PIDL/Pascal: A Pascal-oriented Program Interface Description Language And Its Applications

Xu Baowen

Department of Computer Science and Engineering
Nanjing Aeronautical Institute
Nanjing, 210016
P. R. China

Abstract

This paper introduces a Pascal-oriented program interface description language, PIDL/Pascal, which is designed to specify the interface information between program units so as to remedy the weakness of some design tools. We will also discuss its design ideas, program structure and facilities, applications, and checking tools.

1. Introduction

In software design stage, one should specify not only the functional description and data structure details of each software component, but also the interfaces between all software components. At present, the design representation tools mainly include flow charts, PAD charts, N-S chart, PDL language and so on. There is a common weakness in these tools, that it is difficult or impossible to use them to specify the interface information between software components, and the definition and use information of an entity (identifier) in a software component, e. g. ,the information where and how a variable must or should be used and where the variable must not be used. For example, in implementing a stack abstraction data type, at least, there should be a procedure PUSH, a procedure or function POP, a function ISEMPTY used to determine whether a stack is empty, a function ISFULL used to determine whether a stack is full, a function LENGTH used to indicate the number of the elements in a stack, a data structure STACK used to represente a stack, and a pointer PTR pointing to the top of a stack. In order to maintain the integrity and consistency of operations on a stack, the other program components outside the stack definition could only use(or call) the subprograms PUSH, POP, ISEMPTY, ISFULL and LENGTH, but may not refer to STACK and PTR directly; moreover, in the body of the function ISEMPTY, ISFULL or LENGTH, STACK may not be used, and the value of PTR may not be changed. All of these are very difficult to be specified in design representation tools or programming languages. Although some of them may be obtained by means of analysing the design or(and) the Pascal code, it is very difficult to analyse a large program, let alone the informal design tools.

For this reason, we developed a Pascal-oriented program interface description language PIDL/Pascal used to specify the interfaces between program components and the definition and use information of an entity (idetitier) in a program component. Each PIDL/Pascal program, including its program units(i. e. , procedures and functions), comprises a definition part and a reference part. The definition part specifies the use information of an entity (identifier) defined or declared at the corresponding place in the relevant PDL program(or flow chart, PAD chart) or programming language program: where and how to use it; the reference part specifies which entities(identifiers), defined or declared in the predefined environment and the other program units(the main program and the subprograms), are used in the current program unit, and how they are used in the current unit.

2. Definition Parts

In a Pascal program (or a PDL/Pascal program or a Pascal-oriented flow chart), if an entity (identifier) E, defined or declared in a subprogram (or the main program) A, is used in another subprogram (or the main program) B, the use approach may fall roughly into three categories as follows:

1. E must be used in B. e. g. , STACK and PTR must be used in subprogram PUSH and POP.

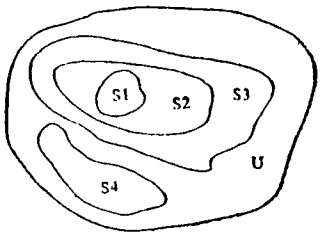
2. E may be used in B, but whether or not E is actually used in B depends on particular implementations. e. g. , PTR may be used in the subprograms ISEMPTY, ISFULL and LENGTH. In the real implementation, if the stack is represented with an array and there is no variable used as element counter, PTR must be used in the three subprograms; if there is a counter, PTR must not be used in the subprogram.

3. E can only be used in a set of subprograms, including B, at most. That is, the use range of E is specified, and in an actual implementation, E may be used in all or some of the subprograms, or even in none of them. e. g. , PTR can only be used in the subprograms PUSH, POP, ISEMPY, ISFULL and LENGTH.

4. E is not allowed in B. e. g. , STACK is not allowed in all subprograms except POP and PUSH.

There is a major difference beetween the second and third category as follows. In the second categoty, the information tells us that an entity(identifier) may be used in other subprograms besides in the specified subprograms. Therefore, this category of information has no use for actual consistency and integrity checks, because it provides no message for automated tools; however, it is very usefull for the program implementation and maintenance, because it can tell the original ideas of the designers to the implementators and maintainers. In the third category, the scope of use for an entity(identifier)is limited to the specified set of subprograms. There is a common characteristic in the two categoried, that is, the entity(identifier) defined may not be used in all of the specified set of subprograms.

