OUKA
The University of Osaka
Institutional Knowledge Archive

Title	プロダクトの関係記述による開発支援環境の構築
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. 1990, 1990-SE-075(89), p. 1-8
Version Type	VoR
URL	https://hdl.handle.net/11094/50130
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 75－3
（1990．11．13）

プロダクトの関係記述による開発支援環境の構築

西村好洋* 飯田元* 荻原剛志* 新田稔+ 井上克郎* 鳥居宏次*

* 大阪大学 基礎工学部 情報工学科
+ ㈱ＳＲＡ

開発過程において生成されるプロダクト（ファイル）について，その依存関係の記述から開発支援環境を構築する方法を述べる．プロダクトの依存関係は木構造によって表すことにしこの木構造を定義する記法を設計した．実際のプログラム開発に用いるプロダクトは，木の葉に対応し，この葉のプロダクトを差し替えることにより様々なプログラム開発に応用することができる．記述された依存関係は関数型言語ＰＤＬ（Process Description Language）に変換され，インタープリタにより実行することにより開発支援環境を構築することができる．

CONSTRUCTION OF SOFTWARE DEVELOPMENT SUPPORT SYSTEM BASED ON DESCRIPTION OF PRODUCT RELATIONS

Yoshihiro Nishimura* Hajimu Iida* Takeshi Ogihara* Minoru Nitta+
Katsuro Inoue* Koji Torii*

* Faculty of Engineering Science, Osaka University
+ Software Research Associates, Inc.

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

We intoroduce a method to construct software development support system from the description of (software) product relations. The dependencies among products have been expressed by tree structures, and a notation for these tree structures has been designed. Each product appearing through software development corresponds to a leaf in a tree. Various kinds of software development are easily expressed only by changing the products at the leaves. The descripion of product relations is translated into a script of Process Description Language PDL. We can obtain a support system by executing the translated PDL script with the PDL interpreter we have developed.

1. はじめに

ソフトウェア開発のための指針やその開発方法，またその開発方法に対する考え方には様々なものがある．これらのソフトウェア開発の過程は，一般にあいまいなものが多く，それらをどの様に解釈し，また実行するのかは人によって異なる場合が多い．そのため，ソフトウェア開発のための指針が必要とされている．最近，この要求に対して，ソフトウェアの開発過程を形式的に記述しようと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 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，あいまいさなくこれらの開発過程を多くの人が理解する事ができ，さらに，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ることができれば，そのスクリプトによって定義された開発過程の支援環境を生成することが可能となる．このような支援環境は，作業内容に合ったツールを順次自動的に起動する事による作業効率の向上や，作成されるソフトウェアの品質の向上を期待することができる．

我々は，これらの目的を達成するために，開発過程を記述するための関数型言語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により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可能となる．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あり，実際に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いる．

ソフトウェア開発過程を理解し，記述する方法は，大きく次の２つの方法があると考えることができる．１つは開発の過程（プロセス）中心の方法であり，もう１つは開発に伴うファイル（プロダクト）中心の方法である．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，入力と出力のプロダクトが存在する．逆に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．

現在まで，我々の進めてきたＰＤＬによる開発過程の記述とその支援環境の構築は，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．そこで今回，もう一つの方法であるプロダクトに主眼をおいた形での開発過程の記述を考えた．

本論文では，開発過程の中で生じるプロダクトとその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について述べる．この記述はＰＤＬに変換することにより，実際の支援環境を構築することが可能なものである．

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．

2. 関数型言語ＰＤＬとＰＤＬインタープリタ

2.1 ＰＤＬの特徴

代数型言語ＡＳＬと同様の意味定義，構文を持つ関数型言語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”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．このようなシステム状態のほかにも整数，論理，文字列，タプル，リスト構造といったデータ型も取り扱うことができる．また，複数のツールを並列に実行することができるよう，基本関数として２つの状態遷移関数を同時に評価する関数@（並列演算子）が用意されている．さらに，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．この関数を使用することにより，開発過程の記述に柔軟性を持たせることができ，必要以上の作業順序の拘束からユーザーを開放することができる．

