているPDLは単純な関数型言語であり，その枠組の上でより複雑なモデルを構築したり，開発過程の持つ特定の側面のみを記述する，といったことを容易に試みることができる．そして，このような特定の側面から単純化したモデルを複数組み合わせることによって，個々のモデルの抽象度を高め，形式的な評価を容易にすることを目指している．

本研究では，作業の系列に特に注目する．そして作業系列を文脈自由文法で記述し，その制限を後で述べる制約条件で与える．この方法では，作業工程のガイドラインとしてのみ文法が用いられる．制約条件は，実際の作業内容を定義するためのものではなく，単に生成された作業系列が満たすべき性質として働く．このことによって，作業系列の見通しの良さと柔軟性との両立を目指している．したがって，Appl/Aにおけるコンパイラ/インタプリタやMarvelにおける推論機構，属性文法における属性評価器などといった計算手段については議論の対象としない．このような機能は，例えばオブジェクト指向データベースなどを基にした，プロダクトサーバ[11]が担うことになり，我々の方法で得られる支援系はその操作の誘導部として機能することになる．

3. 作業系列の構文則による定義

3.1 準　　備

開発過程とは，ソフトウェアを開発するために行われるさまざまな活動のことを指すが，本稿ではソフトウェア開発過程を1人のソフトウェア開発者が1台の計算機（例えばワークステーション）上でソフトウェア開発を行う一連の開発作業（activity，以下では単に作業と呼ぶ）の系列とする．ここでの作業とは，ソフトウェア開発に付随するあらゆる活動であり，一つの作業はさらに細かい作業の集まりとして表される．つまり，ここでは作業（activity）を以下のように定義する．

- 開発過程全体は一つの作業である．
- 各作業は複数のさらに細かい作業の系列として表すことができる．
- 作業の最小単位はある開発ツールの実行や人間の意志決定などに相当する☆．

このとき，開発作業の系列は最小単位の作業の一次元系列であるという仮定をおくことで，これを正規言語や文脈自由言語として表現することができる．開発作業の系列を文法で定義するとは，以下のような対応付けを行った上で，構文則を定めることである．

- 開始記号は開発過程全体を表す．
- 非終端記号は開発過程全体の中の一部の作業系列を表す．
- 終端記号はこれ以上分割できない最小単位の作業を表す．

また，作業の進行は開始記号を生成規則に従って左・深さ優先で展開してゆくことに対応する．

3.2 文法表現による系列定義

ここでは，まずはじめにWilliams[16]などによって比較的広く用いられている正規表現による定義の例を示し，次に，本研究で提案する，文脈自由文法による定義方法を示す．

（1） 正規表現を用いた定義

例えば，*Edit*, *Compile*, *Link* といった一連の作業を表す場合を考える．この時，*Compile* や *Link* でエラーが生じた場合，作業の繰り返しが生じる．ただし，*Compile* や *Link* を単独でやり直すことはない．した

☆ 分割の細かさは，開発過程をモデル化する際に，対象としている開発過程や注目点—例えば，開発過程の全体的な流れに注目するのか，ある一部の設計活動におけるツール操作の手順に注目するのか等—によって記述者が決定する．

がって，この作業系列は正規表現

$((Edit^{+}Compile)^{+}Link)^{+}$

で表される．ここで，定義された表現から生成可能な系列，例えば *Edit*−*Compile*−*Link* や *Edit*−*Edit*−*Comile*−*Edit*−*Compile*−*Link* などのような系列を正しい作業系列とし，*Compile*−*Edit*−*Link* や *Edit*−*Compile*−*Compile*−*Link* などのようにもとの表現からは生成されない系列を不正な作業系列であるとする．

別の例として，ウォーターフォールモデルによる開発プロセスを考える．一般的なウォーターフォールモデルでは要求分析を行った後，設計を行い，コーディング，テスト，結合を行うといったように，各段階の作業が逐次的に行われることになっている．しかし，実際には個々の作業や上流からの一連の作業のやり直しが生じるため，作業系列は正規表現を用いて，例えば次のように表される．

$((((Analysis^{+}Design^{+})^{+}Coding^{+})^{+}Testing^{+})^{+}$
$Integration^{+})^{+}$

ただし，上の表現では繰り返しは必ず *Analysis* まで遡って行われることになり，*Testing* から *Design* に戻るといった系列を含めるためには，より複雑な表現を用いることが必要となる．