Let U be the set of all the subprograms that can use an entity(identifier) E from the angle of the pure scope, S1 be the set of the subprograms that must use E, S2 be the set of the subprograms that may use E, S3 be the set of the subprograms where E can only be used, and S4 be the set of the subprograms where E can not be used, then the following relations hold (see the right figure):



$$S1 \subseteq S2 \subseteq S3 \subseteq U$$
$$S3 \cap S4 = \varnothing$$

In order to be consistent with a Pascal program, the definition part of a PIDL/Pascal program consists of a constant definition part, a type definition part, a variable definition part, and a procedure and function definition part, with each part including a number of definitions. The simplest form of a definition is as follows:

⟨identifier⟩ : = : ⟨reference program unit part⟩

where

⟨reference program unit part⟩ ::=
[⟨must reference program unit group⟩]
[⟨may reference program unit group⟩]
[⟨only reference program unit group⟩]
[⟨must-and-only reference program unit group⟩]
[⟨never reference program unit group⟩]

⟨must reference program unit group⟩ ::=
    [**mustin**](⟨program unit name list⟩);
⟨may reference program unit group⟩ ::=
    mayin(⟨program unit name list⟩);
⟨only reference program unit group⟩ ::=
    onlyin(⟨program unit name list⟩);
⟨must-and-only reference program unit group⟩ ::=
    mustonlyin(⟨program unit name list⟩);
⟨never reference program unit group⟩ ::=
    neverin(⟨program unit name list⟩);

If a program unit name list is composed of all the program units that can use the entity(identifier) defined according to the scope rules, it can be replaced by the reserved word "**all**".

2. 1 Constant Definition Parts

A constant definition specifies where a constant is used.

⟨constant definition part⟩ ::=
    ⟨null⟩
    |**const** ⟨constant definition⟩ { ⟨constant definition⟩ }
⟨constant definition⟩ ::=
    constant ⟨identifier⟩ : = : ⟨reference program unit part⟩

The constant identifiers described here do not include enumeration constant identifiers.

2. 2 Type Definition Parts

A type definition specifies where a type is used.

⟨type definition part⟩ ::=
    ⟨null⟩
    |**type** ⟨type definition⟩ { ⟨type definition⟩ }
⟨type definition⟩ ::=
    type ⟨identifier⟩ : = : ⟨reference program unit part⟩

Enumeration constant identifiers are not involved in both constant definition parts and type definition parts, because it is difficult to design everything in detail; similarly, record fields are also not involved in both type definition parts and variable definition parts below.

2. 3 Variable Definition Parts

A variable definition specifies where and how a variable is used.

⟨variable definition part⟩ ::=
    ⟨null⟩
    |**var** ⟨variable definition⟩ { ⟨variable definition⟩ }
⟨variable definition⟩ ::=
    ⟨variable identifier⟩ : = : ⟨reference program unit part⟩

There are three cases when a variable is used in a program unit: ①it is used as an r-value; ②it is used as an l-value; ③it is used as an l-value and an r-value. It is necessary to distinguish the three cases in program designs. For example, the actions in the subprogram A may be decided by the value of a certain variable assigned in the subprogram B; however, the value of V cannot be changed in A. To this end, the syntax of reference program unit parts in variable definitions is augmented so as to distinguish the three cases: if a variable can only be used as a constant in a certain subprogram, the subprogram name must be followed by the reserved word "**in**"enclosed in parentheses in the reference program unit part of the variable

definition of the variable; if a variable can only be assigned in a certain subprogram (e. g. , the variable can used in the left-hand side of an assignment statement but cannot used in the right-hand side of an assignment statement), the subprogram name must be followed by the reserved word "out" enclosed in parentheses in the reference program unit part of the variable definition of the variable; if a variable can be used in a unrestricted way in a certain subprogram, the subprogram name must be followed by nothing or by the reserved words "in out" enclosed in parentheses. Therefore, a subprogram name (identifier) may appear in more than one program unit groups in the same variable definition. For example,

```
V : = : mustin(P1, P2(in), P3(out));
        neverin(P2(out), P4);
```

The definition specifies that, V must be used as both a l-value and a r-value in P1, as a constant in P2, as a l-value in P3, and cannot used in P4. Please note that this may not happen in a constant definition, a type definition, or a subprogram definition.

2. 4 Procedure And Function Definition Parts

A procedure (or function) definition specifies where a procedure (or function) is used (called). A procedure (or function) definition corresponds to a procedure (or function) declaration in Pascal. A procedure (or function) definition describes not only where a212 procedure (or function), defined at this place in the corresponding Pascal program, is used, but also which entity (identifier) defined externally is used in the procedure (or function), and which entity (identifier) is defined in the body. The syntax of a procedure definition is given as follows:

```
⟨procedure definition⟩ ::=
    procedure ⟨procedure identifier⟩ : = : ⟨reference program unit part⟩
      ⟨reference part⟩
      ⟨parameter part⟩
      ⟨definition part⟩
    end;
```

where

```
⟨parameter part⟩ ::=
    ⟨null⟩
    | parameter
        ⟨parameter definition⟩ {; ⟨parameter definition⟩ };
⟨parameter definition⟩ ::=
    ⟨formal identifier⟩ : = : ⟨reference program unit part⟩
```

The parameter part is put before the definition part in order to be consistent with Pascal language; and it is put after the reference part because the parameter type is predefined or defined externally.

