CONFERENCE RECORD

SEVENTH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS, AND COMPUTERS

EDITED BY

Sydney R. Parker
Naval Postgraduate School
Monterey, California

PAPERS PRESENTED

TUESDAY THROUGH THURSDAY NOVEMBER 27-29, 1973
ASILOMAR HOTEL & CONFERENCE GROUNDS – PACIFIC GROVE, CALIFORNIA

Sponsored by

NAVAL POSTGRADUATE SCHOOL
Monterey, California

UNIVERSITY OF SANTA CLARA
Santa Clara, California

With The Participation Of The

UNIVERSITY OF CALIFORNIA
Davis, California

UNIVERSITY OF CALIFORNIA
Los Angeles, California

UNIVERSITY OF CALIFORNIA
Santa Barbara, California

CALIFORNIA STATE UNIVERSITY
Sacramento, California

STANFORD UNIVERSITY
Stanford, California

UNIVERSITY OF SOUTHERN CALIFORNIA
Los Angeles, California

UNIVERSITY OF ARIZONA
Tucson, Arizona

WASHINGTON STATE UNIVERSITY
Pullman, Washington

In Cooperation With

IEEE SAN FRANCISCO SECTION, IEEE CIRCUITS AND SYSTEMS SOCIETY, AND IEEE CONTROL SYSTEM SOCIETY

CONFERENCE COMMITTEE

Chairman: J. Choma, Jr., Hewlett-Packard Company
Technical Program: T.E. Fanshier, Lockheed Missiles and Space Company
T.J. Higgins, University of Wisconsin
L.P. Huelsman, University of Arizona
L.P. McNamee, University of California, Los Angeles
T. Mills, National Semiconductor Corporation
J.G. Simes, California State University, Sacramento
A. Willson, University of California, Los Angeles
Publication/Finance: S.R. Parker, Naval Postgraduate School
Publicity: S.G. Chan, Naval Postgraduate School

CONFERENCE RECORD

SEVENTH ASILOMAR CONFERENCE ON CIRCUITS, SYSTEMS, AND COMPUTERS

EDITED BY

Sydney R. Parker
Naval Postgraduate School
Monterey, California

PAPERS PRESENTED

TUESDAY THROUGH THURSDAY NOVEMBER 27-29, 1973
ASILOMAR HOTEL & CONFERENCE GROUNDS – PACIFIC GROVE, CALIFORNIA

Sponsored by

NAVAL POSTGRADUATE SCHOOL
Monterey, California

UNIVERSITY OF SANTA CLARA
Santa Clara, California

With The Participation Of The

UNIVERSITY OF CALIFORNIA
Davis, California

UNIVERSITY OF CALIFORNIA
Los Angeles, California

UNIVERSITY OF CALIFORNIA
Santa Barbara, California

CALIFORNIA STATE UNIVERSITY
Sacramento, California

STANFORD UNIVERSITY
Stanford, California

UNIVERSITY OF SOUTHERN CALIFORNIA
Los Angeles, California

UNIVERSITY OF ARIZONA
Tucson, Arizona

WASHINGTON STATE UNIVERSITY
Pullman, Washington

In Cooperation With

IEEE SAN FRANCISCO SECTION, IEEE CIRCUITS AND SYSTEMS SOCIETY, AND IEEE CONTROL SYSTEM SOCIETY

CONFERENCE COMMITTEE

Chairman:	J. Choma, Jr., Hewlett-Packard Company
Technical Program:	T.E. Fanshier, Lockheed Missiles and Space Company T.J. Higgins, University of Wisconsin L.P. Huelsman, University of Arizona L.P. McNamee, University of California, Los Angeles T. Mills, National Semiconductor Corporation J.G. Simes, California State University, Sacramento A. Willson, University of California, Los Angeles
Publication/Finance:	S.R. Parker, Naval Postgraduate School
Publicity:	S.G. Chan, Naval Postgraduate School

AN EXPERIMENT WITH METHODOLOGIES FOR THE SPECIFICATION OF PROGRAMMING SYSTEMS†

Carlos J. Lucena*
Computer Science Department
University of California, Los Angeles
and
Gerry E. Tompkins*
Graduate School of Management
University of California, Los Angeles

Abstract

An experiment with programming is used as the basis for discussion of such topics as: formalisms for systems specification, hierarchical structure of systems and modularity and approaches for testing and debugging.

