The University of Osaka
Institutional Knowledge Archive

Title	あるプロダクト間関係の記述とプロセス記述言語PDLへの変換
Author(s)	西村, 好洋; 飯田, 元; 荻原, 剛志 他
Citation	情報処理学会研究報告. ソフトウェア工学研究会報告. 1991, 1990-SE-077(13), p. 101-106
Version Type	VoR
URL	https://hdl.handle.net/11094/50158
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

ソフトウェア工学 77−17
（1991. 2. 8）

あるプロダクト間関係の記述とプロセス記述言語ＰＤＬへの変換

西村好洋　飯田元　荻原剛志　井上克郎　鳥居宏次

大阪大学　基礎工学部　情報工学科

ソフトウェア開発過程において生成されるプロダクトについて，プロダクト間の関係を表す表記法を設計した．プロダクト間の関係は木構造で表し，この木構造の葉にプロダクトを，節にプロダクトに対する作業を対応させる．その結果，木構造の定義を変更することにより様々なソフトウェア開発に応用することができる．このプロダクト間の関係記述はトランスレータによりプロセス記述言語ＰＤＬ（Process Description Language）のスクリプトに変換され，実行可能な開発支援環境を構築することができる．

A DESCRIPTION OF PRODUCT RELATIONS AND ITS TRANSLATION INTO PROCESS DESCRIPTION LANGUAGE PDL

Yoshihiro Nishimura, Hajimu Iida, Takeshi Ogihara, Katsuro Inoue and Koji Torii

Faculty of Engineering Sience, Osaka University, Toyonaka-shi, 560 Japan

We have designed a notation to express product relations with tree structures. Each leaf in a tree corresponds to a product and, each internal node corresponds to a set of processe sassociated with products. Various kinds of software development are easily expressed with the tree structures only changing the definition of the leaves and internal nodes. These descriptions are translated into scripts of Process Description Language PDL by a translater, and the translated scripts are executed by the PDL interpreter, which establish development support systems.

1. はじめに

近年，ソフトウェア開発を形式的に記述する研究が行なわれている．これは，ソフトウェア開発のための指針やソフトウェア開発の過程が，一般にあいまいなものが多く，それらをどの様に解釈，実行するのかは人によって異なる場合が多いためである．形式的な開発過程の記述法として，Osterweil による Process Programming[1] や，Williams の Behavioral Approach[2] などがあげられる．開発過程を形式的に記述することができれば，多くの人があいまいさなく理解する事ができる．また，その記述（スクリプトと呼ぶ）を直接実行するようなシステムを構築すれば，そのスクリプトによって定義された開発支援環境を実際に生成することが可能となる．

我々は，開発過程を記述するために関数型言語ＰＤＬ（Process Description Language）とそのインタープリタを開発した[3]．ＰＤＬは基本関数として，ツール起動やウィンドウ操作のための関数等を備えており，これらの関数を組み合わせることで開発過程を記述する．記述されたスクリプトはインタープリタにより解釈，実行することができ，これにより目的の支援環境を構築することができる．ＰＤＬインタープリタは現在，ワークステーション上で稼動中で，今までにＪＳＤによる開発支援システム[4]，ＴｅＸによる文章印刷システム，ＳＣＣＳによるバージョン管理システム等を構築してきた．

開発過程を記述するとき，開発の過程（プロセス）を中心に考えることもできるし，開発に伴うファイル（プロダクト）中心に考えることもできる．この２つは互いに密接な関係があり，普通，プロセスについて着目すると，プロダクトはその入力，出力になる．また，プロダクトに着目するとそのプロダクトを生成するプロセス，あるいはそのプロダクトが用いられるプロセスが存在することになる．

現在まで，我々の進めてきたＰＤＬによる開発過程の記述は，上記の２つの方法のなかのプロセス中心の方法であった．また，［５］では開発過程の中で生じるプロダクト間の関係に着目し，プロセスをプロダクトに対する操作としてとらえた開発過程の記述法を提案した．本論文では［５］の方法をさらに関数型言語ＰＤＬのスクリプトに容易に変換できるよう拡張した記述法について述べる．この記述はトランスレータによってＰＤＬスクリプトに変換し，実際の支援環境を構築することができる．

