Integrated program measurement and documentation tools

A. Schroeder

IRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 227

INTEGRATED PROGRAM MEASUREMENT AND DOCUMENTATION TOOLS

Anne SCHROEDER

Juillet 1983

INTEGRATED PROGRAM MEASUREMENT AND DOCUMENTATION TOOLS

Anne SCHROEDER

Abstract:

While a general interest in Software Metrics arises in the literature, one may wonder about several basic and related points: what to measure on software products, how and what for ? The problem can also be stated as follows: considering a program, how to choose measurable quantities that are relevant to answer a given question, and how to present the collected measurements in such a way that makes them easily usable; also, relevant metrics can be of two quite different natures: either static, i.e. depending only on the way the program is written, or dynamic, i.e. collected at run-time, and thus varying from one execution to another and dependent of a given input.

This paper describes an attempt to integrate the collection and the efficient utilisation of measurements in the development and the use of programs. The work presented consists in three parts :

- the design of both static and dynamic measurement tools,
- examples of data processing on measurements collected on a sample of Pascal programs,
- the design of a quantitative documentation of a program, which is automatically built as measurements are collected.

The first and third steps have been developed inside an existing programming environment, *Mentor,* and we shall discuss the advantages we found in integrating the tools in such an environment.

Résumé :

Tandis qu'un intérêt croissant se manifeste pour les problèmes de mesurabilité du logiciel dans la littérature, on est amené à se poser quelques questions élémentaires : quelles quantités faut-il mesurer, comment et pourquoi ? Interrogation qui peut aussi se formuler de la façon suivante : un programme étant donné, comment choisir les variables significatives à observer pour répondre à un problème donné, et comment rendre facilement utilisable les résultats collectés; en outre, les mesures envisageables peuvent être soit statiques, i.e. ne dépendant que du texte du programme, soit dynamiques, i.e. recueillies à l'exécution et donc dépendant d'un jeu de données.

Ce papier décrit une approche à l'intégration du recueil et de l'utilisation efficace de mesures au cours du développement et de l'utilisation des programmes. Le travail exposé comporte trois parties :

- la conception d'outils de mesures statiques et dynamiques,
- des exemples de traitements effectués sur un échantillon de mesures recueillies sur des programmes Pascal,
- la conception d'un documentation quantitative des programmes, construite automatiquement au fur-et-à-mesure de la collecte des mesures.

La première et la troisième parties ont été développées dans un environnement de programmation existant, *Mentor,* et nous décrivons les avantages que nous avons trouvés à l'intégration de nos outils dans un tel environnement.

INTEGRATED PROGRAM MEASUREMENT AND DOCUMENTATION TOOLS

Anne SCHROEDER

INRIA - B.P. 105
78153 Le Chesnay (France)

Introduction

While a general interest in Software Metrics arises in the literature [Cook82, Curtis80, Friedman81, Perlis81], one may wonder about several basic and related points: what to measure on software products, how and what for ? Answers to these questions may be classified into different types: some people are concerned with the productivity of software development groups, and what they are interested in measuring are large programs and global features of these programs that can be correlated with either an error risk or a maintenance cost [Curtis79, Dunsmore79b, Potier82, Schneidewind79]. Others are programmers wanting a help in developing, debugging and optimising their programs [Basili82, Cohen77, Satterthwaite72]; these are interested in measuring any program fragment from one statement to a whole program and the information they expect from such an analysis mostly concerns run-time behaviour [Ingalls72, Knuth71]. Also, in any programming team where programs have to be used and modified by people who have not written them, there is a need for precise documentation and for any tool helping the transfer and maintenance of programs [Harrison82, Yau80]. Eventually, designers and implementers of programming languages and tools are (or should be) interested in getting statistical information about the actual way people use similar languages or systems [Hoaglin82, McDaniel82, Shimasaki80, Sweet82, Wiecek82].

All these cases are more or less various approaches to a unique problem which can be stated as follows: considering a program fragment, how to choose measurable quantities that are relevant to answer a given question, and how to present the collected measurements in such a way that makes them easily usable; also, relevant metrics can be of two quite different natures: either static, i.e. depending only on the way the program is written, or dynamic, i.e. collected at run-time, and thus varying from one execution to another and dependent of a given input.

Some of the questions above find particular answers in this paper, which describes a modest attempt to integrate the collection and the efficient utilisation of measurements in the development and the use of programs. It is indeed our personal belief that collecting measurements can be a most useful step in any software development effort, as soon as one knows exactly what to measure and how to process collected measurements; this processing includes the simplest display of rough measured quantities to the most sophisticated statistical analyses.

The work presented here consists in three parts :
- the design of both static and dynamic measurement tools,
- examples of data processing on measurements collected on a sample of Pascal programs,
- the design of a quantitative documentation of a program, which is automatically built as measurements are collected.

The first and third steps have been developed inside an existing programming environment, *Mentor* [Donzeau75, 80, 83], and we shall discuss in the corresponding sections 1.1 and 3 the advantages we found in integrating the tools in such an environment.

1 – Measurement tools design

1.1 - Integration of measurement tools in a general programming environment

Our tools have been developed as a specialised sub-environment in the general program manipulation system Mentor. During a Mentor session, a program is represented as an operator-operand tree, generally called *abstract syntax tree.* The user interactively operates on such trees through a specialised tree manipulation language *Mentol,* which also supports the definition of procedures. Mentol primitives allow three kinds of actions: basic navigation in the tree; tree pattern matching and instantiation; deletion, permutation, insertion and substitution of any subtree. The measurement tools form a library of Mentol procedures which is invoked from Mentor (later on, MML will stand for Mentol Measurement Library).

As told above, our tools gather both static and dynamic measurements; their implementation is shown on Figure 1.

Static measurements are collected by some procedures of the MML, which directly work on the Mentor representation of the program. For instance, there is a procedure that collects all the information concerning the abstract syntax tree of the program: its size, its depth (or longest path), the counts of terminal and non-terminal nodes, which correspond to the counts of operands and operators used in the program (with a slight variation for CALL statements), the counts of their distinct occurrences, and their distributions; other procedures give static characteristics that depend on the considered language. We shall give in the next section 1.2 the choice of collected variables we have made for an experiment on Pascal programs.