1. INTRODUCTION

The objective of this research was to analyze the possible impact of a number of emerging methodologies on the design and production of software. The current high cost of software indicates that considerable methodological improvements for software production are critical factors for the development of computer technology.

Although there are some programming experiments reported in the literature [1,2] none seems to approach the variety of aspects that are dealt with by this paper.

When developing a programming system from a natural language statement about an application problem a designer typically concerns himself with assuring a number of properties of the system. One usefull classification for those properties is performance, correctness and structural suitability. In fact, the system is expected to work according to certain efficiency parameters, to reproduce the natural language statement of the problem with fidelity and to have certain structural attributes such as being modifiable, adaptable, etc. To formulate a set of synthesis rules and supply the convenient tools to carry on this task effectively is not easy, particularly when we deal with large systems. Among the possible questions to be answered for the achievement of this goal, the following were explored during our experiment: the need and characteristics of a formalism that can be used from the problem definition to the program implementation; hierarchical structure of programming systems versus modularity; programming languages characteristics and feature utilization and approaches for program testing and debugging. The emphasis of the following discussions will be placed on systems which are required to be correct (in the sense mentioned above) and possess the structural characteristics of being changeable and adaptable.

2. THE PROBLEMS AND THE APPROACH TO THEIR SOLUTION

Our experimental problem is a simple MIS in which its programs solve some classical bookkeeping functions which exist in any academic administration. The example was considered convenient for the experiment due to its simplicity and representativety of data processing problems. The system can be schematically described through the steps in figure 1.

The breakdown of the programming system in the steps above illustrates the fact that five (one systems sort program) absolute modules could be used in the design. An absolute module is a program which communicates with other programs via a standard interface provided for by physical media. In absolute modules the synthesis rules for software modules can be satisfied [3,4] through the following requirements:

(1) Syntatic non-interference (modules can be put together without changes).

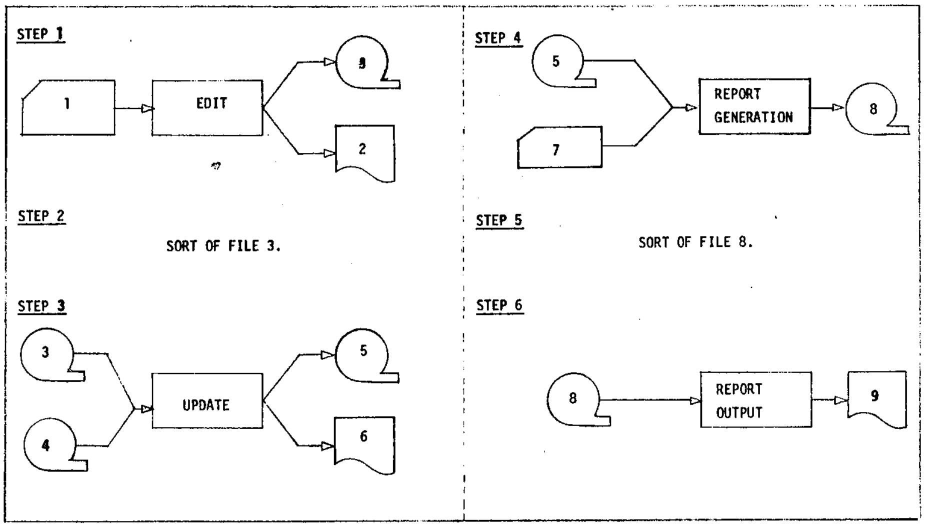
(2) Semantic context independence (modules can be tested independently).

(3) Data generality (module writers can define arbitrary data structures and communicate through them).

This, of course, reduces the original design and implementation problem to a set of smaller sub-problems (one for each absolute module) each to be model-

---

†This research was supported in part by the U. S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and the Brazilian National Research Council.
*On leave from Pontificia Universidade Catolica do Rio de Janeiro.