以下では，ＰＤＬとそのインタープリタの特徴について述べ，次いで開発に伴うプロダクト間の関係の記述方法を説明する．更に，この記述方法による実際の開発過程の記述例を示し，その有効性について論じる．最後に，この記述をＰＤＬスクリプトに変換するためのトランスレータについて簡単に述べる．

2. 関数型言語ＰＤＬとＰＤＬインタープリタ

開発過程を記述するための関数型言語ＰＤＬは，基本データ型として，ソフトウェア及びハードウェアの状態を抽象的に表す，”システム状態：Ｓ”を持つ．このシステム状態のある値に，ツールの起動ウィンドウの操作等を行う遷移関数を施して，目的とするシステム状態にすることにより，支援環境を構築する．また，複数の状態遷移関数のうち，ユーザーが実際の開発過程においてそのうちの１つを選択，実行させることができるメニュー関数も用意されている．

関数型言語ＰＤＬによって記述されたスクリプトは，ＰＤＬインタープリタにより解釈，実行することにより目的とする支援環境を実現することができる．ただし，ＰＤＬインタープリタによる支援環境は一人の人間が１台のワークステーション上で開発作業を行うことを前提としている．

これら，関数型言語ＰＤＬ，並びにそのインタープリタについての詳しいことは，参考文献［３］を参照されたい．

3. プロダクトの関係記述

3.1 ソフトウェア開発に伴うプロダクト

ソフトウェアを開発する場合，一般には複数のプロダクト（ヘッダファイル，ソースファイル）から作成される．さらに，開発途中において中間ファイル（プロダクト）がそれらのソースファイルなどのプロダクトから生成される．また，必要に応じて，仕様書やクロスリファレンスファイル等が作成される場合もある．

開発過程を記述する意味で，これらのプロダクトの関係をなんらかの形で書き表すことを考える．そこで，プロダクトの相互関係の記述を説明するために，次のようなＣプログラム作成の例を考える．

- プログラムはソースファイル test1.c と test2.c から作成される．
- test1.c は def.h をインクルードする．
- test2.c は def.h と function.h をインクルードする．
- ２つのソースファイルはコンパイルされると，test1.c は test2.o に，test2.c は test2.o になる．
- test1.o と test2.o は，リンクされて実行可能なＣプログラムファイル test になる．

ソフトウェア開発におけるプロダクトは，その性質ならびに開発過程における役割によって，いくつかのグループに分類することができる．

例えば，先に示したＣプログラム開発の例では，次のようにプロダクトを分類することができる．

① def.h と function.h
② test1.c と test2.c
③ test1.o と test2.o
④ test

①は，ソースファイルにインクルードされるプロダクトのグループである．同様に，②はソースファイル，③はオブジェクトファル，④は実行ファイルのグループである．これら，同じグループに分類されたプロダクトはこの後述べる構造体で，おなじ種類の構造体によって置き換えられて表される．このことは，構造体による開発過程の記述を考える際にその汎用性を大きくする事につながる．

3.2 プロダクトの関係を表す木構造

さきに述べたプロダクトについてどのような作業がなされているか，プロセスの観点から考えると，次のように分けて考えることができる．（括弧内はプロセスの名前）

① test1.c と def.h から test1.o が作られる．（コンパイル）
② test2.c と def.h と function.h から test2.o が作られる．（コンパイル）
③ test1.o と test2.o から test が作られる．（リンク）

これらの開発過程における関係は，ＰＤＬではプロセスに着目することにより記述することができる．しかし，これらのプロダクトの関係を直接記述することはできない．そこで，図１で示すように木構造により表すことを考える．この木構造は次のような条件を満たすように構成されている．

・葉にはプロダクトが１対１に対応する．
・節には，プロダクトの集合とそのプロダクトに対する作業（プロセス）が対応する．
・開発過程全体を考えた時，プロセスＡよりプロセスＢをさきに行われなけらばならいとき，プロセスＡに対応する節を根とする部分木の中にプロセスＢに対応する節が存在する．