主な組み込み関数を表１に示す．

2.2 ＰＤＬインタープリタの特徴

ＰＤＬインタープリタは，端末，あるいはファイルからＰＤＬスクリプトを入力し，これを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．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．

このインタープリタは単にスクリプトを解釈実行するだけでなく，スクリプトの記述や詳細化を

容易に進めることができるよう次のような様々な機能を備えている．
(1) 抽象的な記述から得られた未定義関数を含むようなＰＤＬスクリプトでも一部実行ができるよう，実行中にその未定義関数に対する値や関数定義を与えることができる．
(2) わかりやすく変更が容易なスクリプトが記述できるよう様々なマクロ機能を持つ．
(3) さまざまなインタープリタに対するコマンドが用意されている．例えば，定義ファイルを取り込む機能 #include は，個人の好みのツールや設定を登録し，ライブラリー化することを可能にする．
(4) Ｘウィンドウを利用したデバッグ用インターフェースが用意されており，これにより，記述されたスクリプトのデバッグが支援されている．

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

１つのＣプログラムを開発する時には，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）が作成される．また，中間ファイル（プロダクト）が同様に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．更に，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．

ここで，これらのプロダクトの関係がなんらかの形で書き表すことを考える．更に，その記述から開発支援を行うことを考える．そこで，プロダクトの相互関係の記述を説明するために，以下のＣプログラム作成の例を用いる．

『Ｃプログラムを作成する．この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．これらのソースファイルは，それぞれインクルードファイルを持ち，test1.c は def.h を，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．さらに，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．』

これらの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．この木構造において，葉にはプロダクトが対応し，接点には，プロダクトの集合，あるいはプロセスが対応することになる．そこで，以下に木構造に対する記述の方法を考え，これにより，プロダクトの関係を記述することを考える．

3.2 プロダクトの型の導入

プログラムの作成において生成される，さまざまなプロダクトは，その性質ならびに開発過程における役割に着目すると，いくつかのグループに分類することができる．ただし，このグループの分類は，開発過程におけるプロセスの入出力の関係とは異なる．ここでは，このグループをプロダクトの型と呼ぶことにする．

まず，前に示したＣプログラム開発の例においては，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．これらのグループは，例をあげると，

① htext型
② ctext型
③ obj型
④ exe型

と，名前をつけることができる．ただし，これらの型の名前は，プロダクトの関係を記述する際に任意に決めることができ，上記のものに固定される必要はない．

3.3 プロダクトの構造体の導入

プロダクトの型の間には互いになんらかの関係がある．このプロダクトの型の間の関係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節に対応していることになる．（図１参照）　構造体はその名前の前に " # " を付けることによりプロダクトの型の名前と区別する．この構造体は次のような形で宣言する．

```
#構造体名　＝　（構造体の構成要素名）
```

構造体の構成要素とは，プロダクトの型の名前，あるいは他の構造体の名前，あるいは，これらの繰り返しによるもである．プロダクトの型の名前が繰り返される場合にはこれを正規表現で表すものとする．

たとえば，０個以上のインクルードファイルがあり，これらのインクルードファイルによって，１つの構造体を作ることを考える．インクルードされるファイルの型の名前を，htext ，構造体の名前を header とすると，この構造体は，

```
#header = (htext*)
```

と，宣言する．さらに，このヘッダファイルの構造体に，Ｃのソースファイル（プロダクトの型を ctext とする）の構成要素を付け加えて新しい構造体 #source を宣言するには，

```
#source =(#header,ctext)
```

とする．同様に，この構造体に，オブジェクトファイル（プロダクトの型は obj）の構成要素を付け加えて，

```
#module(#source,obj)
```

と宣言できる．

これを繰り返すことにより，木を構成する全ての節に構造体を割り当てることができる．図１の木の節に対して構造体の名前を付けたものの例を図２に示す．

3.4 構造体の構成要素に対する作業

開発過程を考えると，プロダクトに対してはなんらかの作業が施され，場合によってはその作業により，別のプロダクトが生成される．このプロダクトに対する作業は普通プロダクトの型を考えると，同一の型のプロダクトには同一の作業が行われることが多い．そこで，プロダクトの関係を表した木構造においては，それらの作業は節，すなわちプロダクトの構造体に対応させて記述することが考えられる．

