INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)

A LANGUAGE-ADAPTIVE PROGRAMMING ENVIRONMENT BASED ON A PROGRAM ANALYZER AND A STRUCTURE EDITOR

Takeshi CHUSHO and Tan WATANABE
Systems Development Laboratory, Hitachi, Ltd.
1099, Ohzenji, Asao-ku, Kawasaki 215, Japan

Toshihiro HAYASHI
Omika Works, Hitachi, Ltd.
Hitachi-shi, Ibaraki, Japan

The language-adaptive and methodology-oriented programming environment is a single integrated system based on a common program analyzer which is employed by all programming tools such as a structure editor, a structured programming language processor and a structural testing tool. In particular, the structure editor, PARSE (the Production And Reduction Screen Editor), supports top-down development by stepwise refinement, structured coding and language construct edition. That is, program refinement and local modification are performed using primitive functions, namely, replacement of refinable language constructs, reduction to a nonterminal and insertion of a nonterminal according to the extended production rules, although global modification is performed by dedicated commands.

1. INTRODUCTION

Software productivity has become the critical problem because of recent rapid increases in the amount of software. In the 1970's, considerable effort had been spent on research in software engineering. As a result, new programming methodologies such as program development by stepwise refinement[1,2], data abstraction[3], and other modularization techniques were proposed. Then several new programming languages supporting these methodologies were developed.

However, these new languages reduce the costs by only a small fraction in comparison with total software development costs[4]. Therefore, more attention has been recently directed to an integrated programming environment. Since a conventional programming environment is the assembly of independent programming tools, their effectiveness as a system is often interfered by such obstacles as interface inconsistency and language specification discrepancy.

Effective approaches for overcoming these defects and for improving usability, are considered as follows:

(1) methodology-oriented integration of tools into one system,
(2) adaptation of tools to the objective language.

We call such a system "a language-adaptive programming environment" (LPE). For example, a text editor and a debugger should be replaced by a structure editor and a structural testing tool, respectively. The latter tools are provided with high-level functions by limiting application to a particular language.

In previous studies, several language-oriented systems have been developed such as Interlisp[5], Mentor[6], the Cornell program synthesizer[7] and the incremental programming environment[8]. All of them include both a structure editor and a debugger. However, new programming methodologies are not applicable enough in these systems which support only conventional languages or their dialects. The Ada programming support environment(APSE)[9] may support new methodologies for large-scale software development because these methodologies are incorporated in Ada. Although another language-oriented system, the Iota system[15], supports modular programming, this system is suitable for education, not for large-scale software.

In our previous studies, a structured programming language, SPL[10], and its support tools had been researched. SPL was designed for supporting top-down development by stepwise refinement. The compiler was developed in 1976 and the performance of separate compilation has been further improved[11].

Next, the interactive support system, CROPS, for SPL has been studied. Our early goal was to let programmers make well-structured programs in SPL because a program written in a structured programming language is not always well-structured. Therefore, two-stage programming was proposed[12,13]. That is, first a well-structured program is produced without considering efficiency and then it is optimized at the second stage. Restructuring commands for the first stage are considered high-level structure editor commands.

Based on these studies on SPL, it has been concluded that the aforementioned LPE should be developed at the next stage of software productivity improvement. This is because all of these programming tools for SPL are language-oriented and requires program analyses such as parsing and flow analysis. Thus, LPE for SPL has been designed so that programming support facilities may employ a common program analyzer. Then the structure editor, called PARSE, was developed.

Previous structure editors support syntax check, manipulation of language constructs, or prompting by syntactic templates. In addition, PARSE supports structured programming such as top-down development by stepwise refinement in cooperation with SPL.

This paper describes the conceptual design of LPE, functional design and implementation of PARSE.

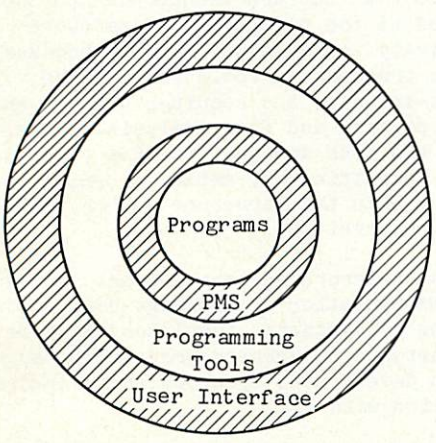
2. CONCEPTUAL DESIGN OF LPE

2.1 Integration of tools

Various programming tools are used in the program development process. However, the effectiveness of the tools as a system is often interfered by such obstacles as inconsistent interfaces, different host machines and discrepant language specifications. Therefore, it is desirable that the programming environment be a single consistent system from the user's viewpoint. For this purpose, all programming tools should be integrated on the basis of the following four items:
(1) a consistent programming methodology,
(2) a common program analyzer,
(3) a common database,
(4) a uniform user interface.
The third and fourth items has been indicated by Howden[14] and others. In addition, the first and second items are also indispensable for the following reasons:

(i) Consistent methodology:
The effectiveness of programming tools is often dependent on input program structures, which are guided by programming methodologies. Therefore, all programming tools should support a consistent methodology. For example, programmers applying modularization techniques, must be confused if a compiler rejects a program composed of many modules or if a debugger does not support module testing. That is, the following three tools are considered as basic tools based on structured programming:
(1) a structure editor,
(2) a structured programming language and its compiler,
(3) a structural testing tool.

(ii) A common program analyzer:
Most high-level programming tools whose input is a source program, need program analysis although such analysis is very costly. For example, the aforementioned three basic tools all require parsing and flow analysis. Therefore, one common program analyzer is positioned at the kernel of the system. Its main functions are as follows:
(1) transformation from a source program to a parse tree,
(2) transformation from this tree to a control flow graph,
(3) data flow analysis,
(4) path analysis and path predicate calculation.
The program analyzer is called the program management system, PMS. Each programming tool is constructed using PMS as shown in Fig. 1.



PMS: Program management system with parsing, flow analysis and path analysis

Fig. 1. A hierarchy of the system's functions.

(iii) A common database:
It is common for output data of a programming tool to become input data of another tool. Therefore, the database is introduced and is consistently managed.

(iv) A uniform user interface:
It is mandatory that user interface including a command language, system messages and terminal operations be uniform among all tools. This enables the programming environment to be a single consistent system from the user's viewpoint.

(v) System configuration:
The system configuration of LPE is based on these design policies and is shown in Fig. 2. Users can perform programming through validation by entering commands from a terminal. Fig. 1 is considered to be the projection of Fig. 2 from the terminal side. The first LPE is being developed for SPL which is widely employed.

2.2 Function classification

Functions of LPE are classified into the following nine groups: O, P, Q, R, S, T, U, V, and W modes. These modes are used usually in this order in the program development process.

(i) Opening mode (O mode):
At the beginning, the following opening operations may be necessary:
(1) allocation of various files used in later modes,

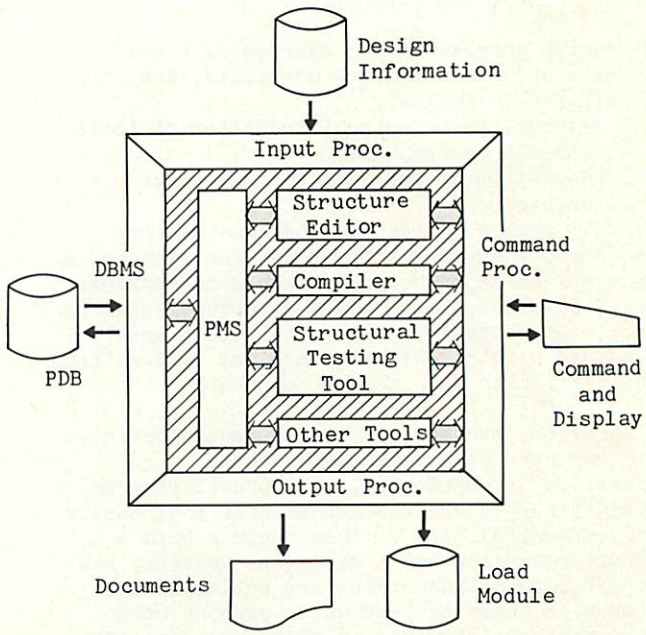


Fig. 2. System configuration of the language-adaptive programming environment.

(2) inclusion of source programs from the external file.
The second item is necessary for programs which are produced without LPE.

(ii) Production mode (P mode):
This mode supports top-down development by stepwise refinement on the basis of handling incomplete and abstract programs. That is, these programs include pseudo-statements and linguistic nonterminals, and can be refined according to extended production rules of a programming language. This mode's functions are described in detail in the next chapter.