1. Students transaction cards
2. Edit report
3. Transaction file
4. Main student file (old)
5. Main student file (new)
6. Update report
7. Map request
8. Map File
9. Map Output

Figure 1. Outline of the example used for the experiment

ed by a programming sub-system. For each sub-system an attempt was made to identify software modules that could be implemented by different programmers.

The specification of each sub-system was expressed in terms of a specification language. For this purpose we adopted the notation proposed by Hoare in [5]. The characteristics we looked for in this notation were:

- a highly declarative language in which most of the problems could be absorbed by the data structure declarations.
- a language where the data structures utilized are of a higher level than those currently found in most programming languages (set theoretical data structures).

As it stands, the notation adopted constitutes a good approximation of these goals. In the specification language utilized, the designer specifies its own types and admissible domains and ranges of variables through enumeration, ordered enumeration and sub-ranges definitions. The data structures comprise structures such as Cartesian Products (multi-valued elements), Discriminated Union of Sets, Generalized Arrays, Powersets and Sequences. The examples below illustrate the referred entities.

```
Cartesian Product

(dsd) type A=(B;C);
(dsd) type B=2...7;
(dsd) type C=(A,B,C);  (vd)x:A;  (lvv)→(3,B),(2,A)....
Union
(dsd) type D=(B,C);  (vd)x:D;  (lvv)→(2) or (A) or
                                     (3) or (C) etc.
Powerset
(dsd) type E=powerset (C);
                (vd)x:E;  (lvv)→(φ),(A),(AB),(AC)....
Generalized Arrays
(dsd) type F=array B of C;
                (vd)x:F;  (lvv)→x[2]:=A....
Sequence
(dsd) type G=sequence A;
                (vd)x:G;  (lvv)→x:=(3,B)||(2,A)....
(dsd) - data structure defition
(vd)  - variable definition
(lvv) - legal value of the variable
```

Figure 2. Data Structures in the Specification Language

The control structures used in conjunction with the the mentioned data structures include such statements as: <u>if then else</u>, <u>do while</u>, <u>case</u> and <u>for</u>. The synthesis steps adopted for the construction of each sub-system were the following:

1. Characterization and definition, through declarations in the specification languages, of all the input and output variables and associated domins and ranges.
2. Selection of the control and data flows of each algorithm taking into consideration the "well-formation" of its structure and in particular the issue of modularity. For this purpose alternative graphs of computation were built for each sub-system until a "good structure" was found. In figure 3 one of

such graphs is shown for the sub-system that models the step 3 of the problem. The algorithm shown is the well known balance line for the update of an old file based on information from a transaction file.

3. After achieving a "good structure" the subsystem is completely described in terms of the specification language.
4. Each module within a sub-system was coded and tested independently of the others. The programming language used for implementation was PL/I. Each module definition was comprised of the specification text and a set of test cases expressed in terms of the variables defined in the specification.

Figure 4 displays the specification and implementation texts in its final form. The specification, which has no associated language processor, appears as comments to the PL/I program. The example is shown just to give the flavor of the notation and illustrate the aspect of the final software produced. We are choosing to show only some of the programs corresponding to step 3 which, in turn, is the smallest sub-system in the system.

The design and implementation strategies adopted are summarized and discussed in the next section. Through their application we were able to generate a program which was demonstrated to be easily changeable and which over extensive use after testing revealed no errors. The effort placed in the design phase did reduce the testing and debugging efforts. Although we do not claim that this methodology is optimal in any sense, its application did highlight a number of interesting topics that we are presently pursuing in our research.