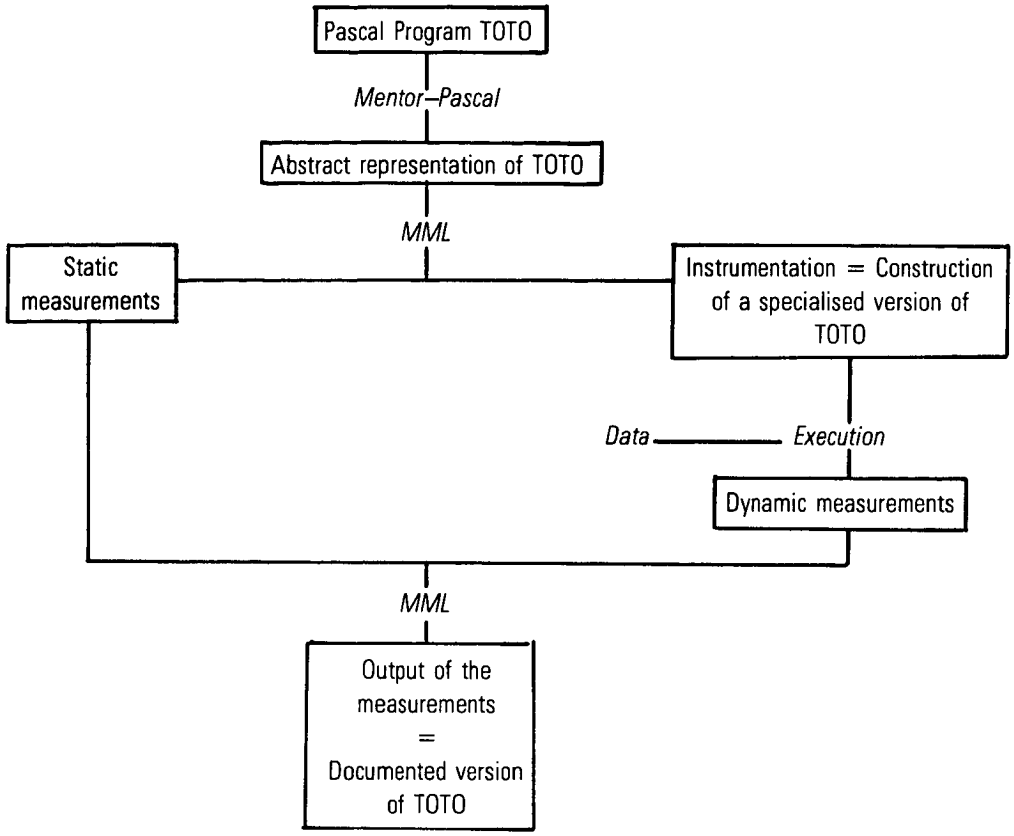

Figure 1

The dynamic measurements we considered are *selective traces*. By this, we mean they are traces of some specified events in the program, or the number of times these specified events actually occurred in a given run.

The collection of such dynamic measurements follows a preliminary step of *instrumentation* of the program. This step is performed by a procedure in the MML which inserts in the program those statements that are needed to collect the required measurements at run-time; they include declarations and initialisations of arrays which are used as counters, the incrementation of these counters and their filing when the execution is over.

As an example, Figure 2 shows a toy-program TOTO and its version instrumented to collect traces of loop iterations.

Initial program

```
program TOTO(OUTPUT);
   var I,J,IMAX:INTEGER;
       FICH:TEXT;
   begin
   FCONNECT(FICH,'vfile dontoto');
   RESET(FICH);
   READ(FICH,IMAX);

   for I:=1 to IMAX do
      begin
      WRITELN('ouh-ouh!');
      if(I<2)or(I>3) then
         for J:=1 to 2 do WRITELN('Youp-la...!...')
      end;
   I:=1;
   while I<=IMAX do
      begin
      WRITE('ooooooouh...');
      J:=0;
      repeat
         WRITELN('whaouh!');
         J:=J+1
      until J=2;
      I:=I+1
      end;
   FCLOSE(FICH)
   end.
```

Instrumented version of the program

```
program TOTO(OUTPUT,INIBOUCLE,OUTBOUCLE,BOUCLE)
   var I,J,IMAX:INTEGER;
       TOTOOUT:TEXT (*??*);
       FICH:TEXT;
       BOUCLE:array[1..200,1..2]of INTEGER (*??*);
   procedure INIBOUCLE;external (*??*);
   procedure  OUTBOUCLE(N:INTEGER;var  NF:TEXT);ex-
ternal (*??*);
   begin
   INIBOUCLE (*??*);
   FCONNECT(TOTOOUT,'vfile  TOTOOUT.mentol') (*??*);
   REWRITE(TOTOOUT) (*??*);
   WRITELN(TOTOOUT) (*??*);
   FCONNECT(FICH,'vfile  dontoto');
   RESET(FICH);
   READ(FICH,IMAX);
   for I:=1 to IMAX do
      begin
      BOUCLE[1,1]:=BOUCLE[1,1]+1 (*??*);
      WRITELN('ouh-ouh!');
      if(I<2)or(I>3) then
         for J:=1 to 2 do
            begin
            BOUCLE[2,1]:=BOUCLE[2,1]+1 (*??*);
            WRITELN('Youp-la...!...')
            end
      end;
   I:=1;
   while I<=IMAX do
      begin
      BOUCLE[3,1]:=BOUCLE[3,1]+1 (*??*);
      WRITE('ooooooouh...');
      J:=0;
      repeat
         BOUCLE[4,1]:=BOUCLE[4,1]+1 (*??*);
         WRITELN('whaouh!');
         J:=J+1
      until J=2;
      I:=I+1
      end;
   FCLOSE(FICH);
   OUTBOUCLE(4,TOTOOUT) (*??*);
   WRITELN(TOTOOUT,'.xin') (*??*);
   RESET(TOTOOUT) (*??*);
   FCLOSE(TOTOOUT) (*??*)
   end.
```

Figure 2

It can be seen on Figure 2 that all instrumenting statements are followed by a particular comment ((*??*)), and that the size of the program has been much increased by the instrumentation; it should be noted, though, that there are two kinds of added statements: only those written in italics (incrementation of the counters) actually vary according to the length of the program, and to the type and frequence of the traced event; the others (declarations, initialisations and output) are independent of the size of the initial program.