（2） 文脈自由文法を用いた定義

このように多くの開発作業系列は正規表現のみで表すことができる．しかし，正規表現の持つ表現能力では，作業系列の持つ構造を直観的な形で表現できないものや，再帰構造のように，そもそも正規表現の枠組内では一般には表現できない系列が存在する[4]．

例えば，**図1**のような簡単な系列においても，正規表現では

$((a\ b)^{+}c(b(a\ b)^{*}c)^{*})^{+}$

のような記述となり，"$(a\ b)$" のように同じ部分の繰り返しが複数の箇所に展開されてしまう．しかし，これを文法形式で記述した場合には以下のようになり，比較的容易に構造を把握できる．

$S ::= A^{+}$
$A ::= aB$
$B ::= b(A|C)$
$C ::= c(B|\varepsilon)$

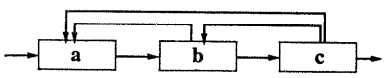


図 1 簡単な系列の例
Fig. 1 Simple example of the sequence.

表 1 文法記述に用いる超記号
Table 1 Meta-symbols used for grammar description.

超記号	意味
::=	定義
X,Y	XとYの連なり
X|Y	選択（X，Yのいずれか）
X*	Xの0回以上の繰り返し
X+	Xの1回以上の繰り返し
(..)	一連の系列をグループ化し，一つの記号のように扱う
"x"	終端記号x
;	生成規則の終り

上の例は非常に単純なものであるが，先の例で用いたウォーターフォールモデルの場合にも，最初の *Analysis* だけでなく，途中の任意の上流工程からやり直しができるように拡張しようとすると，記述がさらに複雑になり，その構造を直観的に知ることは困難となる．

また，再帰的に定義されるような系列は一般に正規表現では表すことができない．例えば，一つのプログラムを複数のモジュールに分解し，次に個々のモジュールについてさらに細かいモジュールに分解してゆくような場合，モジュールの細分化がどのレベルまで行われるか，あらかじめわかっておらず，また，サブモジュールの作成手順は再帰的になっている．このような手順は，文脈自由文法を用いて自然に表現することが可能である．このような理由から，ここでは記述手段として文脈自由文法を用いる．

文脈自由文法のわかりやすい表記法として，BNF (Backus-Naur Form) 記法やその拡張である EBNF (Extended Backus-Naur Form)[6),7)] などが広く用いられているが，本稿では開発過程特有の繰り返し構造の記述を考慮し，EBNF に一部正規表現の記法を取り入れるなどの変更を加えたものを用いる．この記法は**表1**のような超記号を持つ．

3.3 制約条件の付加

3.3.1 制約条件の役割

このようにして定義された開発作業系列は，文脈自由文法によって生成される系列の集合の要素である．しかし現実には，文脈に依存した制御や各時点における生成物の状態に依存した制御が行われるので，集合の要素の中には現実的に意味のない系列も含まれていることになる．例えば，文法 $S ::= ((Edit^{+}Compile)^{+}Link)^{+}$; で表される開発過程において，*Edit*−*Compile*−*Link* という系列は，*Compile* が成功した場合には正当な系列であるが，*Compile* が失敗した場

合には *Link* を行うべきではないので生成されてはならない．

このような制限を文脈自由文法のみで与えることは一般に不可能である．そこで，このような制御に用いられる情報を制約条件として定義する．

3.3.2 制約条件の意味

制約条件は，具体的にはその時点での生成物の満たすべき性質として定義する．ここでいう生成物とは実際に作成するソフトウェアだけではなく，仕様書や設計書のような中間生成物や，開発者間でやりとりされるメッセージや開発におけるデザイン上の意志決定の結果のように，開発作業中に作り出されるすべての情報を指す．このような生成物の集合を抽象的な状態Sとすると，制約条件とは，属性文法で用いられる意味定義を用いて以下のように説明される．

- すべての作業は，相続属性および合成属性としてそれぞれ上で述べた抽象的な状態変数を持つものとして定義する．つまり，すべての作業は与えられた相続属性に必要な変更を加え，合成属性として次の作業に渡す．
- 開発過程における作業の進行は生成規則を左／深さ優先で展開することに相当するので，属性の評価順序は先に述べたように左／深さ優先で行う．
- 例えば，生成規則 X::=Y; について，X，Yそれぞれの相続属性・合成属性を $\{X.S_{in}, X.S_{out}\}$，$\{Y.S_{in}, Y.S_{out}\}$ とする．このとき，この規則の制約条件とは，Xに規則を適用してYに展開する際にXの相続属性 $X.S_{in}$ が満たすべき条件であり，$X.S_{in}$ に対する述語 $Pred(X.S_{in})$ として定義される：

```
X::=Y
{   Y.Sin=X.Sin
    X.Sout=Y.Sout
} when Pred(X.Sin)where/*Pred の定義*/
```