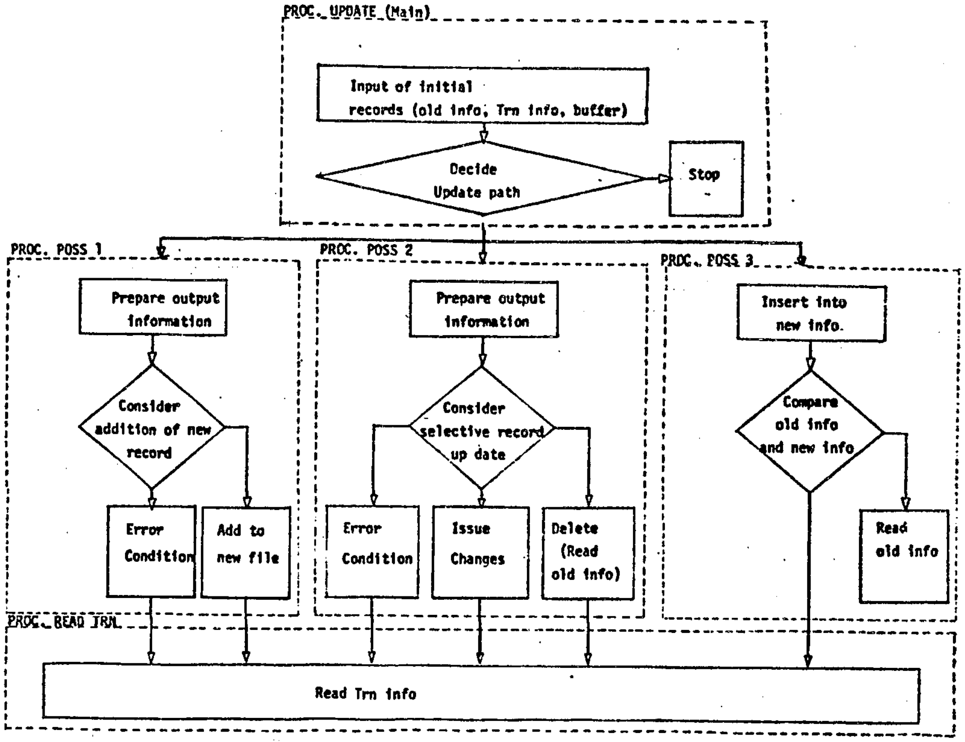


Figure 3. Graph Model associated to step 3 of the problem.

2. COMMENTS ON PROGRAMMING METHODOLOGIES

The design and construction of the software system described was carried on with a critical attitude towards the methodologies applied. We believe that the attitude itself of attempting to build a "good" programming system is responsible for a very high percentage of the quality we achieved for the system.

Despite the fact that we decided to break this section into two major topics, the reader should be aware that there exists a complete interaction between these and that the understanding of this interaction is a fundamental issue.