(iii) Query mode (Q mode):
Various information about programs is necessary at the time of code review whose purpose is refinement, modification and testing of programs. Therefore, this mode is provided with a query facility. For example, the following information is given using static program analysis facilities of PMS:
(1) intermodule relations such as a call graph,
(2) module interfaces such as parameters and external data access,
(3) scope of identifiers and their declaration-reference-relations,
(4) a control flow graph,
(5) data flow such as definition-use-relations,
(6) program complexity.

(iv) Reduction mode (R mode):
A program is modified for the following reasons:
(1) error correction,
(2) functional modification,
(3) well-structuring,
(4) optimization.
Although the first two items need program testing, the other two items do not if the system verifies the correctness automatically. Therefore, this mode is provided with commands for the following two types of modifications:
(1) local modification such as insertion, deletion and replacement of language constructs,
(2) global modification such as renaming of identifiers, module interface change, inline-substitution of procedure invocation and merging or splitting of loops and procedures.
Local modification is performed by reduction according to the extended production rules which were applied in the production mode. This will be described in detail later. Global modification is performed by a specific command with automatic verification as described in other papers[12,13].

(v) Semantic analysis mode (S mode):
Complete semantic analysis of a program is performed in this mode, although syntactic analysis and local semantic analysis are always performed automatically in the production and reduction modes. This is because it is very time-consuming that semantic analysis of a whole program is performed whenever a small part of the program is refined or modified.

(vi) Testing mode (T mode):
This mode supports the following tasks for program testing:
(1) test data selection,
(2) execution of target programs and result check,
(3) debugging.
The static program analysis facilities of PMS enable this mode to support high-level structural testing and module testing.

(vii) Utility mode (U mode):
Assets of existent programming tools are useful sometimes, especially, in transition to the complete LPE. Therefore, this mode permits the use of exsistent tools.

(viii) Validation mode (V mode):
Software validation is one of the most difficult problems in software development. In this system, programs are validated using various methods based on the following items which are monitored in the previous modes:
(1) Q mode: program complexity,
(2) R mode: frequency of program modification,
(3) S mode: existence of semantic errors,
(4) T mode: the number of detected errors and testing coverage on a control flow graph,
(5) U mode: output of testing tools.

(iv) Writing mode (W mode):
This mode prints out documents required during maintenance time such as source program lists, control flow graphs and module interface lists.

3. STRUCTURE EDITOR FOR STRUCTURED PROGRAMMING

3.1 Design concept of a structure editor

Among the three basic tools of LPE, the structure editor has been first developed in advance of the compiler and the structural testing tool. This is because, instead of the latter tools, respective independent tools are already available. Although, a conventional text editor is available instead of the structure editor, the text editor has a semantic gap defect between thought and operation. That is, a program is regarded as statements containing semantics when it is read. However, the program is regarded as a character string without semantics when it is manipulated by a text editor.

The structure editor raises program manipulation level to thought level and eliminates the semantic gap. Furthermore, this editor supports programming methodology together with the objective language SPL according to the basic policy of methodology consistency. Consequently, this editor is provided with the following functions:
(1) top-down development by stepwise refinement,
(2) structured coding,
(3) language construct edition.

Among the nine modes of LPE, the structure editor includes the first five modes, namely, opening, production, query, reduction and semantic analysis. Then its kernel program was developed, which includes the production and reduction modes and is called PARSE (the Production And Reduction Screen Editor). It is intended that the other functions are added stepwise while accompanying enhancement of PMS. In the remainder of this section, overview of PARSE is described.

(i) Top-down development by stepwise refinement:
In this method, a program at an intermediate step is called the abstract program. PARSE handles the abstract program including the following refinable items:
(1) nonterminals,
(2) pseudo-statements,
(3) procedure references,
(4) user-defined type references.
The last two items are supported in SPL and used for stepwise refinement of procedures and data. An example of an SPL program is shown in Fig. 3. In the process module of the first step, the type STACK and the procedure PUSH are referred to. Then they are refined in the environment module and process module of the second step, respectively.

In addition, nonterminals are introduced so that the body of each module may be refined stepwise. Pseudo-statements are also necessary for general use, which are independent in the objective language. For example, instead of the procedure reference, PUSH (VALUE) TO (S1), in Fig. 3(a), the following pseudo-statement can be written:
"Push down the value to the S1 stack."