ここでは相続合成，合成属性はともに一つの抽象的状態であり，意味規則は単なる属性の受渡し関係として用いられているが，受渡し関係はシンボルの展開順序に対応しているため自明であり，省略できる．そこで $S=Y.S_{in}=X.S_{in}$ とおいた上で Pred の記述のみを付加する．例えば次のような形になる．

[付加前]

```
P::=Q*;
Q::=Edit|Compile|Link;
```

[付加後]

```
P::=ε when Pred1(S)|
    Q P when not Pred1(S);
    where Pred1(S)==Exe(S) is newer than
    Src(S);
Q::=Edit when true|
    Compile when Pred2(S)|
    Link when not Pred2(S);
    where Pred2(S)==Src(S) is newer than
    Obj(S);
```

4. 開発支援への適用

4.1 基本方針

前節で記述した開発過程を基に，実際に開発支援系を構築する方法について述べる．

作業系列の文法を基に開発支援を行う方法として，ここでは開始記号から順に構文則に従って系列を生成してゆく方法を考える．すなわち，開発過程の各局面においてそれぞれ適用可能な構文則が一つならば自動的にそれを適用し，また，複数存在する場合にはそれを開発者に提示し，選択させてゆくことによって作業を誘導する．

ここでは，メニュー指向の開発支援系の構成について示す．この支援系は主として以下のような支援機能を持つ．

- **メニューによる開発作業の誘導**
  適用可能な構文則が複数ある場合にその中から一つをメニューによって開発者に選択させる．メニューを用いることで，開発者は作業を複数の候補の中から選択でき，あまり束縛感を感じずに作業誘導に従うことができる．また，メニューに挙げられた候補からしか選べないので，あらかじめ誤った作業の実行を防ぐ効果もある．さらに，操作の簡便化が図れる．
- **作業の進行の誘導**
  構文則の中でも，作業の系列が単純な連接として定義されている部分については，自動的に次の作業へと開発者を誘導する．
- **開発ツールの自動的起動**
  基本作業（終端記号に相当する作業）が，ツールの実行―例えばエディタやコンパイラの起動―である場合に自動的に起動を行う．したがって，起動すべきツールの具体的な名前や使用するオプションなどを開発者が記憶する必要が無く，コマンドとして入力する手間も軽減される．

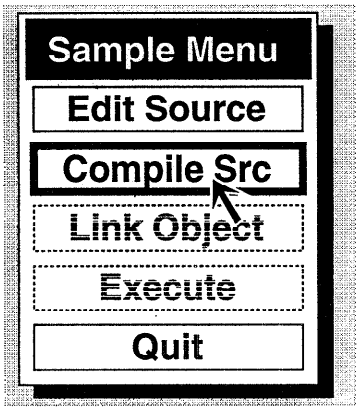


図 2 作業選択メニューの例
Fig. 2 An example of activity selection menu.

4.2 メニュー指向型支援系の構築方法

本稿でいうメニューとは，**図2**に示すようにウィンドウシステム上で複数の選択候補を提示し，人間がその中の一つをマウスによって選択するものをいう．

開発作業は，構文則をもとに生成されたメニューを逐次選択することによって進行する．このような開発支援系の作成は以下のような手順で行う．

1. 文法中の繰り返し部分および選択部分をもとにメニューを構成する
2. 終端記号に対応する各作業の具体的な内容（例えば，どのツールを用い，どのファイルを変更するか）を処理系に応じて定める

このうち作業2の内容は，実際にシステムを構築する環境やシステム記述言語に大きく依存する．したがって，ここでは具体的な例として UNIX 環境上で動作する PDL 処理系を用いた記述を次節で示すにとどめる．