例えば，3.3で示した例で，#header という構体に着目すると，この構成要素であるヘッダファイルにはエディットとビューいう作業を行うことができる．そこで，これを次のように記述する．

```
#header = (htext*)
    { edit(elementselect(htext*));
      view(elementselect(htext*));};
```

{}内に記述された関数がプロダクトに対する作業を表す．関数が複数ある場合にはどの関数を実行するかは，メニューにより選択実行する．ここで関数 elementselect は，引き数として与えられヘッダファイルのうちの１つをメニューにより選択する関数で，選択された項目名を返すものとする．

同様に，構造体 #source には，次のような作の宣言を記述することができる．

```
#source = (#header,ctext)
  { edit(ctext);
    view(ctext);};
```

構造体 #module に対する作業を宣言する際に，構成要素である obj は，コンパイルによって生されるプロダクトである．よって，この場合，関数の引き数の中に #source.ctext と obj が共に宣言される．ここで，#source.ctext は構造体 #modle を宣言する際に用いた，構造体の #source の構成要素である ctext を示すものとる．

```
#module = (#source,obj)
  { compile(#source.ctext,obj);};
```

図３に，これらの構造体の宣言全体を示す．

4．プロダクトと構造体

4.1 構造体によるプロダクトの関係の宣言

３章で説明した構造体に実際のプロダクトを対応させることを考える．前に示した開発過程でのプロダクトを図３で示した構造体で関係を示すと次のようになる．

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

ここで，プロダクトの宣言は，すべての構造体のプロダクトの型に，実際のプロダクトを対応させることによって行う．また，宣言の中で正規表現により表された構造体の構成要素は（）ではなく {} により表すものとする．

さらに，この宣言はいくつかの名前付きの部分木に分け，これを用いることにより木全体を宣言することもできる．

```
#module:mo1 = (({def.h},test1.c),test1.o);
#module:mo2 = (({def.h,function.h},test2.c
                ,test2.o);
#program = ({mo1,mo2},test);
```

4.2 構造体宣言の汎用性

構造体の宣言は，その関係をプロダクトの型によってこれを表し，さらに構造体の宣言を用いることにより，プロダクト間の関係を表している．したがって，開発過程においてプロダクトに対するプロダクトの型が同じものは，すべて等しい構造体の宣言を用いることができる．例えば，図3で示した構造体を用いて次のような開発過程を宣言する．

『ヘッダファイル def.h とソースファイル test.c からオブジェクトファイル test.o を作り，これと inter.o をリンクして実行型のファイル test を作る』

これは，

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

となる．ここで，inter.o は既に存在するプロダクトであると考えられるので，これを生成するためのソースについての宣言は記述せず，()だけを記述する．

この様に，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，特定の開発過程に限るものではなく，汎用性に富むものである．

5．関係記述からの開発支援

5.1 木構造と開発支援との関係

構造体によって表現された木構造は，ソフトウェア開発におけるプロダクトの依存関係を表すものである．その依存関係を表す木構造は開発において生成される葉（プロダクト）の有無に関係なくあらかじめ枝が構成されている状態にある．

開発途中における木構造は，その葉の部分に対応するプロダクトのうちの一部，あるいは全てが存在しない状態にある．したがって，葉のすべてを生成し，木構造全体を完成させることが，目的とするソフトウェアを生成することになる．

開発過程において，開発が進むにつれその木構造における葉の有無，すなわち木の状態が変化する．よって，木構造を構成する葉（プロダクト）の存在を調べることによって，木の状態を解釈しその解釈によって開発支援を行うことができると考えられる．もしある1つの葉（プロダクト）が存在しなければその葉を含む部分木において既に存在している他の葉によってその葉（プロダクトを生成することができる．したがって，この存在しない葉（プロダクト）の生成を支援することが結果的にソフトウェア開発を支援することになるここで，存在しない葉を含む部分木の，根にあたる節には，その存在しない葉を生成するための作業が対応する．すなわち，開発支援のための1つ1つのプロセス（作業）が木構造におけるそれぞれの節に対応することになる．