The advantage of developing those tools in the frame of a programming environment such as Mentor is threefold:

- the programming facilities offered by Mentol (the command language of Mentor) and the existence of predefined patterns corresponding to all syntactic constructions of the studied language make the tasks of counting given patterns (possibly partly instantiated) or of inserting counters in given places very easy;

- also, the existence of the tools (or of the MML) enriches the programming environment itself by providing quantitative data that may be useful in debugging, optimisation and maintenance;

- Mentor actually is a multi-language system [Mélèse82, Kahn83]. A meta-description of a language (BNF type) being given, it provides a syntax-directed editor and a programming environment for this language; the language Mentol that works directly on the abstract representation of programs is of course common to all languages introduced in Mentor; the language-dependent part of a given environment consists in the predefined patterns evoked above and in a specific library of Mentol procedures. Our tools have first been developed in the Pascal dedicated version of the multi-language system, Mentor-Pascal [Mélèse81], which explains why all examples deal with Pascal programs; however, they are implemented in such a way that they may easily be used in any other Mentor environment, which means that some of language-dependent and measurement-dependent Mentol procedures can automatically be built.

The closest approach to ours we found, concerning measurement tools design, is that of De Prycker's [De Prycker82]. His "aim is to develop a system that would analyse *any* high level language, measure *all* usual statistics (static or dynamic), and that could be extended very easily to obtain still other programs statistics". We may say that our own tools have more or less achieved the same purpose. Both approaches take advantage of an existing system able to do the lexical analysis and the parsing of a program in some language, provided a description of the language is given. In De Pryckers's case, this system is composed of a lexical analyser and a parser, which could constitute the front end of a meta-compiler; in our case, it is a whole syntax-oriented program editor system (Mentor), in which one can use a specific program manipulation language to develop the tools for chosen statistics. The former approach may be (this is just an intuition, no tests...) a little more efficient because of its *ad hoc* nature, but the second one offers the user a complete environment for the meta-described language.

There exist a number of commercial and non-commercial program analysers which provide classical measurements (operators/operands counts, cyclomatic complexity, procedure call graphs) for

usual high-level languages like Fortran, Pascal and Cobol; we deliberately choose not to enumerate them here, since users can find them in specialised software directories; they rarely support any dynamic measurement.

Also, though it is not their first purpose, some systems provide, on the side, statistics on program behaviour; it is obviously the case in debugging systems [Satterthwaite72, for instance], or in more general programming tools, such as: Scope [Masinter80] which interactively builds and updates a data base concerning the data flow, cross references and organisation of a program at the same time it is developed; or, the Builder System [Brown81] which also gathers information during program building; or also, the system described in [Cohen77] which provides a data base of run-time information and a special language to inquire about the dynamic behaviour of a program. All these systems integrate data collection in a more general purpose system; in our approach, we try to integrate not only the collection of measurements, but also their processing in a multi-language programming environment.

1.2 - Metrics implemented for Pascal programs

As noted above, experiments applied until now to Pascal programs. In this section, we present the metrics for which collecting tools have been implemented; they should be considered as examples of what kind of information is easy to get rather than as *the* list of existing measurement tools.

First of all, all statistics concerning the abstract syntax tree, which are language independent, are collected by a procedure of the MML. As noted in 1.1, it gives, among others, the counts of total and distinct occurrences of operands and operators. These measurements are those needed to compute derived characteristics defined in the *Software Science* approach [Halstead77]; this approach to quantify software complexity and quality gave rise to an abundant and contradictory literature [Baker79, Curtis79, Dunsmore79a, Elshoff78, Fitzsimmons78, Gordon79a, Gordon79b, Johnston80, Kavi82, Lassez81, Shen81, Shen83, Smith81, Woodfield81, among many others]. We shall, in section 2.1, present some results on such measurements. When traversing the syntax tree, its depth also is computed, and so is the (static) distribution of the operators of the language, which contains the distribution of the statements; statistics on how programmers actually use the operators of a language constitute important information for implementers, and some articles report results on static statement distribution [Brookes82, De Prycker82, Perrott81, Robinson76, Shimasaki80].

Another procedure gives characteristics of the control structure of the program: mean and maximum values of nesting depths of control statements (i.e. statements that induce some branching, such as loops, tests and goto's), and cyclomatic number of the program (number of independent cycles in the control graph) [McCabe76]; such measurements, and others close to them, are often used to estimate the structural complexity of a program [Baker80, Comer81, Hansen78, Harrison81a, Harrison81b, Laurent81, Myers77, Piwowarski82, Woodward79, Zolnowski77].

Another MML procedure gathers statistics on variable referencing; for each local or inherited variable in the measured program, it gives the exact number of references which are (statically) made to it.

Also, standard procedures in the Pascal environment [Mélèse81] collect information about the structure (static) and the actual profile (dynamic) of the call graph of a program.

As we told in section 1.1, we dynamically measure selective traces. This choice results from our own programmer's experience in facing enormous listings of exhaustive execution profiles for actually using a few lines of information. Without denying the importance of the introduction of such profiling

techniques [Ingalls72, Fitch77], which Knuth's historical paper [Knuth71] on Fortran programs initiated, we think that, most of the time, the information users really need concerns the dynamic behaviour of some specific event in their program; for instance, tracing loop iterations would be sufficient in an optimising purpose, while tracing either the paths selected after conditional statements or the statements that may have changed the value of a given variable seems a more appropriate approach in a debugging purpose [Charlton83]. Several alternative approaches to a selective profiling may also be found in [Hennell76, Cabrera82].

Also the dynamic distribution of the operators of the language is as important for implementers to know than the static one, or even more; its computation corresponds to tracing actual executions of all the operators [De Prycker82, Torsun81]. Operators distribution behaviour can also be modelled [Zweben77], and such models, of course, need to be validated.