2.1 Hierarchical Structures and Modularity

Enough has been written about the advantages of imposing a hierarchical structure on a programming system. We contend that its most relevant advantage is the possibility of applying informal proofs of correctness to the system both at the design and implementation levels. This advantage is followed by the possibility of investigating duplication of efforts at the same and different levels of the systems tree. The order in which the design tree is produced is altogether a different matter. We do not think that the top down order of design is a natural or, much less, an inseparable companion of hierarchical structures. At this point we quote

```
PROCEDURE UPDATE(OLDINFO,TRNINFO)

     'TYPE' MAIN_STUDENT_FILE = 'SEQUENCE' STUDENT_FILE;
     'TYPE' STUDENT_TRANSACTIONS_FILE='SEQUENCE'
                                       STUDENT_TRANSACTION;
     'TYPE' STUDENT_FILE=(STUD_RANGE;NAMES;ADDRESS;DATE;
                          SOCSECNUM;COLLEGE;STATE;COUNTRY;
                          NEWDATA;OLDDATA);
     'TYPE' STUDENT_TRANSACTION=(STUD_RANGE;CODERANGE;QUESTION;
                                 NAMES;ADDRESS;DATE;SOCSECNUM;
                                 COLLEGE;STATE;COUNTRY;OLDDATA;
                                 NEWDATA);
     'TYPE' STUD_RANGE=00000001...99999999;
     'TYPE' CODERANGE=1...4;
     'TYPE' QUESTION=1...30;
     'TYPE' NAMES=(FIRSTNAME;MIDDLENAME;LASTNAME);
     'TYPE' ADDRESS=(STREET;CITY;ZIPCODE;PHONE);
     'TYPE' DATE=(CERTAIN_MONTHS;CERTAIN_YEARS);
     'TYPE' SOCSECNUM=000000001...999999999;
     'TYPE' COLLEGE,STATE,COUNTRY='SEQUENCE' LETTER;
     'TYPE' OLDDATA=0...26;
     'TYPE' NEWDATA=0...26;
     'TYPE' FIRSTNAME,MIDDLENAME,LASTNAME='SEQUENCE' LETTER;
     'TYPE' STREET=(DIGIT;DIGLET);
     'TYPE' CITY='SEQUENCE' LETTER;
     'TYPE' ZIPCODE,PHONE='SEQUENCE' DIGIT;
     'TYPE' LETTER=(A,B,C,...,Z);
     'TYPE' DIGIT=(0,1,2,3,4,5,6,7,8,9);
     'TYPE' DIGLET='SEQUENCE' (DIGIT,LETTER);

     OLDINFO,NEWINFO: MAIN_STUDENT_FILE;
     TRNINFO: STUDENT_TRANSACTIONS_FILE;
     OLD_STUD: STUDENT_FILE;
     TRN_STUD: STUDENT_TRANSACTION;
     BUFFER: STUDENT_FILE;
     STUD_CODE: STUD_RANGE;
UPDATE: PROCEDURE OPTIONS(MAIN);

DCL 1 OLD_STUD1                    CHAR(240);
DCL 1 OLD_STUD DEFINED OLD_STUD1,
      2 STUD_CODE                  PIC '(8)9',
      2 REST                       PIC '(232)X';
DCL 1 BUFFER1                      CHAR(240);
DCL 1 BUFFER DEFINED BUFFER1,
      2 STUD_CODE                  PIC '(8)9',
      2 REST                       PIC '(232)X';
DCL 1 TRN_STUD1                    CHAR(80);
DCL 1 TRN_STUD DEFINED TRN_STUD1,
      2 STUD_CODE                  PIC '(8)9',
      2 FILLER                     PIC '(72)X';
DCL LINE1(132)                     CHAR(1) INIT((132)' ');

DCL LINE2(132)                     CHAR(1) INIT((132)' ');
DCL PAGENO                         PIC '9999';
DCL SWITCH                         PIC '9';
DCL OLDINFO FILE RECORD INPUT,
    TRNINFO FILE RECORD INPUT,
    NEWINFO FILE RECORD OUTPUT;
 ON ENDFILE(TRNINFO) TRN_STUD.STUD_CODE = 99999999;
 ON ENDFILE(OLDINFO) OLD_STUD.STUD_CODE = 99999999;

          TRN_STUD 'FROM' TRNINFO;
          OLD_STUD 'FROM' OLDINFO;
          BUFFER:=OLD_STUD;
          'DO WHILE' TRN_STUD.STUD_CODE = BUFFER.STUD_CODE ¬=
                                          99999999;
             'IF' TRN_STUD.STUD_CODE < BUFFER.STUD_CODE 'THEN'
                  POSS1(OLD_STUD,BUFFER,TRN_STUD);
             'ELSE' 'IF' TRN_STUD.STUD_CODE = BUFFER.STUD_CODE
                         'THEN' POSS2(OLD_STUD,BUFFER,TRN_STUD);
             'ELSE' POSS3(OLD_STUD,BUFFER,TRN_STUD);
          'END';
   'END';

        SWITCH = 1;
        CALL PRINT(LINE1,LINE2,SWITCH);
        OPEN FILE(OLDINFO),