PARSE supports stepwise refinement of a program as follows:
(1) The current refinable item (CRE) on the screen is marked by a particular method such as high-lighting or coloring depending on the terminal.
(2) How to refine CRE is displayed as a selection menu,
(3) CRE is replaced by the right-hand side of the corresponding production rule or input text.

```
process P1(E1);
  func EVAL opt(sub);
    var S1:STACK(100);
    .....
     PUSH(VALUE) TO(S1);
    .....
end P1;
```

(a) The process module of the first step

```
environment E2(E1);
  declaration;
    type STACK(LENGTH:int)
              =(TOP:int init(0),
                STK(LENGTH):real,
                MAX:int init(LENGTH));
  end;
end E2;
```

(b) The environment module of the second step

```
process P2(E2);
  func PUSH(V) TO(S) opt(open);
    par V:real,
        S:STACK(*);
     S.TOP=S.TOP+1;
     S.STK(S.TOP)=V;
  end PUSH;
  .....
end P2;
```

(c) The process module of the second step

Fig. 3. An example of an SPL program developed by stepwise refinement.

(ii) Structured coding:
The basic control statements of SPL are limited to those for compound, selection and iteration. For guidance of such structured coding, these control statements are introduced only by replacing <statement> with the right-hand side of the production rule which is called a template. However, the other statements can be introduced by either direct text input or application of a production rule.

(iii) Edition of language constructs:
Basic modifications of language constructs, are performed by two primitive functions, that is, insertion of a nonterminal and reduction to a nonterminal as follows:
(1) Insertion: after a nonterminal is inserted at the cursor position, the nonterminal is refined,
(2) Deletion: after reducing the cursor-positioned part to the corresponding nonterminal, the nonterminal is deleted by inputting the null text, if possible,
(3) Replacement: after reducing the cursor-positioned part to the nonterminal, the nonterminal is refined.

3.2 PARSE commands

How to enter the commands is classified into two styles, namely, (1) a function key and (2) a command name following a period, but they may alter depending on the terminal. In this section, a function key name is expressed by a bold-faced word.

(i) System control:
There are such system control commands as **undo** for cancel of the latest program manipulation, .save for file output of an editted program and .end for system termination.

(ii) Screen control:
The screen is splitted into the upper and lower partitions. The program is displayed in the upper partition. The lower partition is used for man-machine communication such as error messages, selection menus and user input. In addition to screen control of conventional editors, the following language-oriented functions characterize PARSE:
(1) A display part in a program is specified by **display** following a procedure name or a module name.
(2) A procedure or a module can be specified as a scroll unit.
(3) CRE is moved by **CRE-up** and **CRE-down.**

(iii) Program refinement:
First, when CRE is a nonterminal, the right-hand side of the corresponding production rule is displayed. CRE is replaced with the n-th alternative by entering .n. Furthermore, the nonterminal can be directly replaced with input text since PARSE analyzes the syntax. If the input text is null and the nonterminal is optional, it is deleted. The text input facility is efficient for expert programmers. However, text input is not permitted for nonterminals which construct basic program structure, that is, the upper nonterminals in comparison with <specification>, <declaration> and <statement> in a parse tree because of excluding severe grammatical errors. Moreover,

text input of a control statement is not permitted also for supporting structured programming. Instead, template commands are introduced such as .if for an if-statement and .ru for a repeat-until-statement since these statements are frequently used.

Next, when CRE is either a pseudo-statement, a procedure reference or a user-defined type reference, **refine** converts CRE into a comment and inserts the respective nonterminals of <statement> or <user-defined type>. Also, when CRE is either a procedure reference or a user-defined type reference, the body can be defined instead of the inline-refinement. That is, **define** displays the template of a procedure definition or a type definition. Then a user can refine this template. When '.' is entered as a command, this state is terminated and the previous screen is redisplayed. For example, let CRE be the reference to the procedure PUSH in Fig. 3(a), and if **refine** is entered, the reference is replaced by the following:

```
/* PUSH (VALUE) TO (S1) */
 <statement>...
```

However, if **define** is entered, the following template is displayed:

```
func PUSH (<par-name>) TO (<par-name>)
                          [opt(<option>)];
[<declaration>]...
 <statement>...
end PUSH;
```

In each example, the underlined part means a new CRE.

(iv) Program modification:
Basic commands for modification are **reduce** and **insert** whose usage was described previously. The simplest reduction algorithm for **reduce** is to replace the cursor-positioned part with the left-hand side of the corresponding production rule. This means deleting the minimum subtree corresponding to the cursor-positioned part in the parse tree which is the internal representation of a user program. This algorithm, however, is not practical because, for example, **reduce** must be repeatedly entered for deletion of a statement.