This day, tools are implemented to provide profiles, or traces, of the following events:

- calls of procedures and functions (see the example in Section 2.2);
- iterations of loops (FOR, WHILE and REPEAT);
- executed paths in conditional statements (in an IF statement, counters give the total number of executions of the statement and the number of executions of the THEN clause; in a CASE statement, all alternative paths are counted);
- heads of blocks (in the Pascal sense, i.e. bodies of programs, procedures or functions);
- all statements;
- dynamic use of the operators.

In all cases, an event is defined as a predefined pattern of the environment, possibly fully or partly instantiated; thus, the existing selective profiling tools are easy to adapt to any other event which can be defined in those terms.

2 – Processing measurements

2.1 - Static measurements: measuring the complexity of a program

In an attempt to better understand the relationships between static complexity metrics, we performed statistical analyses on such measurements collected on a sample of 18 Pascal programs including 921 Pascal blocks (program, procedure or function). A thorough statistical analysis of these static measurements can be found in [Schroeder83a]. In this section, we shall briefly present a result among others as an example of a data analysis approach to the problem of comparing complexity metrics.

The variables considered for each Pascal block are given, together with their basic characteristics, in Table 1, and their correlations in Table 2.

Variables		Mean	Std. Dev.	Minimum	Maximum
Number of statements	(Nst)	13.2	19.1	0	190
Size of the syntax tree	(TT)	77.7	130.4	2	1337
Vocabulary	(Voc)	26.1	21.4	2	192
Count of distinct operands	(Ond)	14.4	16.8	1	169
Count of distinct operators	(Otd)	11.7	6.7	1	56
Cyclomatic number	(cmd)	4.4	6.3	1	60
Depth of syntax tree	(DpT)	7.1	3.4	1	25
Maximum nesting depth	(Mxd)	1.5	1.5	0	9
Mean nesting depth	(Mnd)	1.1	0.9	0	5.9

Table 1

	Nst	TT	Voc	Ond	Otd	cmd	DpT	Mxd	Mnd
Nst	1.00								
TT	0.95	1.00							
Voc	0.87	0.84	1.00						
Ond	0.84	0.84	0.97	1.00					
Otd	0.68	0.59	0.77	0.58	1.00				
cmd	0.77	0.74	0.81	0.75	0.69	1.00			
Dpt	0.56	0.50	0.60	0.45	0.80	0.60	1.00		
Mxd	0.55	0.46	0.57	0.42	0.76	0.62	0.83	1.00	
Mnd	0.52	0.44	0.56	0.42	0.76	0.59	0.82	0.97	1.00

Correlation Matrix
Table 2

One may note the strong correlation relating the size TT of a block with most of the other variables; it is even stronger than that relating the size expressed as a number of statements to the other variables; actually, the size of a block (expressed as the total number of operators and operands required) considers at the same time the number of statements to execute and the complexity due to expressions and argument lists, and can thus be considered as a more precise notion of the global block size.

Global relationships between the variables are analysed with the help of a multidimensional data analysis method, called *Correspondence Analysis* [Benzecri73, Hill74, Schroeder78]. Very shortly speaking, this method gives a good low-dimension representation of a set of points in a high-dimension space. In our case, the set of 921 Pascal blocks on which 9 variables have been measured can be considered as a set of 921 points in a 9-dimension space, and in the same time, the set of measurements can be considered as a set of 9 points in a 921-dimension space, the 921x9 data array giving the coordinates of both sets in the two spaces. The approximate representation provided by Correspondence Analysis is

the projection of the considered set of points on its maximum inertia subspace, given distance measures in the 921- and 9-dimension spaces. The choice of this distance allows a simultaneous representation of the two sets put into correspondence through the given data; in such a representation, two measurement points are close to each other if their distributions on the set of blocks are close (details may be found in any of the three above references).

The three-dimension approximate representation of our data is given on Figure 3; it consists in two projections on the planes respectively generated by the (1st and 2nd) and (2nd and 3rd) inertia axes of the set. The respective parts of the initial dispersion that are kept through projection on each axis are also given on Figure 3, indicating that the 1st axis carries 67.5%, the first plane (1st and 2nd axes) 80.8%, and the set of the two planes together 88.8% of the initial dispersion.

Observing this approximate representation, the following comments can be made:

- the main trend in the data (observed on the 1st preponderant axis) is that due to the size of the blocks, since 67.5% of the initial information consist in ranking the blocks according to their sizes, expressed either as the number of statements or as the size of the syntax tree;

- then, on the second plane (2nd and 3rd axes), on which the size variables (Nst and TT) are close to the centre, one may examine the respective positions of the other variables, independently of the global influence of the size. At first, the outlying position of the two variables concerning the nesting depth of control statements (Mnd and Mxd) indicates that these variables carry some proper information compared with the others; this influence also seems to operate a clustering among the set of points representing the blocks (dots on the figure) between linearly written blocks and those in which the nesting depth is greater than one. Among the other variables, one can distinguish three different natures of structural complexity: the one that arises from a large number of control statements (cmd), which also is the closest to the nesting depths variables; the one which is due to a deep syntax tree (DpT), which means a high general nesting depth including the nesting of statements, that of expressions and the use of lists; and the one that comes from using a large number of distincts operands, which certainly makes a program more complex though it can involve no statement nor expression nesting at all. Further work has been done to characterise these different types of complexity through their correlations with some external variable (understanding by a reader, error risk, programmer's style, for instance); part of it may be found in [Schroeder83b], together with other static measurement analyses.