3. Reference Parts

A reference part specifies where and how an entity (identifier), defined in an enclosing program unit or in the predefined environment, is used in the current program unit. That is, it both describes where an entity (identifier), which will be used in the current program unit, is defined, and specifies where and how the entity is used here. Similar to a definition part, a reference part consists of a constant reference part, a type reference part, a variable reference part and a procedure and function reference part, each part including a number of references. The general form of a reference is as follows:

```
⟨identifier⟩ =:: ⟨definition program unit part⟩
              : = : ⟨reference program unit part⟩
```

where the definition program unit part is used to indicate where the specified entity is defined. If the entity

(such as BOOLEAN, PRED, WRITE etc.) is predefined, it is the reserved word "predefined"; if the entity is defined in the other program unit, it is indicated by the name of the program unit; if the entity is defined in the main program, it is indicated by the reserved word " main " or by the name of the main program. for example:

```
① const PI =:: main;
          : = mustin(PROC1,PROC3);
              neverin(PROC2, FUNC6);
② type BOOLEAN =:: predefined;
                 : = : mayin(FUNC5);
```

4. An Example

Let's consider the following PIDL/Pascal program named PROG which will be the name of a corresponding Pascal program. The Pascal program includes a procedure named PROC and a function named FUNC besides the subprograms on stack oprations, and needs to use some predefined types and subprograms, such as integer type INTEGER, real type REAL, boolean type BOOLEAN, input-output procedures READ and WRITELN. The data structure cannot be used in the main program and the subprograms PROC and FUNC. The subprograms used to operate the stack can not be called in main program. FUNC may only call the function ISEMPTY, ISFULL and LENGTH, but PROC can call all the subprograms on stack operations. The PIDL/Pascal program is as follows:

```
program PROG(INPUT, OUTPUT);
  reference
    type
      BOOLEAN =:: predefined
              =:  mustin(ISEMPTY, ISFULL);
                  mayin(PROC, FUNC, PROG);
      INTEGER =:: predefined;
              : = : mustin(PUSH, POP, ISEMPTY, ISFULL, LENGTH);
      REAL =:: predefined;
           : = : mustin(PUSH, POP, PROC, PROG);
                 mayin(FUNC);
      procedure READ =:: predefined;
                     : = : mayin(main , PROC);
                           neverin ( PUSH, POP, ISEMPTY, ISFULL, LENGTH,
                                     FUNC);
      procedure WRITELN =:: predefined;
                        : = : mustin(PROG);
                              onlyin(PROC);
  definition
    var
      STACK : = : mustonlyin(PUSH, POP);
      LEN   : = : mustin(ISFULL, ISEMPTY);
                  onlyin(PUSH, POP);
      PTR   : = : mustonlyin(PUSH, POP, ISEMPTY, ISFULL, LENGTH);
    procedure PUSH : = : mustonlyin(PROC);
      reference
        var
          STACK =:: main; {In this program, main=PROG}
```

```
                    : = : self(out);
          PTR       =:: main;
                    : = : self;
        type
          REAL      =:: predefined;
                    : = : self;
      parameter
        INVALUE     : = : self(in);
    end{PUSH};
    function POP    : = : mustin(PROC);
                          onlyin(main);
      reference
        var
          STACK     =:: main;
                    : = self(in);
          PTR       =:: main;
                    : =:self;
        type
          REAL      =:: predefined;
                    : = : self;
      return REAL;
    end{POP};
    function ISEMPTY : = : mayin(POP, PROC, FUNC);
      ... ...
    end{ISEMPTY};
    function ISFULL : = : mayin(PUSH, PROC, FUNC);
      ... ...
    end{ISFULL};
    function LENGTH : = : onlyin(PROC, FUNC);
      ... ...
    end{LENGTH};
    procedure PROC : = : mustonlyin(main);
      ... ...
    end{PROC};
    function FUNC : = : onlyin(main, PROC);
      ... ...
    end{FUNC};
end{PROG}.
```

There are some points that should be explained with respect to this program:

①PIDL languages is designed to specifies the relations between definitions and references in developing large or middle sized softwares, and are not suitable to be used in small programming, especially in writing exercise programs. This example is used to to explain the use of the language and thus seems to be longer than the corresponding Pascal program. This will not happen in developing large or middle sized softwares; Even though this happns, it would be necessary.