Thus PARSE adopts the following algorithm. A parse tree is denoted by T(N,A), where N is a set of nodes and A is a set of arcs directed from the root to the leaves. Each node has an attribute which is TT for a terminal symbol node, NT for a nonterminal symbol node with text input permission or NN for a nonterminal symbol node without it. Some notations are introduced here with respect to node x, that is, ATTR(x) for an attribute of x, SUC(x) for a set of successor nodes of x and PRED(x) for the predecessor node of x. For function f, $f^*(x)$ is defined recursively:

$$f^*(x) = \{p \mid p \in f(x) \vee (p \in f(q) \text{ for } \forall q \in f^*(x))\}.$$

The reduction algorithm is as follows:
(1) Let c be the node corresponding to the cursor position,
(2) If ATTR(PRED(c))=NN, delete the subtree whose root is PRED(c) while leaving PRED(c),
(3) If not, node r is found on the following condition:

$$\exists q,r \in \text{PRED}^*(c) \text{ and } q=\text{PRED}(r) \wedge \text{ATTR}(q)=\text{NN} \wedge \text{ATTR}(r)=\text{NT},$$

and then, delete the subtree whose root is r while leaving r.

The second step means that the minimum subtree including the cursor position is replaced by the root node whose attribute is NN. The third step means that the maximum subtree whose root has an attribute of NT, is replaced by the root among subtrees including the cursor position.

This algorithm has an opposite defect of the simplest algorithm. That is, for example, a statement must be reproduced from <statement> even for spelling corrections. Therefore, PARSE displays the deleted part in the text input area so that a user can modify it by text edition and re-enter it. This is considered to be a hybrid method for a structure editor and a text editor.

Next, **insert** inserts all nonterminals which can be grammatically inserted at the cursor position. The insertion algorithm in a parse tree is as follows, where L(x) and R(x) denote the left-most and right-most successor nodes of node x, respectively:
(1) Let c be the node at the cursor position or directly after that if the position is space, and let b be the node directly before the cursor position,
(2) For $\forall p \in \{p \mid c \in L^*(p)\}$, insert optional successor nodes of p to the left side of L(p), if possible,
(3) For $\forall p \in \{p \mid b \in R^*(p)\}$, insert optional successor nodes of p to the right side of R(p), if possible,
(4) For p satisfying the following condition:

$$\exists q,r \in \text{SUC}(p) \text{ and } b \in R^*(q) \wedge c \in L^*(r),$$

insert optional successor nodes of p between q and r, if possible.

4. IMPLEMENTATION

4.1 Localization of language dependency

Language specification is written in the form of BNF and is automatically transformed into internal representation which is called the BNF table. PARSE refers to this table for the following language-dependent processing:
(1) a selection menu display for prompting to refine CRE,
(2) parse tree manipulation by **refine**, **define**, **reduce** and **insert**,
(3) parsing of input text,
(4) pretty printing by an unparser.

For these uses, the BNF table has several features. For example, a production rule is expressed nonrecursively with such metasymbols as ... for iteration and [] for option. The number of nonterminals is minimized for reduction of the parse tree depth, that is, for operability. Pretty printing rules are automatically added to the BNF table, because the input of the BNF table generator is written in screen display form. However, template commands, a comment format and a line format are processed by action routines because they are language-dependent but are not included in the BNF table.

4.2 Parsing algorithm

Input text is parsed while referring to the BNF table. Input parameters of the parser are the nonterminal number LNO and the text pointer PTR. Output parameters are SUC for result and TREE transformed from the text. In the following algorithm, the procedure PP is introduced for partial parsing, RNO implies one of alternative production rules for LNO, and CNO is the component number in the right-hand side:

```
procedure PARSER(LNO,PTR,SUC,TREE);
  {initialize TREE}
  PP(LNO,PTR,SUC);
  if {PTR is not empty} then SUC:=false;
end;
procedure PP(LNO,PTR,SUC);
  case {attribute of LNO} of
  TT: {individual parsing}
  NN: {error handling}
  NT: {save TREE and PTR}; RNO:=1;
      while {RNO is valid} do
        CNO:=1;
        while {CNO is valid} do
          if {CNO is terminal}
            then if {CNO matches to text}
                    then {add to TREE}
                         {advance PTR}
                         goto LP2;
            else PP( {LNO of CNO},PTR,SUC)
                 if SUC then goto LP2;
          if {CNO is not option} then goto LP1;
          LP2: CNO:=CNO+1;
        endwhile;
        SUC:=true; return;
        LP1:{recover TREE and PTR}; RNO:=RNO+1;
      endwhile;
      SUC:=false;
  endcase;
end;
```

Furthermore, additional procedures are required, such as for iterative components marked with ..., suitable error messages and exception handling.

4.3 Extensibility and transportability

For easy functional extention, support of different terminals and transportation to different host machines, corresponding processing is localized. Moreover, the system is written in Pascal. The first version of PARSE is executed on the TSS system of the Hitachi M series computer.

5. CONCLUSIONS

The language-adaptive and methodology-oriented programming environment was designed on the basis of our previous studies on a structured programming language and its support tools. In this system, all programming facilities employ a common program analyzer and consistently support new programming methodologies.

Among the basic tools, the structure editor, PARSE, was newly developed. The first version supports top-down development by stepwise refinement, structured coding and language construct edition. In particular, program refinement and local modification are performed by primitive functions, namely, replacement of refinable language constructs, reduction to a nonterminal and insertion of a nonterminal according to the extended production rules, although global modification is performed by dedicated commands. In the implementation of PARSE, language dependency was localized in the BNF table for extensibility and transportability.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to Dr. Takeo Miura and Dr. Jun Kawasaki for his valuable suggestions and advice. They are also indebted to Atushi Tanaka, Yuki Takahashi and Kiyomi Mori for implementing PARSE.

REFERENCES

[1] E.W.Dijkstra, Notes on structured programming, Strustured programming, Academic Press, London and New York, 1972,1-82.

[2] N.Wirth, Program development by stepwise refinement, CACM, vol.14, no.4, April 1971, 221-227.

[3] B.Liskov et al., Abstraction mechanism in CLU, CACM, vol.20, no.8, August 1977, 564-576.

[4] T.A.Standish, The importance of ADA programming support environments, Proc. NCC'82, 1982, 333-339.

[5] W.Teitleman et al., The Interlisp programming environment, Computer, vol.14, no. 4, 25-32.

[6] V.Donzeau-Gouge et al., A structure oriented program editor: a first step towards computer assisted programming, IRIA Research Report, no.114, 1975.

[7] T.Teitelbaum et al., The Cornell program synthesizer: a syntax-directed programming environment, CACM, vol.24, no.9, September 1981, 563-573.

[8] R.Medina-Mora et al., An incremental programming environment, IEEE Transactions on Software Engineering, vol.SE-7, no.5, September 1981, 472-482.

[9] Requirements for Ada programming support environments "STONEMAN", US Dept. of Defense, February 1980.

[10] T,Chusho et al., A language with modified block structure for data abstraction and stepwise refinement, Proc. RIMS Simposium on 3rd Mathematical Methods in Software Science and Engineering, 1981, 156-173.

[11] T.Chusho et al., Performance analyses of paging algorithms for compilation of a highly modularized program, IEEE Transaction on Software Engineering, vol. SE-7, no.2, March 1981, 248-254.

[12] T.Chusho et al., Two-stage programming: interactive optimization after structured programming, Proc. the third USA-Japan Computer Conference, 1978, 171-175.

[13] T.Chusho, A good program = a structured program + optimization commands, Proc. IFIP'80, 1980, 269-274.

[14] W.E.Howden, Contemporary software development environments, CACM; vol.25, no.5, May 1982, 318-329.

[15] R.Nakajima et al., The Iota programming system - a support system for hierarchical and modular programming, Proc. IFIP'80, 1980, 299-304.

REPRINTED FROM:

INFORMATION PROCESSING 83

Proceedings of the IFIP 9th World Computer Congress
Paris, France,
September 19-23, 1983

Edited by

R. E. A. MASON
Institute of Computer Science
University of Guelph
Guelph, Ontario, Canada

PROGRAM COMMITTEE

D. C. Tsichritzis *(Chairman)*
F. H. Sumner *(Past Chairman, Vice-Chairman)*, R. E. A. Mason *(Editor)*,
J. Arsac *(Organizing Committee Liaison)*, H. Schorr, D. Bjørner, V. E. Kotov, W. M. Newman,
J. W. Schmidt, G. Capriz, N. Naffah, R. Mori, C. J. van Rijsbergen, T. Ohlin

1983

NORTH-HOLLAND
AMSTERDAM • NEW YORK • OXFORD