5.2 木構造に対するアプローチ

前節で述べた通り，本論文で考える開発支援はプロダクトの関係の記述によって木を構成し，その葉の部分にあたるプロダクトを生成することを支援することによって行う．関係記述によって木構造が記述された場合，その木構造に対するアプローチを2通り考える．それは，(1) トップダウンによる方法と，(2) ボトムアップによる方法である．以下では，この2つのアプローチの方法について順に説明する．

5.2.1 トップダウンによる方法

プロダクトの関係記述によって表された木構造において，その根の部分にあたる節に直接つながる葉（プロダクト）が，最終的に求められているプロダクトである．（図2参照．test がこれにたる）この葉は，根に直接つながる部分木の葉から生成すればよいが，もし，この部分木の葉が存在しないときには，以下の作業を再帰的に行うことにより部分木の葉を生成し，そののちに最終的に目的とする葉（プロダクト）を生成すればよい

(1) 部分木の根に直接つながる葉（プロダクト）が存在するか調べる．

(2) 存在しなければ，その葉を生成するのに必要な葉についてその存在を調べる．もし，必要とする葉のすべてが存在していれば，それらの葉から新しい葉を生成する．逆に，必要な葉が存在していなければ，その葉を含む部分木について，(2) を繰り返す．

(3) 必要となる葉が他の葉を用いないで直接生成される場合には（例えば，ソースファイルなど）

エディタ等を用いて，その葉を生成する．

実際の開発支援では，この部分木における葉の存在を自動的に判別し，葉の生成が必要となった場合には，その生成作業のためのツールを自動的に起動したり，あるいは，その葉を自動的に生成することによって行う．

5.2.2 ボトムアップによる方法

木構造全体を考えると，この木はいくつかの部分木に分割することができる．その部分木を，更にいくつかの部分木に分解していくと，ついにはそれ以上分解することのできない最小単位の木に分けることができる．この最小単位の木は，木構造を表現する構造体を考えたときに，その構成要素がプロダクトの型だけの構造体に対応する．したがって，これら最小単位の木をまず作成し，順次その木を部分木とする木を作成して行くと，最後には木全体が作り上げられて，目的とするプログラムがつくり出される．これがボトムアップによるアプローチである．

ボトムアップによるアプローチも結果的にはトップダウンによるアプローチと同じように思えるが，トップダウンによるアプローチは，必要となる葉が存在していれば，その葉を含む部分木について他の葉が存在していなくても開発を進めていくことができる（例えば，オブジェクトファイルが既に存在している場合，そのソースファイルは不必要である）．しかしながら，ボトムアップの場合には，そのすべての葉を作成していかなければ，開発作業を進めていく事ができない点で異なる（オブジェクトファイルが存在していてもそのソースファイルが存在していなければ，再度，そのソースファイルを作成する必要がある）．

5.3 ＰＤＬへの変換

構造体によって記述されたプロダクトの関係記述はそのままの形では，ＰＤＬインタープリタで解釈実行することができない．そこで，ＰＤＬに一度変換し，ＰＤＬインタープリタによって解釈実行させる．ＰＤＬへの変換は，構造体の１つ１つにメニュー関数を対応させる事によって行い，トップダウンによるアプローチによって行う．これは，一般的な開発過程に対応させるためである．

現在の所では，ＰＤＬへの変換を行うトランスレータを開発することを考えている．トランスレータには構造体の定義と実際の開発で生成されるプロダクトの宣言を入力し，これよりＰＤＬスクリプトを生成するものである．はじめにあげた例を構造体で記述したが，この構造体に対応するように考え，実際にＰＤＬスクリプトに変換したものを図４にあげておく．このＰＤＬスクリプトはインタープリタによって解釈実行させることにより，実際に開発支援環境を構築する事が可能なものである．

6．あとがき

開発過程におけるプロダクトの関係を構造体を用いて記述し，その記述から開発支援環境を構築する方法を述べた．この構造体によるプロダクトの関係記述による方法は，構造体が同一の開発過程においては，プロダクトの宣言の部分を書き換えるだけで記述することができるため，開発過程の記述に関する作業効率が向上し，ＰＤＬに変換されたスクリプトによって，実際の支援環境が容易に構築することができる．

インタープリタによる，関係記述のＰＤＬへの変換は，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ることができると考えられる．

今後の課題としては，今回考えた構造体に対応するようなデータ構造をＰＤＬの中に導入し，直接ＰＤＬによって構造体によるプロダクトの関係記述が表現できるような方法を考えたい．

［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).