②Although the prgram in this example specifies almost all the definitions and references of all entity idetifiers, it is not required like this in the practical software developments but is required to specify the key

or important entity identifiers in the designs. For example, an entity identifier, defined in a subprogram and used only in the statement part of the subprogram, is unnecessary to appear (in the definition part of the subprogram definition) in the PIDL/Pascal program.

③The procedure reference

```
procedure WRITELN =:: predefined;
                  : = mustin(PROG);
                      onlyin(PROC);
```

is equivalent to

```
procedure WRITELN =:: predefined;
                  : = mustin(PROG);
                      onlyin(PROG, PROC);
```

5. Applications

As a complementary representation tool to other design representation tools(e. g. , PDL/Pascal etc.), PIDL/Pascal language may be used in the design, implementation, analysis and maintenance of softwares, as follows:

5. 1 Design Consistency And Integrity Checking Tool

PIDL/Pascal may be used to check the integrity and consistency between the different parts of a design or code.

5. 2 Design And Code Understanding Tool

PIDL/Pascal formally specifies the definition and reference relations of all entities in a design or code, and hence may be used to understand and analyse designs or codes effectively.

5. 3 Design Complexity Analysing Tool

PIDL/Pascal specifies the interfaces between the main program and the subprograms and between the different subprograms, and hence may be used to analyse and evaluate the interface complexities of a design so as to improve all or some parts of the design.

5. 4 Maintenance Tool

PIDL/Pascal may be used to understand the structures of a design or code, and thus may be used in the software maintenance.

6. Checking Tools

PIDL/Pascal may be used to check two types of consistency, i. e. , the consistency within a design (within a PIDL/Pascal program) and the consistency between a design and its code. In order to check the two types of consistency, we have developed a PIDL/Pascal checking tool.

6. 1 Checking Consistency Within Design

To check the consistency within a design is to check the consistency within the corresponding PIDL/Pascal program. Within a PIDL/Pascal program,

① if the definition part of a subprogram P1, where an entity E is defined, specifies that E must or may be used in another subprogram P2, or that P2 is one of the subprograms where E can only be used, and the reference part of P2 specifies that E must be used there, it is considered to be consistent or compatible;

②if the definition part of P1 specifies that E must be used in P2, and the reference part of P2 specifies that E may be used there, it is considered to be quasi-consistent or not very consistent;

③ if the definition part of P1 specifies that E is not allowed tobe used in P2, but the reference part of

P2 specifies that E must be used there, it is considered to be inconsistent.

There are other possibilities to be checked. When the checking tool runs, it will reports the quasi-consistency and the inconsistency in the design or the Pascal code, which will be referred in checking or modifying the design or the Pascal code.

6. 2 Checking Consistency Between Design And Code

The second type of consistency checking is to check the consistency between the a design and the corresponding the Pascal code, so as to check whether or not the design requirements of the interfaces are met in the Pascal code.

① if the definition part of a subprogram P1, where an entity E is defined, specifies that E must or may be used in another subprogram P2, or that P2 is one of the subprograms where E can only be used (or if the reference part of P2 specifies that E must or may be used there), and E is used in the corresponding place in the Pascal code, it is considered to be consistent or compatible.

② if the definition part of P1 specifies that E may be used in P2 (or if the reference part of P2 specifies that E may be used there), and E is not used in the corresponding place in the Pascal code, it is considered to be quasi-consistent or not very consistent;

③ if the definition part of P1 specifies that E must be used in P2 (or if the reference part of P2 specifies that E must be used there, but E is not used in the corresponding place in the Pascal code, it is considered to be inconsistent or incompatable; if the definition part of P1 specifies that E can only be used in the subprograms P1, P2 and P3, but E is also used in the subprogram P4 in the Pascal code, it is considered to be inconsistent; if the definition part of P1 specifies that E is not allowed to be used in the subprogram P2, but E is used in the corresponding place in the Pascal code, it is also considered to be inconsistent.

Thus, we may see that the case that an entity "may be used" is of no use for checking tools. The case is designed to assist in understanding designs. The following situation may be encountered in designing softwares: it is uncertain whether or not an entity will be used in a subprogram when coding, but according to the known information, the entity may quite be used in the subprogram. If this kind of information is recorded in PIDL/Pascal programs, it is conductive to understanding and maintaining softwares.

References

[1] Jensen, K. et al, Pascal User Manual and Report, 3rd ed., Springer, 1985

[2] Xu, B., CRL/Pascal: A Pascal-oriented Cross Reference Language, to appear.