作業を選択するためのメニューは，構文則の記述のうち同一の右辺を持つもの，つまり一つの非終端記号から生成される系列が複数存在するものをもとに作成される．

例えば，構文則 $X_0::=X_1|X_2|\cdots|X_n$；からは，選択項目として X_1 から X_n までを持つメニューが得られる．また，繰り返し（+, *）についても，これらを用いない等価な表現に変換することによって，メニューを得ることができる．実際に支援系を構築する際にはメニューの各選択項目の名前（ラベル）として適切な文字列を与える．メニューの具体的な構成例は5.3節に示す．

4.3 メニューにおける制約条件の効果

メニューの各選択項目にはそれぞれ，3.3節に述べた制約条件が付随する．開発者がメニューのある項目を選択するには，それに対応する制約条件が選択時に満たされていなければならない．逆に，制約条件の満たされていない項目は選択不可能な作業であることを使用者に提示する．図2はその実現例である．選択肢のうち制約条件を満たしていないものはその項目が他のものより薄く表示されており，選択できないことを示している．このとき，手続き的記述における条件分岐のようにただ一つのみの選択肢が制約条件を満たすわけではなく，それぞれの条件が排他的である必要もない．また，制約条件を特には持たない選択肢も許され，この場合には制約条件は常に満たされているとみなす．開発者は，制約を満たす項目の中から一つを選択する．

5. PDL 処理系を用いた構築

本節では，前節で示したメニュー指向型開発支援系を具体的に構築する例を示す．システムの記述言語には開発過程を形式的に記述するための言語として我々が開発した関数型言語 PDL (Process Description Language) を用いる[3]．我々は PDL の実行処理系を作成しており[13]，開発支援系のプロトタイピングに用いることができる．

5.1 PDL

PDL は代数的仕様記述言語の部分クラスであり，簡明な意味定義を持つことや様々な抽象度における記述能力を持つことなどを特徴とする．

PDL では，データ型の一つとしてシステム状態型というものを用意している．システム状態型とはファイルシステムやその他の計算機資源のすべて，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一つだけ存在する．

したがって，PDL では，一つの作業をある状態 S_1 を引数とし，遷移後の状態 S_2 を返り値とする状態遷移関数として表す．例えば，Edit, Compile, Link を行うプロセスを表す状態遷移関数 Main の単純な実現例は以下のようになる．

```
Main(S)==if error(Trans_1(S)) then
            Main(Trans_1(S))
            else Trans_2(Trans_1(S));
Trans_1(S)==Compile(Edit(S));
Trans_2(S)==if error(Link(S)) then
```

```
        Main(Link(S))
        else S;
```

PDL で記述されたプログラムは，試作した PDL インタプリタを用いて実行することができる．PDL インタプリタは，ウィンドウ環境への対応やデバグ機能，ツール起動関数を持つなど，開発支援系を構築するための様々な機能を備えている．PDL システムを用いることで，開発過程の記述を開発支援プログラムとして実行することができる．

5.2 PDL のメニュー関数

制約条件つきメニューによる選択実行の部分を PDL で簡潔に記述できるよう，我々は組み込み関数 *menubranch* を新たに用意した．*menubranch* は，3 字組［文字列，ブール式，状態式］のリストと状態型の引数を取り，システム状態を返す状態遷移関数で，例えば次のように記述して用いる．

```
P(S)==menubranch( {["s1", pre_s1(S), s1(S)],
                   ["s2", pre_s2(S), s2(S)],
                   ["s3", pre_s3(S), s3(S)]}, S);
```

ここで記号 {…} はリストを［…］は組をそれぞれ表している．この関数を評価すると，まず "s1", "s2", "s3" の三つの選択項目を持つメニューが表示される．このとき，制約条件 pre_s*i*(S) の値が偽の項目は表示が薄くなり，実際には選択不可能となる．ユーザが一つの項目をマウスによって選択すると，その項目 *i* に対応した状態式 s*i*(S) を評価して返す．

5.3 文脈自由文法から PDL への変換

文脈自由文法から PDL への変換は次のように行う．

1. $X_0::=X_1X_2\cdots X_n$ の形をした構文則は

   ```
   X_0(S)==X_n(…X_2(X_1(S))…);
   ```

   という状態遷移関数 X_0 の定義式に変換する．

2. $X_0::=X_1|X_2|\cdots|X_n$ の形をした構文則は

   ```
   X_0(S)==menubranch(
     {["X_1", R_1(S), X_1(S)],
      ["X_2", R_2(S), X_2(S)],
      …
      ["X_n", R_n(S), X_n(S)]
     }, S);
   ```