そこで，以下に木構造を定義する記述の方法を述べ，これにより，開発支援環境を構築できることを示す．

3.3 構造体の導入

プロダクト間の関係，プロダクトに対する作業を構造体によって表すことを考える．この構造体が前出の木構造における葉，節に対応する（図１参照）．構造体はその名前の前に "＃" を付け，次のような形で宣言する．

```
＃構造体名（木構造で子にあたる構造体名）
{
    関数の定義
}
```

『木構造で子にあたる構造体名』の部分には，木構造を考えたとき，宣言している構造体の直接子になる構造体の名前を記述する．ただし，構造体の名前が繰り返される場合にはこれを正規表現で表す．『関数の定義』の部分には，プロダクトに対する作業を表す関数 procedure() や，作業を行う際の条判断を行なうための関数等を記述する．この時，関数の定義には『木構造で子にあたる構造体』の関数を参照することができる．これは，『木構造で子にあたる構造体名』の部分でタグにより構造体名を置き換え，タグ名と必要とする関数名をピリオドでつなぐことにより表される．

以下に，この構造体宣言の具体的な説明を記す．

3.4 構造体による木構造の構成

構造体によって木構造を構成するためには，それぞれの構造体がどの構造体を子に持つかを宣言する必要がある．構造体＃Ａが表す節が，構造体＃Ｂ，＃Ｃが表す節を子に持つとき，これを宣言するには

```
＃Ａ（＃Ｂ，＃Ｃ）
 {
    関数の定義
}
```

の形で宣言する．

たとえば，構造体 #header がインクルードファルを表す構造体 #htext（０個以上あるとする）を子として持つとき，正規表現を用いて，

```
#header(#htext*)
 {
    関数の定義
}
```

と，宣言される．さらに，この構造体 #header とＣのソースファイルを表す構造体 #ctext を子として持つ構造体 #source を宣言するには，

```
#source(#header,#ctext)
 {
    関数の定義
}
```

とする．

この構造体の宣言繰り返し，木を構成する全ての節に行うことにより木構造を構造体により定義することができる．

3.5 構造体によるプロダクト対する作業の宣言

プロダクトに対する作業の定義は構造体宣言の中の『関数の定義』の部分に記述する．関数の定義は，関数型言語ＰＤＬと同じものを用い，さらに構造体宣言で定義した自分の子にあたる構造体を参照することができる．ただし，構造体においてのプロダクトの作業はすべて関数 procedure() を定義することによって行なう．

例えば，次のように構造体＃Ａを定義することを考える．

- 構造体＃Ａは構造体＃Ｂを子に持つ
- 構造体＃Ｂのタグ名は『b』とする
- b が持つプロダクトに対して，エディットを行なう．これは関数 edit() によって実行される
- 構造体＃Ｂの持つ関数 get_element() を実行すると構造体＃Ｂのもつプロダクト名が返される．

これは，次のように記述される．

```
#A (b : #B)
{
 procedure() = edit( b.get_element());
}
```

たとえば，構造体 #source のもつ作業は，次のように記述できる．（一部）

```
#source(head:#header,ct:#ctext)
{ procedure() = edit(ct.get_element());}
```

このほかにも，『関数の定義』の部分にはさまざまな関数を定義することができる．例えば，構造体＃Ａのもつプロダクトと構造体＃Ｂのもつプロダクトの両方が存在しているかを調べる関数を構造体＃Ｃのなかで関数 all_exist() として定義するならば

```
#c ( a : #A, b : #b)
{
 all_exist() =
        exist(a.get_element())
        & exist(b.get_element());
}
```

と，定義できる．この関数は構造体＃Ｃを子として持つ構造体の中で用いることができる．

図２に，先に示した開発過程の構造体による記述について構造体の完全な宣言全体の一例を示す．

4．プロダクトと構造体の関係

4.1 構造体によるプロダクトの関係の宣言

構造体の宣言は，単に木構造を考えたとき，節に対応する１つ１つの構造体が，子としてどの構造体を持つかということと，その節におけるプロダクトに対する作業を宣言したものである．実際のソフトウェア開発を考えた場合には，この構造体の宣言の他に，実際に木構造の構成と各節に対応する構造体の名前を宣言しなければならない．すなわち，プロダクト名と構造体名の宣言を行なう必要がある．実際の開発過程の記述では，構造体の宣言と，プロダクト名及び構造体名の宣言の２つから構成される．