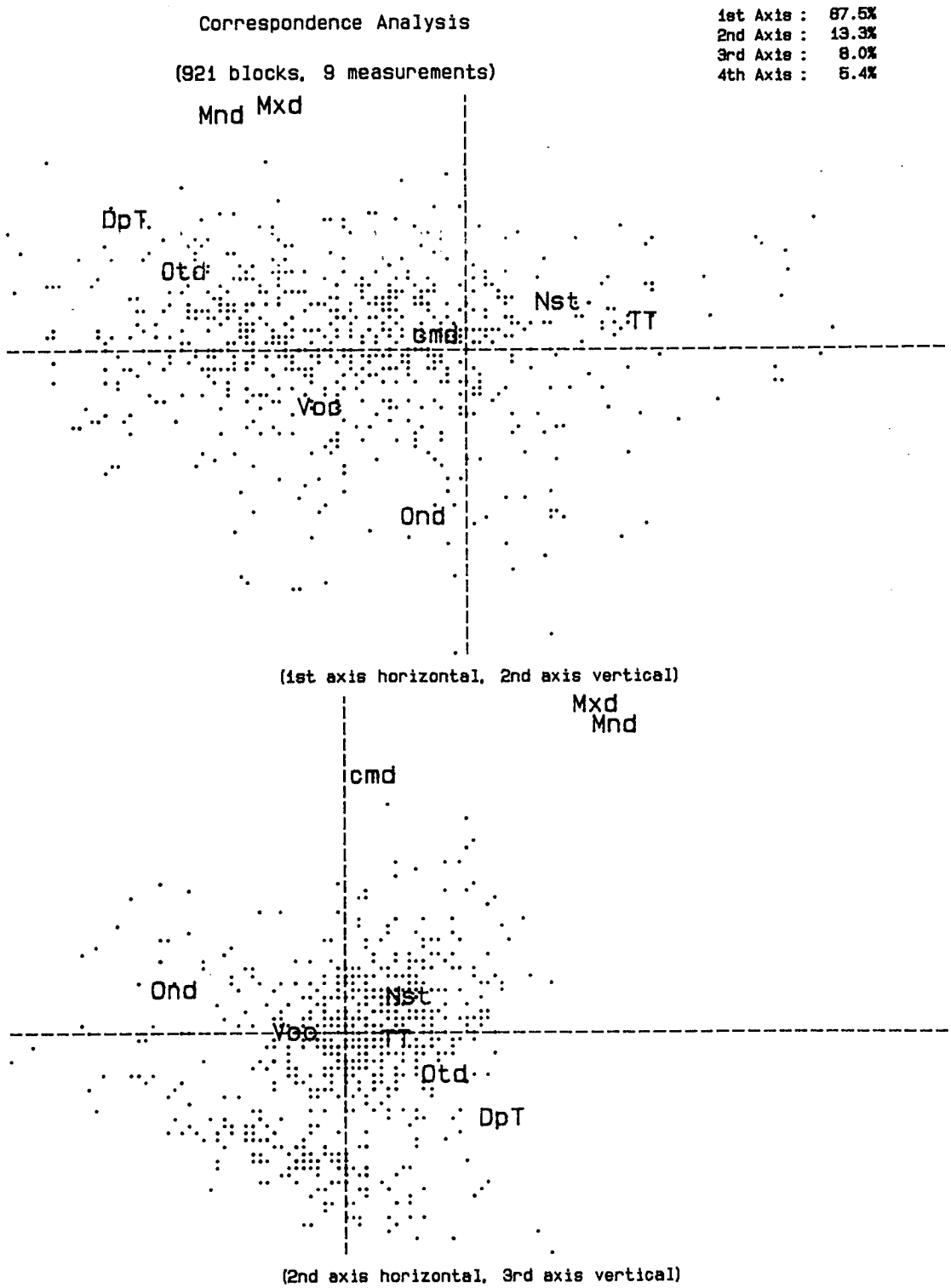
Correspondence Analysis
(921 blocks, 9 measurements)
1st Axis : 87.5%
2nd Axis : 13.3%
3rd Axis : 8.0%
4th Axis : 5.4%
Mnd
Mxd
DpT
Otd
Nst
TT
cmd
Voo
Ond
(1st axis horizontal, 2nd axis vertical)
Mxd
Mnd
cmd
Ond
Nst
Voo
TT
Otd
DpT
(2nd axis horizontal, 3rd axis vertical)

Figure 3

2.2 - Dynamic measurements

We have not done much work yet on the statistical processing of dynamic measurements. Let us indicate an analysis of both static and dynamic Pascal statement distributions, measured on our own sample and compared with some published results [Schroeder83b].

Also, the simplest processing made on selective traces consists in cumulating traces issued from several different executions. An example of such a cumulated profile is given in Figure 4; it shows the same toy-program as in Figure 2, in which the Call statements profile have been collected during two executions.

```
program TOTO(OUTPUT);
     var  I,J,IMAX:INTEGER;
          FICH:TEXT;
 (*2*)
     begin
      (*2*)
     FCONNECT(FICH,'vfile dontoto');
      (*2*)
     RESET(FICH);
      (*2*)
     READ(FICH,IMAX);
     for I:=1 to IMAX do
          begin
           (*22*)
          WRITELN('ouh-ouh!');
          if(I<2)or(I>3) then
               for J:=1 to 2 do (*36*)
               WRITELN('Youp-la...!...')
          end;
     I:=1;
     while I<=IMAX do
          begin
           (*22*)
          WRITE('ooooooouh...');
          J:=0;
          repeat
                (*44*)
               WRITELN('whaouh!');
               J:=J+1
          until J=2;
          I:=I+1
          end;
      (*2*)
     FCLOSE(FICH)
     end (*
     begin
     FCLOSE:=2;
     WRITE:=12;
     WRITELN:=52;
     READ:=52;
     RESET:=12;
     FCONNECT:=2
     end*).
```

Figure 4

The number of observed executions labels the head of the current block, and every Call statement in the program is labelled by the number of times it has actually been executed; at the end of the program, a summary table recapitulates the total number of executions of each distinct procedure or function.

3 – Creating a quantitative documentation of a program

3.1 - The actual state of the tools: structured comments

The observation of Figure 4 in the previous section is the best introduction to the current paragraph; it shows how collected dynamic measurements are displayed. Let us give some more detail: once a program is instrumented, its execution automatically builds a file containing the required counters; then, inside a Mentor session, this file can be loaded together with the initial program (non instrumented, of course) and it is a profiling procedure of the MML (see Figure 1) which contructs a *profiled* version of the program such as that on Figure 4, in which the contents of the counters appear as Pascal comments.