   という状態遷移関数 X_0 の定義式に変換する．ただし，ここで R_1…R_n はそれぞれ制約条件を表す関数の名前である．

3. すべての終端記号 T_i について，対応する状態遷移関数 T_*i*(S) を PDL で記述する．例えば，プログラムのコーディングであれば具体的にはテキストエディタを用いたソースファイルの編集であるから

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

   のような記述となる．

4. 各メニューの構成要素の制約条件 R_i に対応する論理関数 R_*i*(S) についても同様に PDL で記述する．例えば，オブジェクトファイルよりも新しいソースファイルが存在するか，という条件は，PDL のシステム関数 time_stamp を用いて

   ```
   Res(S)==time_stamp(Sourcefile(S))>=
           time_stamp(Objectfile(S));
   ```

   のように定義する．

5.4 実 行 例

以上述べたようにして得られた PDL プログラムは支援系の基本制御機構として動作する．PDL インタプリタは X ウィンドウシステム上で動作するメニューその他のインタフェースを持ち，マルチウィンドウ上での開発支援システムを提供する．

例として，Kellner らによって提案されているソフトウェアプロセスの例題を取り上げ，その実行例を示す．

5.4.1 ソフトウェアプロセスモデル化のための例題

開発過程の形式化，モデル化には，これまでに様々な方法が提案されている．この問題は，これらの理解や比較を助けるものとして国際ソフトウェアプロセスワークショップにおいて提案されたもので，すでにいくつかのモデルによる記述がなされている[10]．問題は自然語（英文）で十数ページにわたって，あるモジュールに関する設計変更，プログラムの変更，および，それらのレビュー，単体テストなどの作業が規定されている．**図 3** に，この問題が規定する八つの作業と，その関連を示す．

5.4.2 例題の記述と実行

ここでは，この問題の中心となる核問題，および，一部の拡張問題を対象とした．これは，作業計画の立案後，設計変更（ModifyDesign），設計のレビュー（ReviewDesign），コード変更（ModifyCode），単体試験計画の変更（ModifyTestPlan），単体試験パッケージの変更（ModifyUnitTestPackage），および，その実行（TestUnit）などを行う作業である．これらを文法表現を用いて**図 4** のように記述した．この記述では終端記号は小文字で始まり，非終端記号は