             FILE(TRNINFO),
             FILE(NEWINFO);
        READ FILE(OLDINFO) INTO (OLD_STUD1);
        READ FILE(TRNINFO) INTO (TRN_STUD1);
        BUFFER =  OLD_STUD;
        DO WHILE
          ((TRN_STUD.STUD_CODE ¬= 99999999) |
           (  BUFFER.STUD_CODE ¬= 99999999));
             IF TRN_STUD.STUD_CODE < BUFFER.STUD_CODE THEN
                CALL POSS1(OLD_STUD1,BUFFER1,TRN_STUD1);
                ELSE IF TRN_STUD.STUD_CODE = BUFFER.STUD_CODE THEN
                     CALL POSS2(OLD_STUD1,BUFFER1,TRN_STUD1);
                     ELSE IF TRN_STUD.STUD_CODE > BUFFER.STUD_CODE
                          THEN
                          CALL POSS3(OLD_STUD1,BUFFER1,TRN_STUD1);

        END;
        CLOSE FILE(OLDINFO), FILE(TRNINFO), FILE(NEWINFO);
```

```
PROCEDURE POSS1(OLD_STUD,BUFFER,TRN_STUD)


      NAME:NAMES;
      CARDCODE:CODERANGE;
      STUD_CODE:STUD_RANGE;
      'WITH' TRN_STUD 'DO' 'CASE' CARDCODE 'OF'
              (01):'BEGIN' BUFFER.STUD_CODE=TRN_STUD.STUD_CODE;
                           BUFFER.NAME=TRN_STUD.NAME;
                 'END';
                 'ELSE' ERROR MESSAGE);
      READ(TRN_STUD);
      'END';

POSS1: PROCEDURE(OLDSTUDENT,BUFFER,TRNSTUDENT);
       DCL 1 OLDSTUDENT          CHAR(240);
       DCL 1 OLD_STUD1 DEFINED OLDSTUDENT,
             2 STUDID            PIC '(8)9',
             2 FILLER            PIC 'XX',
             2 DATE1             PIC '9999',
             2 NAME,
               3 LAST            PIC '(30)X',
               3 MIDDLE          PIC 'X',
               3 FIRST           PIC '(14)X',
             2 REST              PIC '(181)X';
       DCL 1 BUFFER              CHAR(240);
       DCL 1 BUFFER1 DEFINED BUFFER,
             2 STUDID            PIC '(8)9',
             2 FILLER            PIC 'XX',
             2 DATE1             PIC '9999',
             2 NAME,
               3 LAST            PIC '(30)X',
               3 MIDDLE          PIC 'X',
               3 FIRST           PIC '(14)X',
             2 REST              PIC '(181)X';
       DCL 1 BUFFER3 DEFINED BUFFER,
             2 BUFFER2(240) CHAR(1);
       DCL 1 TRNSTUDENT          CHAR(80);
       DCL 1 TRNSTUD1 DEFINED TRNSTUDENT,
             2 STUDID    PIC '(8)9',
             2 CARD_CODE PIC 'Z9',
             2 QUESTION  PIC 'Z9',
             2 NAME,
               3 FIRST   PIC '(14)X',
               3 MIDDLE  PIC 'X',
               3 LAST    PIC '(30)X',
             2 FILL1     PIC '(23)X';
      DCL 1 LINE1,
            2 BLANKS(132)        CHAR(1) INIT((132)' ');
      DCL 1 LINE1_O DEFINED LINE1,
            2 P_STUD_ID          PIC '(8)9',
            2 P_CARD_CODE        PIC 'BBB99',
            2 P_QUESTION         PIC '(6)B99',
            2 X                  PIC '(18)X',
            2 P_FIRST            PIC '(15)X',
            2 P_MIDDLE           PIC 'X',
            2 X3                 PIC '(48)X',
            2 P_ERROR1           PIC '(29)X';
      DCL 1 LINE2,
            2 BLANKS(132)        CHAR(1) INIT((132)' ');
      DCL 1 LINE2_O DEFINED LINE2,
            2 X4                 PIC '(39)X',
            2 P_LAST1            PIC '(30)X',
            2 X5                 PIC '(34)X',
            2 P_ERROR2           PIC '(29)X';
      DCL SWITCH                 PIC '9';

      LINE1, LINE2 = ' ';
      P_STUD_ID    = TRNSTUD1.STUDID;
      P_CARD_CODE  = CARD_CODE;
      P_QUESTION   =  QUESTION;
      P_FIRST      =  TRNSTUD1.FIRST;
      P_MIDDLE     =  TRNSTUD1.MIDDLE;
      P_LAST1      =  TRNSTUD1.LAST;
      IF CARD_CODE = 1 THEN
         DO;
             BUFFER2 = ' ';
             BUFFER1 = TRNSTUD1, BY NAME;
         END;
         ELSE P_ERROR1 = 'STUDENT NOT IN FILE';
      SWITCH = 0;
      CALL PRINT(LINE1,LINE2,SWITCH);
      CALL READTRN(TRNSTUDENT);
   END POSS1;
```