For static metrics, the same MML procedure which collects measurements also creates a static quantitative documentation of the program; this display of information consists, once more, in a Pascal comment generation. Figure 5 shows the static documentation of the toy-program TOTO, which is nothing else than TOTO itself, annotated by specialised comments.

```
(*************************************************************************
 *
 *   DOCUMENTATION STATIQUE
 *
 *                          - Utilisateur : SCHROEDER
 *                          - Date        : 06/15/83  17:08
 *
 *************************************************************************
 **)
program TOTO(OUTPUT)
 (**
   *  NOMBRE d'INSTRUCTIONS executables : 17
   *
   *  TAILLE (comptee en nombre d'operandes et operateurs utilises) : 81
   *         - Nb. total d'operandes      :          35
   *         - Nb. d'operandes distincts  :          14
   *
   *         - Nb. total d'operateurs     :          46
   *         - Nb. d'operateurs distincts :          21
   *
   *
   *  PROFONDEUR de l'arbre syntaxique du code executable : 7
   *
   *  STATISTIQUES portant sur les REFERENCES aux VARIABLES :
   *     Nb. de variables locales utilisees      : 4
   *     Nb. de references aux variables locales : 19
   *     I : 7
   *     J : 5
   *  IMAX : 3
   *  FICH : 4
   *     Nb. total de variables utilisees      : 4
   *     Nb. total de references aux variables : 19
```

```
  *
  *  NOMBRE CYCLOMATIQUE du graphe de controle du programme :
  *     - 1ere variante : 1+ nb. de branchements (DO, WHILE, REPEAT,
  *          IF et nb. d'alternatives des CASE ) : 6
  *     - 2eme variante : idem en ponderant les IF du nb. d'operateurs
  *          logiques qu'ils contiennent          : 7
  *
  *  STATISTIQUES portant sur les NIVEAUX d'IMBRICATION des instructions
  *  de branchement :
  *     Niveau d'imbrication maximum : 3
  *     Niveau d'imbrication moyen   : 9 divise par 5
  *
  *

*************************************************************************
*);
     var  I,J,IMAX:INTEGER;
          FICH:TEXT;
     begin
     ...
     end.
```

Figure 5

3.2 - The (near) future in a multi-language environment

In Section 3.1, we have shown the actual quantitative documentation building from both static and dynamic measurements, provided by the MML this day. In our plans, this is only a first attempt, the main quality of which is its simplicity. We must, though, underline the importance of Mentor's implementation of comments in what has been done. In Mentor, comments are considered as a particular case of various possible annotations which are themselves abstract syntax trees in some formalism (or language) known by Mentor. To this day, comments could be either alphanumeric strings organised in lines, or some subtree in the current language (Pascal subtree, in our examples). This implies that computation can be made on comments: for instance, if some comment in a program is a Pascal expression (which is the case in the profiles such as that shown in Figure 4), then it can be added to some other expression, which is exactly what is done in the MML procedure which computes cumulative profiles over several executions. Also, one may have noted, in Figure 4, the particular form taken by the summary table appearing as a comment at the end of the program in order to recapitulate the numbers of executions of each distinct procedure or function; this comment obviously is a Pascal *begin ... end* composed of a list of assignment statements, the left hand parts of which are Pascal identificators containing names of procedure or function, while the right hand parts contain expressions; as in the previous example, the structure of this comment is what allows an easy profile computing and updating. The result of this implementation is that the documentation of the program is nothing else than the program itself, adorned by particular comments.

A current effort in the Mentor development group is towards a generalisation of comments to all kinds of attributes which would themselves be abstract syntax trees in their own language; then, the multi-language aspect of Mentor could allow to handle together annotations of different natures [Donzeau83]. In our own application to integrated measurement tools design, we plan to define different formalisms that build documents of different natures to display numerical results: simple

numerical tables, or graphical material. Such material could consist in scatter plots, histograms or Correspondence analysis plots (as in Figure 3), which would be results of analyses made on samples of programs (for static measurements) or of executions (for dynamic measurements). Thus, the static documentation attached to a program could contain not only statistic information about the program itself, but also about other programs already measured in order to locate it in a global comparison. In order to do that, we plan to design a language specialised in describing the statistical information attached to a program, say *DOP (DOcumentation of Programs)* which will be one more formalism introduced in Mentor. The *documented* program will then be the initial program, annotated by Dop trees, and while a Pascal-decompilation of such an object would give a regular Pascal program with strange comments, the Dop-decompilation of the same object would provide the documentation itself. The final purpose of such a system will be to build dynamically and automatically paper, on-line and/or graphical documents containing all the information collected statically and at run-time for a given program or for a given language. This is what we call a *quantitative documentation.* The first step in such a development, that is the design and integration in Mentor of a special purpose language to display numerical information, is actually beginning; then the (quantitative) documentation of a program will be a structure in this formalism.

Concluding remarks

In this paper, we presented an attempt to integrate measurement collecting and processing in a general environment for program development. It actually seems to us that the utility of measurement collecting is directly correlated with the facility of using the information isuued from them. The existing tools include both static and dynamic measurement tools, as well as tools for automatic building of a quantitative documentation derived from the measurements. This work has been developed inside the Mentor programming environment, and we emphasised the advantages we found in building our tools in such a syntax-directed environment; this is not intended to prove that our approach is the best one, but only to underline the inherent quality of integration between different purpose tools and the flexibility it gives; this to convince programming environment designers of the importance of the system to be open to various applications.

References

[Baker79]
A. L. Baker and S. H. Zweben
The Use of Software Science in Evaluating Modularity Concepts
IEEE Transactions on Software Engineering, Vol. SE–5, No. 2, March 1979. pp. 110–120

[Baker80]
A. L. Baker and S. H. Zweben
A Comparison of Control Flow Complexity
IEEE Transactions on Software Engineering, Vol. SE–6, No. 6, November 1980. pp. 506–512

[Basili82]
V. R. Basili and H. D. Mills
Understanding and Documenting Programs
IEEE Transactions on Software Engineering, Vol. SE–8, No. 3, May 1982. pp. 270–283

[Benzecri73]
J.–P. Benzecri et Coll.
L'Analyse des Données
Dunod, Paris , 1973.