先に示した例について，プロダクト名，構造体名の宣言の一例を図３に示す．

この宣言は，中間節の構造体名の宣言を省略し，次のようにプロダクト名だけで，表すことも可能である．

```
#program = ({(({"def.h"},"test1.c"),
                        "test1.o"),
        (({"def.h","function.h"},"test2.c"),
                        "test2.o")},
                                "test");
```

ここで，() は構造体，{} はリストを表している．

4.2 構造体宣言の汎用性

構造体の宣言は，単に木構造を考えたときの節同士の対応関係を示し，さらにプロダクトに対する作業を構造体に付随させて宣言したものである．したがって，他のソフトウェア開発過程を記述するにはプロダクト名，構造体名の宣言を変えるだけで，構造体の宣言を変更することなく用いることができるすなわち，この構造体の宣言は開発過程におけるプロダクトの関係のみを記述するものであるので，特定のソフトウェア開発に限るものではなく，汎用性に富んでいる．

5．関係記述からの開発支援環境の構築

5.1 ＰＤＬトランスレータについて

ＰＤＬインタープリタは，構造体によって記述されたプロダクトの関係記述を直接，解釈実行することができない．そこで，この構造体の記述をＰＤＬスクリプトに変換したのちにインタープリタによって解釈実行させる．このＰＤＬスクリプトへの変換は，トランスレータによって行う．トランスレータは，構造体の宣言と構造体名，プロダクト名の宣言の２つから，ＰＤＬスクリプトを生成する．生成されたＰＤＬスクリプトはインタープリタによって，実際に開発支援環境を構築する事が可能である．

この構造体宣言を変換したＰＤＬスクリプトには実行順序についての記述が存在していない．これは構造体の宣言がプロダクトに対する作業を記述し，開発過程全体についての実行順序は記述していないためである．そこで実際の実行可能なスクリプトにするためには，それぞれのプロダクトに対する作業について，どの作業から始めるかを指定するための

スクリプトを付随させなければならない．そのため，トランスレータは，メニューにより開発者が作業順序を自由に選択できるようなスクリプトを，構造体を変換したスクリプトと共に出力する．

5.2 構造体宣言からＰＤＬスクリプトへの変換

構造体の宣言中にある関数の左辺は

（構造体名）＿（関数名）(S)--

の形で変換し，右辺に現れるマクロを用いた関数は

（実際の子にあたる構造体名）＿（関数名）(S)

の形でＰＤＬスクリプトに変換することができる．（ただし，式中に現れる S は，さきに述べた関数型言語ＰＤＬの ” システム状態：S ” である．）
例えば，構造体の宣言が

```
#source(he:#header,ct:#ctext){
 condition() = allexist(ct.get_elememnt());
 procedure() = if he.condition() then
                    edit(ct.get_element());
}
```

となっており，構造体名の宣言が

```
#source = sol;
sol.child = hed1,ctext1;
```

である場合，構造体宣言中の関数 condition() と procedure() をＰＤＬスクリプトに変換すると

```
sol_condirion(S) --
        allexist(ctext1_get_element(S));

sol_procedure(S) --
 if hed1_condition(S) then
      edit(ctext1_get_element(S)) else S;
```

となる．このように，構造体による開発過程の記述は容易にＰＤＬスクリプトに変換することができる．

6．X-window を用いた関係記述の作成支援環境

プロダクトの関係記述は，構造体宣言の部分，プロダクト名および子レベルの構造体名宣言の部分の２つに分けられる．このうち，構造体宣言については，ソフトウェア開発過程を記述する際には，他の開発過程の構造体宣言を再利用することが可能であり，プロダクト名，構造体名の宣言だけを新たに記述すればよい．

X-window による関係記述の作成支援環境は，構造体宣言に基づいて実際にプロダクト間の関係を表す木構造を画面上で作成し，構造体名，プロダクト名の宣言を出力する．さらに，この支援環境にトランスレータの機能を付け加えることにより，画面上でのプロダクト間の関係の入力から直接実行可能なＰＤＬスクリプトを得ることができる．