Figure 4. Specification and Implementation texts for Part of Step 3

Goos: "design without iteration looks like throwing a ball into a hole; whether we succeed depends on the size of the hole and of the ball as well as on our knowledge about the position of the hole and our experience in throwing." An interesting discussion about order of design decisions can be found in [6]. We can illustrate this point through a situation that we observed during our experiment. One argument in favor of "pure" top-down design is that the if then else - do while - sequence Type of control structure completely takes care of the dimension of control in an informal definition of well structured sequential programs. It means that from the point of view of testability an ideal program would be generated by proceeding top-down using only those structures. Suppose now that in a given design process an instance of the following basic situation is found (figure 5).

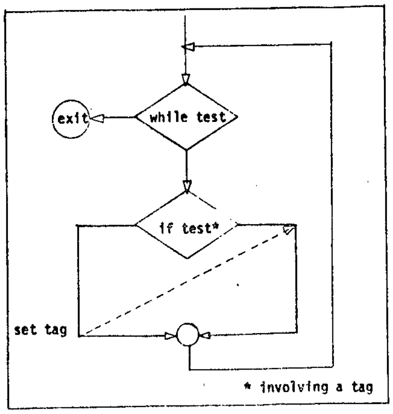


Figure 5. Dominant Tagging Effect

We called this situation a dominant tagging effect. In this case we are simulating a goto. All the undesirable aspect of gotos can be present when dominant tags are used. The designer has to decide whether to review the developing structure based on the size of the substructure that involves the tag and the eventual burden that it is adding to the program's testability.

When the complete hierarchical structure of a system is ready and informally proven correct, another dimension of well-formed programs has to be considered: the dimension of modularity. Not necessarily the partitions induced by the hierarchical structures (sets of nodes with the same degree) constitute themselves a good modularization. The following objections show in practice when attempting to use partitions directly as modules. When coding progresses bottom-up (just one level of language: implementation level), with the testing of the individual partitions being done in the same order, some very hard problems in debugging can appear. They are of the following nature: which level should be held responsible for a detected error [7]? In a top-down testing procedure, semantic context independence has to be enforced by the creation of program stubs [8] which besides being themselves additional sources of error may create some distortions to the system. The distortions appear when we try during the design phase to anticipate some decisions about input. This is done with the objective of producing better testing conditions at the early partitions of the hierarchy.

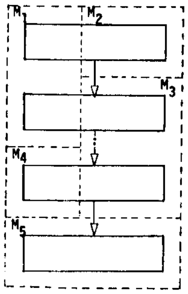


Figure 6. Module Configuration

We found that the process of defining modules can be effectively based on the systems hierarchical structure. The segmentation process that yields the modules may take place horizontally, vertically or both. The heuristics of the configuration is aimed at providing the software system with the characteristics of adaptability and changeability. This is achieved by defining modules which possess syntactic non-interference, semantic context independence, and data generality. The data structure interface between modules is the most difficult problem to be solved. We might find that if the modules do not form a class of equivalence under independent implementation the designer may be forced to provide more than one module to be implemented by the same programmer (programmers group). See [3] for a proposed solution. Two other activities take place during module configuration; identification of parallel processes and structure reconfiguration. By structure reconfiguration we mean the process of eventually transforming the tree of the system into a directed acyclic graph. This new structure proves to have the same advantages for testability as the preceeding one and might reduce considerably some duplications of work. (figure 3 is a directed acyclic graph).

2.2 Language for Design Specification

In the discussions above we left undefined the notation used to express the design. We think that this is a very critical point for the quality of the software product being designed. The proof of correctness type of approach or any other type of testing will try to match a given program with the documentation about the program [9]. If the documentation is incomplete or ambiguous, any effort towards program validation is useless. In terms of proof of correctness it would be said that assertions cannot be generated. Of course, this concern grows considerably as the complexity of the system increases.

It is not difficult to estimate the amount of extra design information that such a design as that illustrated in figure 7 would require (example from [8]).

```
Level (0)
        f = "Add member to library"
          f expands to: g then h
Level (1)
        g = "Update library index"
        h = "Add member to library text"
          g expands to: if p then i else j etc.
```

Figure 7. Need for Specification Language

We considered two notations for the present experiment. One proposed by Parnas [10] and another proposed by Hoare [5]. Parnas notation seems to apply better when the modules are already known and implementation decisions can begin. Hoare's notation seems more adequate to express the design decisions from an early stage up to, excluding, the implementation phase. We have not considered their combination although this is an interesting possibility.

The notation adopted in our experiment provided a very efficient communication mechanism between the authors. From the implementation point of view, with the use of the documentation provided by the specification text, the task of programming could approximately be described as the embedding of statements between "assertions" about the program. Together with each module specification the designer is able to provide a set of input data (in terms of the specification language data structures) that exhaustively tests the design. Those test cases can be adapted and used at the implementation level.

The acceptance of the implementation took place when a systematic inspection of the specification by the programmer was accomplished. Although it was applied manually we have hopes that this procedure will be even more effective when applied semi-automatically. The implementation of different modules could conceivably be carried out in different programming languages.

Experience with the language has indicated a number of possible improvements. Some have to do with enforcing some data protection mechanisms that the programmer will follow in the implementation. Others have to do with both language features and coding techniques at the specification level. We would like the programmer to receive from the specification phase, all the assertions that would be required for a proof of correctness in the sense of [9] to be carried out. Even if he is not proving that the implementation is correct, it would considerably strengthen the method for acceptance of implementation mentioned before.

3. CONCLUSIONS

The questions of completness and consistency of design were only dealt with informally in the exercise. Testability, modularity and informal proof of control correctness were the criteria used for design acceptance. We felt that a more formal interconnection can be established between the textual form of the design in the specification language and their respective graph models of computation. We lacked a tool that would take the form of a "data dependency table" and that could also be formally related to the two previously mentioned mechanisms. If such formal interconnection is achieved some very interesting semi-automatic procedures can be developed to deal with the issues of completeness and consistency of design. That, plus a partial mechanization of the implementation validation would give us a hope that more generally applicable methodologies could be produced.

We were strongly tempted to use analogues of the absolute modules referred to in section 1 to solve the problem of providing modules with data generality. For this purpose we sketched with considerable loss in efficiency some software simulators of standard interfaces that would have solved the problem. This idea together with the previous ones could serve as the basis for the design of a convenient programming environment for program development.

BIBLIOGRAPHY

1. Parnas, D. L. "Some Conclusions From an Experiment in Software Engineering Techniques," Department of Computer Science, Carnegie-Melon University, June 1972.
2. Henderson, P. and Snowdon, R. "An Experiment in Structured Programming," BIT 12, pp. 38-53, 1972.
3. Lucena, C. J. "On the Synthesis on Programs and Reliability," Seventh Asilomar Conference on Circuits, Systems, and Computers, November 1973.
4. Dennis, J. B. "Modularity," Advanced Course in Software Engineering, Springer-Verlag, 1973.
5. Hoare, C. "Notes on Data Structures," Structured Programming, Academic Press, 1972.
6. Goos, G. "Hierarchies," Advanced Course in Software Engineering, Springer-Verlag, 1973.
7. Parnas, D. L. "Response to Detected Errors in Well-Structured Programs," Department of Computer Science, Carnegie-Melon University, July 1972.
8. Mills, H. "Top Down Programming in Large Systems," Current Computer Science Symposium 1: Debugging Techniques in Large Systems, Prentice-Hall, 1971.
9. Igarashi, S. London, R., Luckham, D. "Automatic Program Verification 1: Logical Basis and Its Implementation," USC Information Sciences Institute, Report |S|/RR-73-1|, May 1973.
10. Parnas, D. L. "Software Module Specification With Examples," CACM, May 1972.