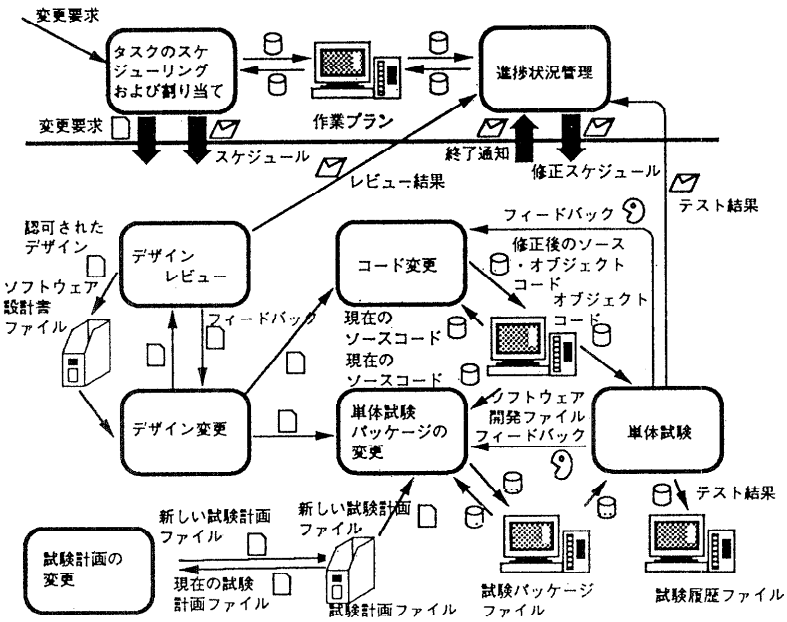


図 3 ソフトウェアプロセスモデル化の例題の概要
Fig. 3 Overview of SPM (Software Process Modeling) example problem.

```
Process ::= Schedule Assign Tasks, Main;

Main ::=
 (
    (ModifyDesign, ReviewDesign)+,
    (ModifyCode, (ModifyTestPlan,ModifyUnitTestPackage,
    TestUnit)+)+
 )+;

ModifyDesign ::=
    CheckOutDesign, (EditDesign,CheckInDesign)+, Release;

CheckOutDesign ::= execute_sccs_get;

CheckInDesign ::= execute_sccs_put

ReleaseDesign ::=
    execute_mail, execute_lpr, send_hardcopy_by_hand;

ModifyCode ::=
    CheckOutText,
    ((EditText,CheckInText,Compile)+,(CodeCheck|),)+,Release;

CheckOutText ::= execute_sccs_get;

CheckInText ::= execute_sccs_put;

Edit ::= execute_emacs;

CodeCheck ::= execute_browser;

Release ::= execute_mail, execute_lpr, send_hardcopy_by_hand;
```

図 4 例題に対する解（作業系列の文法）の記述例
Fig. 4 Activity grammar for SPM example problem.

大文字で始まる．また，開始記号は Process である．この記述では設計やコードの各変更ごとにその版が保存されるように，版管理ツールを起動して，チェックイン，チェックアウトの作業を行うようにしている．また，設計変更（ModifyDesign）やコード変更（ModifyCode）の終りには，電子メールで進捗状況管理部に連絡し，さらに，それぞれの変更の結果を印刷して次の作業に手渡しすることを指定している．

この記述に制約条件を付加した後，PDL に変換し，実際のツール割り当てなどの記述を加えて実行させた．Kellner の問題自身は，複数の人間が分担してこれらの作業を行うことになっているが，得られたプログラム上では，1人の人間が1台のワークステーションのウィンドウシステム上で，これらの作業を順次行う．これは，PDL が1人の人間の1台のワークステーション上での作業の制御しか前提としていないためである．したがって，この方法では，元の問題が要求する作業を正確に支援するシステムは直接生成できないが，1人の人間がそれに対応した作業を行う場合のシステムが得られる．また，図4の記述が元の問題と対応しているか否かを知るための一種のシミュレータとみなし，その動作の確認をすることができる．

6. おわりに

ソフトウェア開発プロセスの性質を形式的に表す一手段として，作業系列を文法によって定義する方法を提案した．正規表現や文脈自由文法を用いることにより，開発過程の記述に対する形式的な評価が期待できる．どのような評価・検証が可能であるか，今後検討を進めていく予定である．

本稿では，さらに，作業の選択幅の広い開発支援環境をこの文法記述をもとに構築する方法を示した．例で示した支援系は，制約条件つきのメニューによる選択／実行を基本とした動作により，作業の自由度，正しい作業系列の生成，優れた操作性を実現する．

一方，この手法は作業系列というソフトウェア開発過程の持つ一側面のみをモデル化するものであり，生成物の管理機構を表すためのモデル（例えばオブジェクト指向モデル）や生成物の内容の持つ意味モデル（例えば代数的仕様）と協調することによって，開発過程全体のモデルを形成する．したがって，生成されるシステムもまた，作業の具体的な内容などや操作の意味を規定するようなプロダクト管理のモデル[11),12)]とは独立しており，さまざまなプログラム管理システムと協調して，一種の誘導システムとして開発支援に役立てることができる．

将来への課題としては，この文脈自由文法を用いたモデルは，並行した作業を持つ系列には対応していないという点が挙げられる．また，複数の開発者による作業への対応も検討中であるが，この場合には全体を一つの作業系列と見るのではなく，個々の開発者の作業系列間の通信の問題として検討中である．さらに，現在は文法情報から PDL による支援系を得るための変換は手作業で行っているが，これを支援するための処理系を作成することを考えている．

謝辞 本研究に関して，有益なご助言をいただきました(株) SRA の新田稔氏に感謝します．

参考文献

1) Avrunin, G. S., Dillom, L. K., Wileden, J. C. and Riddle, W. E.: Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems, *IEEE Trans. Softw. Eng.*, Vol. SE-12, No. 2, pp. 278-292 (1986).

2) 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—，信学論 (D-I)，Vol. J 72-D-I, No. 12, pp. 874-882 (1989).

3) Inoue, K., Ogihara, T., Kikuno, T. and Torii, K.: A Formal Adaptation Method for Process Descriptions, *Proc. 11 th ICSE*, pp. 145-153 (1989).

4) 福村晃夫，稲垣康善：オートマトン・形式言語理論と計算論，岩波講座情報科学-6，岩波書店 (1982).