現在，この支援環境をトランスレータと並行して開発中である．

7．あとがき

あるソフトウェア開発におけるプロダクト間の関係を構造体により記述し，その記述によって開発支援環境を構築する方法を述べた．構造体の宣言を用いることにより，さまざまなソフトウェアの開発過程が，構造体名，プロダクト名の宣言の部分を入れ換えることで記述することができる．これにより，開発過程の記述に関する作業効率が向上し，実際の支援環境が容易に構築することができる．

関係記述のＰＤＬへの変換は，トランスレータにより行ない，直接ＰＤＬスクリプトを記述するより短時間で目的の作業環境を作成することができると考えられる．

今後の課題としては，開発作業中にプロダクト関係が変化する（例えば，プロダクトの数が開発途中で変化するなど）場合にも対応できるような構造体の宣言等を考えていきたい．

［参考文献］

[1] Osterweil,L. : "Software Processes are Software too" , Proc. of 9th ICSE, pp.2-13 (1987).

[2] Willams,L.G.:"Software process modeling: A behavioral approach", Proc. of 10th ICSE,pp.174-186(1988).

[3] 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” ，信学技報，COMP88-73(1988).

[4] 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” ，信学技報，COMP88-74(1988).

[5] 西村，飯田，荻原，新田，井上，鳥居：” プロダクトの関係記述による開発支援環境の構築” 情報処理研報，SE-75-3(1990).

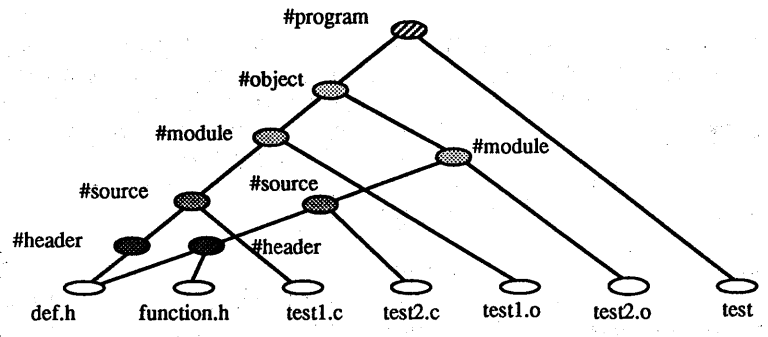

図1　木構造によるプロダクトの関係

```
#htext {
    procedure() = nothing();
    get_element() = element;
}
#header (ht : #htext*) {
    procedure() = menubranch(edit(elementselect(ht.get_element())),
                             view(elementselect(ht.get_element()));
    condition() = allexist(ht.get_element());
}
#ctext {
    procedure() = nothing();
    get_element() = element;
}
#source (he : #header,ct : #ctext){
    procedure() = if he.condirion() then
            menubranch(edit(ct.get_element()),
                       view(ct.get_element()));
    condition() = allexist(ct.get_element());
    get_element() = ct.get_element();
}
#obj {
    procedure() = nothing();
    get_element() = element;
}
#module (so : #source,ob : #obj){
    procedure() = if so.condition() then
                  compile(so.get_element());
    condition() = allexist(ob.get_element());
    get_element() = ob.get_element();
}
#object (mo : #module+){
    procedure() = nothing();
    get_element() = mo.get_element();
    condition() = alltrue(mo.condition());
}
#exe {
    procedure() = nothing();
}
#program (obje : #object,ex : #exe){
    procedure() = if obje.condition() then
                  link(obje.get_element(),ex.get_element());
}
```

図2　構造体の宣言の一例

```
#htext = htext1,htext2;
htext1.element = "def.h";
htext2.element = "function.h";

#header = hed1,hed2;
hed1.child = htext1;
hed2.child = htext1,htext2;

#ctext = ctext1,ctext2;
ctext1.element = "test1.c";
ctext2.element = "test2.c";

#source = so1,so2;
so1.child = hed1,ctext1;
so2.chiild = hed2,ctext2;

#obj = obj1,obj2;
obj1.element = "test1.o";
obj2.element = "test2.o";

#module = mdl1,mdl2;
mdl1.child = so1,obj1;
mdl2.child = so2,obj2;

#object = objs;
objs.child = mdl1,mdl2;

#exe = execpro;
execpro.element = "test";

#program = pro;
pro.child = objs,execpro;
```

図3　プロダクト名，構造体名の宣言の一例