表1．主な組み込み関数（抜粋）

関数と引数	結果	意味
1．基本関数		
exec(C,S)	: S	Cシェルコマンドを実行する
wopen(S)	: S	ウィンドウを開く
wclose(S)	: S	ウィンドウを閉じる
2．リスト型処理関数		
head(L)	: X	リストの先頭の要素を取り出す
tail(L)	: L	リストの先頭の要素を取り除いたリストを返す
lcons(X,L)	: L	リストの先頭に要素をつけ加える
3．プロダクト情報を取り出す関数		
exist(C,S)	: B	指定した名前のファイルが存在するかを調べる
time_stamp(C,S)	: I	指定した名前のファイルのタイムスタンプを返す
4．選択機能を実現する関数		
menu(C,L,S)	: S	与えられた文字列をメニューウィンドウに表示して，マウスでその中の一つを選択し，それを返り値とする

＊リストの要素はタプルであり，そのタプルの要素は1番目が文字列，2番目が選択可能／不可能を表す論理型である
L = {[C1,B1],[C2,B2],...}

引数と結果の型は以下のとおりとする
S：システム状態　I：整数　C：文字列
[...]：タプル　B：論理　X：任意　L：リスト

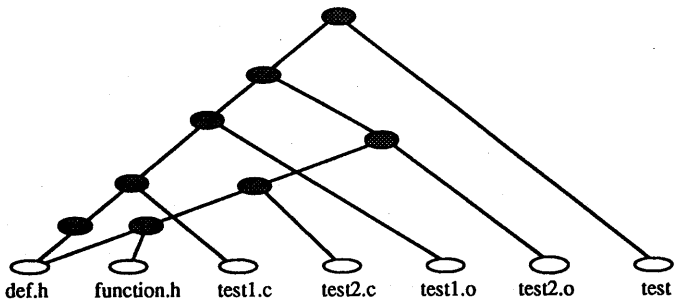

図1. 構造体による木構造

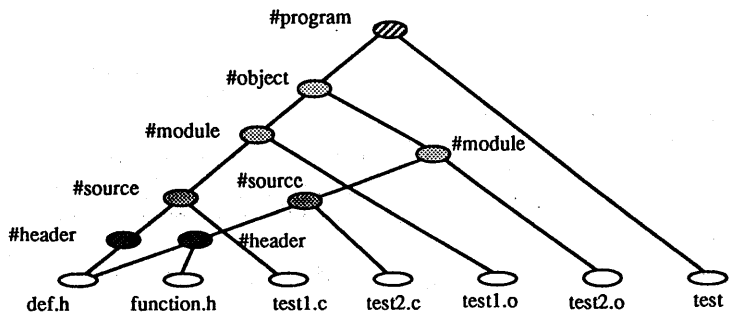

図2．構造体に体する名前

```
#header = (htext*)
 { edit(elementselect(htext*));
   view(elementselect(htext*));
  };
#source = (#header,ctext)
  { edit(ctext);
    view(ctext);
   };
```

```
#module = (#source,obj)
  { compile(#source,obj);};
#object = (#modlue+);
#program = (#objcet,exe)
 { link(elementlist(#object,obj),exe);};
```