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

6) Jensen, K. and Wirth, N.: *PASCAL User Manual and Report*, 3rd edition, Springer-Verlag (1985) (邦訳：原田賢一訳：PASCAL 原書第三版，情報処理シリーズ 2，培風館 (1988)).

7) Johnson, S.: YACC: Yet Another Compiler Compiler, Technical Report 32, Computing Science (1975).

8) Kaiser, G. E. and Feiler, P. H.: An Architecture for Intelligent Assistance in Software Development, *Proc. 9 th ICSE*, pp. 180-188 (1987).

9) Katayama, T.: A Hierarchical and Functional Software Process Description and Its Enaction, *Proc. 11 th ICSE*, pp. 343-352 (1989).

10) *Proc. 6 th International Software Process Workshop* (1990).

11) Kishida, K., et al.: SDA: A Novel Approach to Software Environment Design and Construction, *Proc. 10 th ICSE*, pp. 69-79 (1988).

12) Matumoto, Y. and Ajisaka, T.: A Data Modeling in the Software Project Database Kyoto-DB, *Advances in the Software Science and Technology*, 日本ソフトウェア科学会編, Vol. 2, pp. 103-121 (1990).

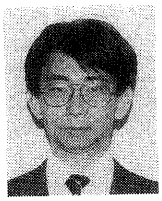
13) 荻原剛志，井上克郎，鳥居宏次：ソフトウェア開発を支援するツール起動自動制御システム，信学論 (D-1)，Vol. J 72-D-I, No. 10, pp. 742-749 (1989).

14) Osterweil, L.: Software Processes are Software Too, *Proc. 9 th ICSE*, pp. 2-13 (1987).

15) Taylor, R. N., et al.: Foundations for the Arcadia Environment Architecture, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (SIGSOFT Software Engineering Notes, 13-5)*, pp. 1-13 (1988).

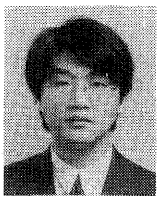
16) Williams, L. G.: Software Process Modeling: A Behavioral Approach, *Proc. 10th ICSE*, pp. 174-186 (1988).

(平成4年 4 月17日受付)
(平成4年12月10日採録)



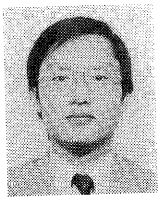
飯田　元（正会員）

昭和39年生．昭和63年大阪大学基礎工学部情報工学科卒業．平成2年同大学大学院博士前期課程修了．同年同後期課程入学．平成3年大阪大学基礎工学部情報工学科助手，現在に至る．工学修士．ソフトウェア開発プロセスおよび開発支援環境の研究に従事．日本ソフトウェア科学会会員．



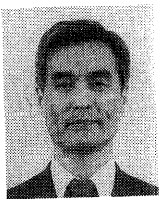
荻原　剛志（正会員）

昭和36年生．昭和60年山梨大学工学部計算機科学科卒業．昭和62年同大大学院修士課程修了．平成2年大阪大学博士課程修了．同年同大情報処理教育センター助手，現在に至る．工学博士．ソフトウェア工学，データ圧縮などの研究に従事．著書（共著）「NeXT ユーザーガイドブック」．電子情報通信学会，日本ソフトウェア科学会各会員．



井上　克郎（正会員）

昭和31年生．昭和54年大阪大学基礎工学部情報工学科卒業．同年同大学基礎工学部情報工学科助手．昭和59〜61年ハワイ大学助教授．平成3年大阪大学基礎工学部情報工学科助教授，現在に至る．ソフトウェアプロセス，関数型言語の処理系等の研究に従事．工学博士．電子情報通信学会，ACM，IEEE 各会員．



鳥居　宏次（正会員）

昭和13年生．昭和37年大阪大学工学部通信工学科卒業．昭和42年同大大学院博士課程電子工学専攻修了．同年電気試験所（現電子技術総合研究所）入所．昭和50年ソフトウェア部言語処理研究室室長．昭和59年大阪大学基礎工学部情報工学科教授．平成3年奈良先端科学技術大学院大学情報科学研究科教授（大阪大学併任），図書館長，現在に至る．工学博士．ソフトウェアメトリクスやフォーマルな開発プロセスなど，定量的取扱いを中心とするソフトウェア工学に興味を持つ．IEEE Transaction on Software Engineering および IEEE Software 誌などの編集委員．IEEE，ACM および電子情報通信学会各会員．