[Brookes82]
G. R. Brookes, I. R. Wilson and A. M. Addyman
A Static Analysis of Pascal Program Structures
Software–Practice and Experience, Vol. 12, 1982. pp. 959–963

[Brown81]
P. J. Brown
Dynamic Program Building
Software–Practice and Experience, Vol. 11, 1981. pp. 831–843

[Cabrera82]
L. P. Cabrera
Some Elements of a Programming Environment with Efficient Run-Time Performance Analysis of Programs
6th International Conference on Software Engineering, Poster Session, Tokyo, 1982. pp. 11–12

[Charlton83]
C. C. Charlton and P. H. Leng
Aids for Pragmatic Error Detection
Software–Practice and Experience, Vol. 13, 1983. pp. 59–66

[Cohen77]
J. Cohen and N. Carpenter
A Language for Inquiring about the Run-time Behaviour of Programs
Software–Practice and Experience, Vol. 7, 1977. pp. 445–460

[Comer81]
J. R. Comer, J. R. Rinewalt and M. M. Tanik
A Comparison of Two Different Program Complexity Measures
Performance Evaluation Review , Summer 1981. pp. 26–28

[Cook82]
M. L. Cook
Software Metrics: An Introduction and Annotated Bibliography
Software Engineering Notes, ACM Sigsoft, Vol. 7, No. 2, April 1982. pp. 41–60

[Curtis79]
B. Curtis, S. B. Sheppard and P. Milliman
Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics
4th International Conference on Software Engineering, IEEE Cat. No. CH1479–5, 1979. pp. 356–360

[Curtis80]
B. Curtis
Measurement and Experimentation in Software Engineering
Proceedings of the IEEE, Vol. 68, No. 9, September 1980. pp. 1144–1157

[De Prycker82]
M. De Prycker
On the Development of a Measurement System for High Level Language Program Statistics
IEEE Transactions on Software Engineering, Vol. C–31, No. 9, September 1982. pp. 883–891

[Donzeau75]
V. Donzeau–Gouge, G. Huet, G. Kahn, B. Lang and J.–J. Lévy
A Structure Oriented Program Editor: a First Step toward Assisted Programming
International Computing Symposium, North Holland Pub. Co., 1975. pp. 113–120

[Donzeau80]
V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang
Programming Environments Based on Structured Editors: the MENTOR Experience
Rapport de Recherche INRIA No. 26, 1980.

[Donzeau83]
V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse and E. Morcos
Outline of a Tool for Document Manipulation
IFIP'83, Paris , 1983.

[Dunsmore79a]
H. E. Dunsmore and J. D. Gannon
Experimental Analysis of some Programming Effort Factors
Computer Science and Statistics: Proceedings of the 12th Symposium on the Interface, J. F. Gentleman (ed.), May 1979. pp. 279–286

[Dunsmore79b]
H. E. Dunsmore and J. D. Gannon
Data Referencing: An Empirical Investigation
Computer , December 1979. pp. 50–59

[Elshoff78]
J. L. Elshoff
A Study of the Structural Composition of PL/I Programs
SIGPLAN Notices, Vol. 13, No. 6, 1978. pp. 29–37

[Fitch77]
J. Fitch
Profiling a Large Program
Software–Practice and Experience, Vol. 7, 1977. pp. 511–518

[Fitzsimmons78]
A. Fitzsimmons and T. Love
A Review and Evaluation of Software Science
Computing Surveys, Vol. 10, No. 1, 1978. pp. 3–18

[Friedman81]
H. P. Friedman
On Measurement and Evaluation of Software: A View from the Chair
Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, W. F. Eddy (ed.), Springer–Verlag, 1981. pp. 187–191

[Gordon79a]
R. D. Gordon
Measuring Improvements in Program Quality
IEEE Transactions on Software Engineering, Vol. SE–5, No. 2, March 1979. pp. 79–90

[Gordon79b]
R. D. Gordon
A Qualitative Justification for a Measure of Program Quality
IEEE Transactions on Software Engineering, Vol. SE–5, No. 2, March 1979. pp. 121–128

[Halstead77]
M. H. Halstead
Elements of Software Science
Elsevier North–Holland Inc., N.Y., 1977.

[Hansen78]
W. J. Hansen
Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)
SIGPLAN Notices, Vol. 13, No. 3, March 1978. pp. 29–33

[Harrison81a]
W. A. Harrison and K. I. Magel
A Complexity Measure Based on Nesting Level
SIGPLAN Notices, Vol. 16, No. 3, 1981. pp. 63–74

[Harrison81b]
W. A. Harrison and K. I. Magel
A Topological Analysis of the Complexity of Computer Programs with less than Three Binary Branches
SIGPLAN Notices, Vol. 16, No. 4, 1981. pp. 51–63

[Harrison82]
W. Harrison, K. Magel, R. Kluczny and A. DeKock
Applying Software Complexity Metrics to Program Maintenance
Computer , September 1982. pp. 65–79

[Hennell76]
M. A. Hennell, M. R. Woodward and D. Hedley
On Program Analysis
Information Processing Letters, Vol. 5, No. 5, November 1976. pp. 136–140

[Hill74]
M. O. Hill
Correspondence Analysis: a neglected multivariate method
Applied Statistics, Vol. 23, No. 3, 1974. pp. 340–354

[Hoaglin82]
D. C. Hoaglin, V. C. Klema and S. C. Peters
Exploratory Data Analysis in a Study of the Performance of Nonlinear Optimization Routines
ACM Transactions on Mathematical Software, Vol. 8, No. 2, June 1982. pp. 145–162

[Ingalls72]
D. Ingalls
The Execution Time Profile as a Programming Tool
Courant Computer Science Symposium 5, in Design and Optimization of Compilers, W. Rustin (ed.), Prentice–Hall, 1972. pp. 107–128

[Johnston80]
D. B. Johnston and A. M. Lister
An Experiment in Software Science
Proceedings of the Symposium on Language Design and Programming Methodology, J. M. Tobias (ed.), Springer Verlag, 1980. pp. 195–215

[Kahn83]
G. Kahn, B. Lang, B. Mélèse and E. Morcos
Metal: a Formalism to Specify Formalisms
Science of Computer Programming (to appear), 1983.