図３．構造体の宣言

```
#let   Editor  : "vi "
#let   Viewer  : "more "
#let   Compiler  : "cc -c "
#let   Linker  : "cc -o "

#let   HED1    : "def.h"
#let   HED2    : "function.h"
#let   SRC1    : "test1.c"
#let   SRC2    : "test2.c"
#let   OBJ1    : "test1.o"
#let   OBJ2    : "test2.o"
#let   EXE     : "test"

main(S) == if (exist(OBJ1,S) = true)&(exist(OBJ2,S) = true)
          then programmenu(S)
        else if (exist(OBJ1,S) = false) then main(obj1process(S))
        else if (exist(OBJ2,S) = false) then main(obj2process(S))
                                  else S;

programmenu(S) == menubranch("PDL 'test' Menus ",
      { ["Link",linkchack(S),programmenu(Link(S))],
        ["OBJ1",true,programmenu(obj1menu(S))],
        ["OBJ2",true,programmenu(obj2menu(S))],
        ["EXIT",true,exit(S)]},S);

linkchack(S) ==  if (exist(OBJ1,S) = true)&(exist(OBJ2,S) = true)
          then true else false;

Link(S) == exec(Linker + EXE+" " +OBJ1+" "+OBJ2,S);

obj1process(S) == if (exist(SRC1,S) = true)&(exist(HED1,S) = true)
              then obj1menu(S)
            else if (exist(SRC1,S) = false) then obj1process(SRC1process(S
))
            else if (exist(HED1,S) = false) then obj1process(OBJ1DEF1proce
ss(S))
                                  else S;

obj2process(S) == if (exist(SRC2,S) = true)
                  &(exist(HED1,S) = true)
                  &(exist(HED2,S) = true)
                  then obj2menu(S)
            else if (exist(SRC2,S) = false) then obj2process(SRC2process(S
))
            else if (exist(HED1,S) = false) then obj2process(OBJ2DEF1proce
ss(S))
            else if (exist(HED2,S) = false) then obj2process(DEF2process(S
))
                                  else S;
```

```
obj1menu(S) ==  menubranch("PDL 'test1.o' Menus ",{
      ["Compile",obj1compilechack(S),Compile(SRC1,S)],
      ["SRC1",true,obj1menu(SRC1process(S))],
      ["HED1",true,obj1menu(OBJ1DEF1process(S))],
      ["EXIT",true,main(S)]},S);

obj1compilechack(S) == if (exist(SRC1,S) = true)&(exist(HED1,S) = true)
                then true else false;
obj2menu(S) ==  menubranch("PDL 'test2.o' Menus ",{
      ["Compile",obj2compilechack(S),Compile(SRC2,S)],
      ["SRC2",true,obj2menu(SRC2process(S))],
      ["HED1",true,obj2menu(OBJ2DEF1process(S))],
      ["HED2",true,obj2menu(DEF2process(S))],
      ["EXIT",true,main(S)]},S);

obj2compilechack(S) == if (exist(SRC2,S) = true)
                  &(exist(HED1,S) = true)
                  &(exist(HED2,S) = true)
                 then true else false;

compile(filename,S) == exec(Compiler+filename,S);
Compile(filename,S)==wclose(compile(filename,wopen(S)));
SRC1process(S) == menubranch("PDL 'test1.c' Menus ",{
      ["Edit",true,SRC1process(Edit(SRC1,S))],
      ["view",exist(SRC1,S),SRC1process(View(SRC1,S))],
      ["EXIT",true,obj1process(S)]},S);

SRC2process(S) == menubranch("PDL 'test2.c' Menus ",{
      ["Edit",true,SRC2process(Edit(SRC2,S))],
      ["view",exist(SRC2,S),SRC2process(View(SRC2,S))],
      ["EXIT",true,obj2process(S)]},S);

OBJ1DEF1process(S) == menubranch("PDL 'def.h' Menus ",{
      ["Edit",true,OBJ1DEF1process(Edit(HED1,S))],
      ["view",exist(HED1,S),OBJ1DEF1process(View(HED2,S))],
      ["EXIT",true,obj1process(S)]},S);

OBJ2DEF1process(S) == menubranch("PDL 'def.h' Menus ",{
      ["Edit",true,OBJ2DEF1process(Edit(HED1,S))],
      ["view",exist(HED1,S),OBJ2DEF1process(View(HED2,S))],
      ["EXIT",true,obj2process(S)]},S);

DEF2process(S) == menubranch("PDL 'def.h' Menus ",{
      ["Edit",true,DEF2process(Edit(HED2,S))],
      ["view",exist(HED2,S),(View(HED2,S))],
      ["EXIT",true,obj2process(S)]},S);

edit(filename,S) == exec(Editor+filename,S);
Edit(filename,S)==wclose(edit(filename,wopen(S)));

view(filename,S) == exec(Viewer+filename,S);
View(filename,S) == wclose(view(filename,wopen(S)));
```

図４．変換後のＰＤＬスクリプト