[Kavi82]
K. M. Kavi and U. B. Jackson
Effect of Declarations on Software Metrics: An Experiment in Software Science
Performance Evaluation Review, Vol. 11,No. 2, Summer 1982. pp. 57–71

[Knuth71]
D. E. Knuth
An Empirical Study of Fortran Programs
Software–Practice and Experience, Vol. 1, 1971. pp. 105–133

[Knuth82]
D. E. Knuth
The WEB System of Structured Documentation
Stanford University Research Rept., September 1982.

[Lassez81]
J.–L. Lassez, D. van der Knijff, J. Shepherd and C. Lassez
A Critical Examination of Software Science
The Journal of Systems and Software, Vol. 2, 1981. pp. 105–112

[Laurent81]
J.–P. Laurent et J.–M. Fouet
Outillage de manipulation de programmes fondé sur une représentation arborescente
Premier Colloque de Génie Logiciel, AFCET, Paris, Juin 1981. pp. 105–118

[Masinter80]
L. M. Masinter
Global Program Analysis in an Interactive Environment
Xerox, Palo Alto Research Center, Report No. SSL–80–1, January 1980.

[McCabe76]
T. J. McCabe
A Complexity Measure
IEEE Transactions on Software Engineering, Vol. SE–2, No. 4, December 1976. pp. 308–320

[McDaniel82]
G. McDaniel
An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies
SIGPLAN Notices, Vol. 17, No. 4, April 82. pp. 167–176

[Mélèse81]
B. Mélèse
Mentor : l'environnement Pascal
Rapport Technique INRIA No. 5, Octobre 1981.

[Mélèse82]
B. Mélèse
Metal : un méta-langage pour le système Mentor
TSI (Technique et Science Informatiques –Dunod–), Vol. 1, No. 4, 1982. pp. 275–285

[Myers77]
G. J. Myers
An Extension to the Cyclomatic Measure of Program Complexity
SIGPLAN Notices, Vol. 12, No. 10, October 1977. pp. 61–64

[Perlis81]
A. J. Perlis, F. G. Sayward and M. Shaw (eds.)
Software Metrics
The MIT Press, 1981.

[Perrott81]
R. H. Perrott and P. S. Dhillon
An Experiment with Fortran and Pascal
Software–Practice and Experience, Vol. 11, 1981. pp. 491–496

[Piwowarski82]
P. Piwowarski
A Nesting Level Complexity Measure
SIGPLAN Notices, Vol. 17, No. 9, 1982. pp. 44–50

[Potier82]
D. Potier, J.-L. Albin, R. Ferréol et A. Bilodeau
Experiments with Computer Software Complexity and Reliability
6th International Conference on Software Engineering, IEEE Cat. No. 82CH794-4, September 1982. pp. 94-103

[Robinson76]
S. K. Robinson and I. S. Torsun
An Empirical Analysis of Fortran Programs
The Computer Journal, Vol. 19, No. 1, 1976. pp. 56-62

[Satterthwaite72]
E. H. Satterthwaite
Debugging Tools for High Level Languages
Software-Practice and Experience, Vol. 2, 1972. pp. 197-217

[Schneidewind79]
N. F. Schneidewind and H.-M. Hoffmann
An Experiment in Software Error Data Collection and Analysis
IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979. pp. 276-286

[Schroeder78]
A. Schroeder
How Multidimensional Data Analysis Techniques can be of Help in the Study of Computer Systems
Proceedings of the CPEUG Meeting, Boston, October 1978.

[Schroeder83a]
A. Schroeder
Analyse quantitative statique de programmes Pascal
Submitted for publication, available from the author, 1983.

[Schroeder83b]
A. Schroeder
On the Distribution of Statements in Pascal Programs
Submitted for publication, available from the author, 1983.

[Shen81]
V. Y. Shen and H. E. Dunsmore
Analyzing Cobol Programs via Software Science
Department of Computer Sciences, Tecnical Report No. 348, Purdue University, September 1981.

[Shen83]
V. Y. Shen, S. D. Conte and H. E. Dunsmore
Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support
IEEE Transactions on Software Engineering, Vol. SE–9, No. 2, March 1983. pp. 155–165

[Shimasaki80]
M. Shimasaki, S. Fukaya, K. Ikeda and T. Kiyono
An Analysis of Pascal Programs in Compiler Writing
Software–Practice and Experience, Vol. 10,, 1980. pp. 149–157

[Smith81]
C. P. Smith
The Application of Halstead's Software Science Difficulty Measure to a Set of Programming Projects
IBM Report No. TR03.125, 1981.

[Sweet82]
R. E. Sweet and J. G. Sandman Jr.
Empirical Analysis of the Mesa Instruction Set
SIGPLAN Notices, Vol. 17, No. 4, April 82. pp. 167–176

[Torsun81]
I. S. Torsun and M. M. Al–Jarrah
Dynamic Analysis of COBOL Programs
Software–Practice and Experience, Vol. 11, 1981. pp. 949–961

[Wiecek82]
C. A. Wiecek
A Case Study of VAX-11 Instruction Set Usage for Compiler Execution
SIGPLAN Notices, Vol. 17, No. 4, April 82. pp. 177–184

[Woodfield81]
S. N. Woodfield, V. Y. Shen and H. E. Dunsmore
A Study of Several Metrics for Programming Effort
The Journal of Systems and Software, Vol. 2, 1981. pp. 97–103

[Woodward79]
M. R. Woodward, M. A. Hennell and D. Hedley
A Measure of Control Flow Complexity in Program Text
IEEE Transactions on Software Engineering, Vol. SE–5, No. 1, January 1979. pp. 45–50

[Yau80]
S. S. Yau and J. S. Collofello
Some Stability Measures for Software Maintenance
IEEE Transactions on Software Engineering, Vol. SE–6, No. 6, November 1980. pp. 545–552

[Zolnowski77]
J. M. Zolnowski and D. B. Simmons
Measuring Program Complexity
Digest of Papers of Fall COMPCON77, IEEE Cat. No. 77CH1258–3C, 1977. pp. 336–340

[Zweben77]
S. H. Zweben
A Study of the Physical Structure of Algorithms
IEEE Transactions on Software Engineering, Vol. SE–3, No. 3, May 1977. pp. 250–258

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique