Some Stability Measures for Software Maintenance

STEPHEN S. YAU, FELLOW, IEEE, AND JAMES S. COLLOFELLO, MEMBER, IEEE

Abstract—Software maintenance is the dominant factor contributing to the high cost of software. In this paper, the software maintenance process and the important software quality attributes that affect the maintenance effort are discussed. One of the most important quality attributes of software maintainability is the stability of a program, which indicates the resistance to the potential ripple effect that the program would have when it is modified. Measures for estimating the stability of a program and the modules of which the program is composed are presented, and an algorithm for computing these stability measures is given. An algorithm for normalizing these measures is also given. Applications of these measures during the maintenance phase are discussed along with an example. An indirect validation of these stability measures is also given. Future research efforts involving application of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.

Index Terms—Algorithms, applications, logical stability, module stability, maintenance process, normalization, potential ripple effect, program stability, software maintenance, software quality attributes, validation.

I. Introduction

IT IS well known that the cost of large-scale software systems has become unacceptably high [1], [2]. Much of this excessive software cost can be attributed to the lack of meaningful measures of software. In fact, the definition of software quality is very vague. Since some desired attributes of a program can only be acquired at the expense of other attributes, program quality must be environment dependent. Thus, it is impossible to establish a single figure for software quality. Instead, meaningful attributes which contribute to software quality must be identified. Research results in this area have contributed to the definition of several software quality attributes, such as correctness, flexibility, portability, efficiency, reliability, integrity, testability, and maintainability [3]-[6]. These results are encouraging and provide a reasonably strong basis for the definition of the quality of software.

Since software quality is environment dependent, some attributes may be more desirable than others. One attribute which is almost always desirable except in very limited applications is the *maintainability* of the program. Software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization [7]. The cost of these software maintenance activities has been very high, and it has been estimated ranging from 40 percent [1] to 67 percent [2] of the total cost during the life cycle of large-scale software systems. This very high software maintenance cost suggests that the maintainability of a program is a very critical software quality attribute. Measures are needed to evaluate the maintainability of a program at each phase of its development. These measures must be easily calculated and subject to validation. Techniques must also be developed to restructure the software during each phase of its development in order to improve its maintainability.

In this paper, we will first discuss the software maintenance process and the software quality attributes that affect the maintenance effort. Because accommodating the ripple effect of modifications in a program is normally a large portion of the maintenance effort, especially for not well designed programs [7], we will present some measures for estimating the *stability* of a program, which is the quality attribute indicating the resistance to the potential ripple effect which a program would have when it is modified. Algorithms for computing these stability measures and for normalizing them will be given. Applications of these measures during the maintenance phase along with an example are also presented. Future research efforts involving the application of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.

II. The Maintenance Process

As previously discussed, software maintenance is a very broad activity. Once a particular maintenance objective is established, the maintenance personnel must first understand what they are to modify. They must then modify the program to satisfy the maintenance objectives. After modification, they must ensure that the modification does not affect other portions of the program. Finally, they must test the program. These activities can be accomplished in the four phases as shown in Fig. 1.

The first phase consists of analyzing the program in order to understand it. Several attributes such as the complexity of the program, the documentation, and the self-descriptiveness of the program contribute to the ease of understanding the program. The *complexity* of the program is a measure of the effort required to understand the program and is usually based on the control or data flow of the program. The *self-descriptiveness* of the program is a measure of how clear the program is, i.e., how easy it is to read, understand, and use [5].

The second phase consists of generating a particular mainte-

Manuscript received April 1, 1980; revised July 25, 1980. This work was supported by the Rome Air Development Center, U.S. Air Force System Command, under Contracts F30602-76-C0397 and F30602-80-C-0139.

S. S. Yau is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

J. S. Collofello was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the Department of Computer Science, Arizona State University, Tempe, AZ 85281.

0098-5589/80/1100-0545$00.75

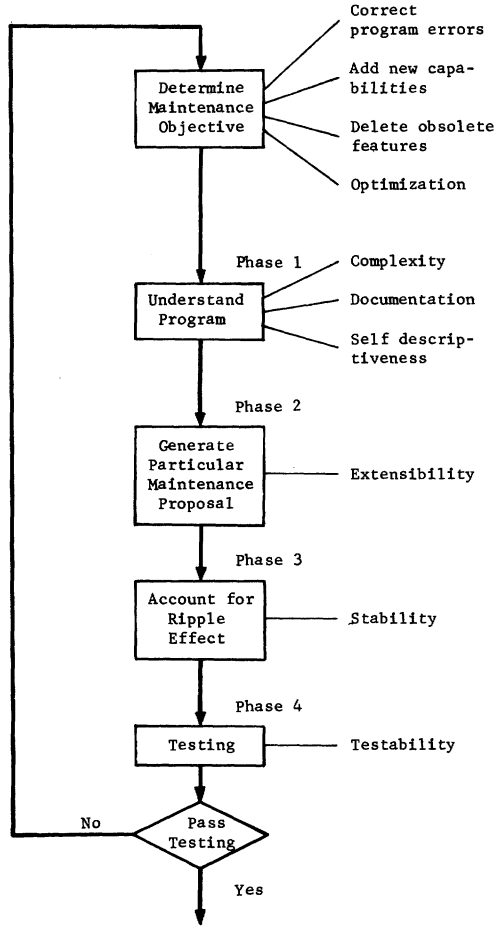


Fig. 1. The software maintenance process.

nance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified. However, the ease of generating maintenance proposals for a program is primarily affected by the attribute *extensibility*. The extensibility of the program is a measure of the extent to which the program can support extensions of critical functions [5].

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications. In software, the effect of a modification may not be local to the modification, but may also affect other portions of the program. There is a ripple effect from the location of the modification to the other parts of the programs that are affected by the modification [7]. One aspect of this ripple effect is logical or functional in nature. Another aspect of this ripple effect concerns the performance of the program. Since a large-scale program usually has both functional and performance requirements, it is necessary to understand the potential effect of a program modification from both a logical and a performance point of view [7]. The primary attribute affecting the ripple effect as a consequence of a program modification is the *stability* of the program. Program stability is defined as the resistance to the amplification of changes in the program.

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before. It is important that cost-effective testing techniques be applied during maintenance. The primary factor contributing to the development of these cost-effective techniques is the *testability* of the program. Program testability is defined as a measure of the effort required to adequately test the program according to some well defined testing criterion.

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these software quality attributes must be combined to form a maintainability measure. One of the most important quality attributes is the *stability* of the program. This fact can be illustrated by considering a program which is easy to understand, easy to generate modification proposals for, and easy to test. If the stability of the program is poor, however, the impact of any modification on the program is large. Hence, the maintenance cost will be high and the reliability may also suffer due to the introduction of possible new errors because of the extensive changes that have to be made.

Although the potential benefits of a validated program stability measure are great, very little research has been conducted in this area. Previous stability measures have been developed by Soong [3], Haney [6], and Myers [4]. There exist several weaknesses in these measures which have prevented their wide acceptance. Their largest problem has been the inability to validate the measures due to model inputs that are questionable or difficult to obtain. Other weaknesses of these measures include an assumption that all modifications to a module have the same ripple effect, a symmetry assumption that if there exists a nonzero probability of having to change a module i given that module j is changing then there exists a nonzero probability of having to change module j given that module i is changing, and a failure to incorporate a performance component as part of the stability measure.

III. Development of Logical Stability Measures

The *stability* of a program has been defined as the resistance to the potential ripple effect that the program would have when it is modified. Before considering the stability of a program, it is necessary to develop a measure for the stability of a module. The stability of a module can be defined as a measure of the resistance to the potential ripple effect of a modification of the module on other modules in the program. There are two aspects of the stability of a module: the logical aspect and the performance aspect. The *logical stability* of a module is a measure of the resistance to the impact of such a modification on other modules in the program in terms of logical considerations. The *performance stability* of a module is a measure of the resistance to the impact of such a modification on other modules in the program in terms of performance considerations. In this paper, logical stability measures will be developed for a program and the modules of which the program is composed. Performance stability measures are currently under development and the results will be reported in a subsequent paper. Both the logical and the performance stability measures are being developed to overcome the weaknesses of the previous stability measures. In addition, the stability measures are being developed with the following requirements to increase their applicability and acceptance:

1) ability to validate the measures,
2) consistency with current design methodologies,
3) utilization in comparing alternate designs, and
4) diagnostic ability.

It should be noted that the stability measures being described are not in themselves indicators of program maintainability. As previously mentioned, program stability is a significant factor contributing to program maintainability. Although the measures being described estimate program stability, they must be utilized in conjunction with the other attributes affecting program maintainability. For example, a single module program of 20 000 statements will possess an excellent program stability since there cannot be any ripple effect among modules; however, the maintainability of the program will probably be quite poor.

Development of a Module Logical Stability Measure

The logical stability of a module is a measure of the resistance to the expected impact of a modification to the module on other modules in the program in terms of logical considerations. Thus, a computation of the logical stability of a module must be based upon some type of analysis of the maintenance activity which will be performed on the module. However, due to the diverse and almost random nature of software maintenance activities, it is virtually meaningless to attempt to predict when the next maintenance activity will occur and what this activity will consist of. Thus, it is impossible to develop a stability measure based upon probabilities of what the maintenance effort will consist of. Instead, the stability measure must be based upon some subset of maintenance activity for which the impact of the modifications can readily be determined. For this purpose, a primitive subset of the maintenance activity is utilized. This consists of a change to a single variable definition in a module. This primitive subset of maintenance activity is utilized because regardless of the complexity of the maintenance activity, it basically consists of modifications to variables in the modules. A logical stability measure can then be computed based upon the impact of these primitive modifications on the program. This logical stability measure will accurately predict the impact of these primitive modifications on the program and, thus, can be utilized to compute the logical stability of the module with respect to the primitive modifications.

Due to the nature of the logical stability of a module, an analysis of the potential logical ripple effect in the program must be conducted. There are two aspects of the logical ripple effect which must be examined. One aspect concerns intramodule change propagation. This involves the flow of program changes within the module as a consequence of the modification. The other aspect concerns intermodule change propagation. This involves the flow of program changes across module boundaries as a consequence of the modification.

Intramodule change propagation is utilized to identify the set Z_{ki} of interface variables which are affected by logical ripple effect as a consequence of a modification to variable definition i in module k. This requires an identification of which variables constitute the module's interfaces and a characterization of the potential intramodule change propagation among the variables in the module. The variables that constitute the module's interfaces consist of its global variables, its output parameters and its variables utilized as input parameters to called modules. Each utilization of a variable as an input parameter to a called module is regarded as a unique interface variable. Thus, if variable x is utilized as an input parameter in two module invocations, then each occurrence of x is regarded as a unique interface variable. Each occurrence must be regarded as a separate interface variable since the complexity of affecting each occurrence of the variable as well as the probability of affecting each occurrence may differ.

Once an interface variable is affected, the flow of program changes may cross module boundaries and affect other modules. Intermodule change propagation is then utilized to compute the set X_{kj} consisting of the set of modules involved in intermodule change propagation as a consequence of affecting interface variable j of module k. In the worst case logical ripple effect analysis, X_{kj} is calculated by first identifying all the modules for which j is an input parameter or global variable. Then, for each of these modules in X_{kj}, the intramodule change propagation eminating from j is traced to the interface variables within the module. Intermodule change propagation is then utilized to identify other modules affected and these are added to X_{kj}. This continues until the ripple effect terminates or no new modules can be added to X_{kj}. An algorithm for performing this worst case ripple effect has already been developed [7], [8].

The worst case ripple effect tracing can significantly be refined if explicit assumptions exist for each module in the program for its input parameters or global variables. Intermodule change propagation tracing would then examine if a module's assumptions have been violated to determine whether it should become a part of the change propagation. If a module's assumptions have not been violated, then the ripple effect will not affect the module.

There are many possible approaches to refining the worst case ripple effect which would not require a complete set of assumptions made for each interface variable for every module. For example, a significant refinement to the worst case change propagation can result by utilizing the simple approach of examining whether or not a module makes any assumptions about the values of its interface variables. These assumptions can be expressed as program assertions. If it does not make any assumptions about the values of its interface variables, then the module cannot be affected by intermodule change propagation. However, if it does make an assumption about the value of an interface variable, then the worst case is automatically in effect and the module is placed in the change propagation resulting from affecting the interface variable if the interface variable is also in the change propagation as a consequence of some modification.

Both intramodule and intermodule change propagation must be utilized to compute the expected impact of a primitive modification to a module on other modules in the program. A measure is needed to evaluate the magnitude of this logical ripple effect which occurs as a consequence of modifying a variable definition. This measure must be associated

with each variable definition in order that the impact of modifying the variable definition during maintenance can be determined. This logical complexity of modification figure will be computed for each variable definition i in every module k and is denoted by LCM_{ki}. There are many possible measures which may be used for LCM_{ki}. All of these measures are dependent upon computation of the modules involved in the intermodule change propagation as a consequence of modifying i. The modules involved in the intermodule change propagation as a consequence of modifying variable definition i of module k can be represented by the set W_{ki} which is constructed as follows:

$$W_{ki} = \bigcup_{j \in Z_{ki}} X_{kj}.$$

The simplest measure for LCM_{ki} would be the number of modules involved in the intermodule change propagation as a consequence of modifying i. This measure provides a crude measure of the amount of effort required to analyze the program to ensure that the modification does not introduce any inconsistency into the program. Other measures which examine not only the number of modules involved in the intermodule change propagation, but also the individual complexity of the modules, provide more realistic measures of the amount of effort required to analyze the program to ensure that inconsistencies are not introduced. One such easily computed measure is McCabe's cyclomatic number [9]. The cyclomatic number $V(G)$ is defined in terms of the number of basic paths in the module. A basic path is defined as a path in the module that when taken in combination can generate all possible paths. Computation of the cyclomatic number is, thus, based on a directed-graph representation of the module. For such a graph G_j, the cyclomatic number can be calculated as the number of branches in G_j minus the number of vertices in G_j plus two. Utilizing the cyclomatic number or any other complexity measure, the complexity of modification of variable definition i of module k can be computed as follows:

$$\mathrm{LCM}_{ki} = \sum_{t \in W_{ki}} C_t$$

where C_t is the complexity of module t.

Since the logical stability of a module is defined as the resistance to the potential logical ripple effect of a modification to a variable definition i on other modules in the program, the probability that a particular variable definition i of a module k will be selected for modification, denoted by $P(ki)$, must be determined. Now, a basic assumption of utilizing primitive types of maintenance activity is that a modification can occur with equal probability at any point in the module. This implies that each occurrence of each variable definition has an equal probability of being affected by the maintenance activity. Thus, for each module we can calculate the number of variable definitions. If the same variable is defined twice within a module, each definition is regarded separately. The probability that a modification to a module will affect a particular variable definition in the module can then be computed as 1/(number of variable definitions in the module).

With the information of LCM_{ki} and $P(ki)$ for each variable definition i of a module k, the potential logical ripple effect of a primitive type of modification to a module k, denoted by LRE_k, can be computed. The potential logical ripple effect of a module is a measure of the expected impact on the program of a primitive modification to the module. Thus, the potential logical ripple effect can be computed as follows:

$$\mathrm{LRE}_k = \sum_{i \in V_k} [P(ki) \cdot \mathrm{LCM}_{ki}]$$

where V_k is the set of all variable definitions in module k.

A measure for the logical stability of a module k, denoted by LS_k, can then be established as follows:

$$\mathrm{LS}_k = 1/\mathrm{LRE}_k.$$

Development of a Program Logical Stability Measure

A measure for the potential logical ripple effect of a primitive modification to a program, denoted by LREP, can easily be established by considering it as the expected value of LRE_k over all of the modules in the program. Thus, we have

$$\mathrm{LREP} = \sum_{k=1}^{n} [P(k) \cdot \mathrm{LRE}_k]$$

where $P(k)$ is the probability that a modification to module k may occur, and n is the number of modules in the program. A basic assumption of utilizing primitive modifications is that a modification can occur with equal probability to any module and at any point in the module. Utilizing this assumption, the probability that a modification will affect a particular module can be computed as $1/n$, where n is the number of modules in the program. This assumption can be relaxed if additional information regarding the program is available. For example, if the program has only recently been released and it is believed that a significant part of the maintenance activity will involve error correction, then the probabilities that particular modules may be affected by a modification may be altered to reflect the probabilities that errors in these modules may be discovered. This can be accomplished by utilizing some complexity or software science measures [10].

A measure for the logical stability of a program, denoted by LSP, can then be established as follows:

$$\mathrm{LSP} = 1/\mathrm{LREP}.$$

IV. Algorithm for the Computation of the Logical Stability Measures

In this section, an algorithm will be outlined for the computation of these logical stability measures. The following description of this algorithm assumes that there does not exist any prior knowledge which might affect the probabilities of program modification, and McCabe's complexity measure [9] is utilized. The algorithm can easily be modified to allow for prior knowledge concerning the probabilities of program modification or to utilize a different complexity measure. The algorithm consists of the following steps.

Step 1: For each module k, identify the set V_k of all variable definitions in module k. Each occurrence of a variable in a variable definition is uniquely identified in V_k. Thus, if the same variable is defined twice within a module, then V_k

contains a unique entry for each definition. The set V_k is created by scanning the source code of module k and adding variables which satisfy any of the following criteria to V_k.

a) The variable is defined in an assignment statement.
b) The variable is assigned a value which is read as input.
c) The variable is an input parameter to module k.
d) The variable is an output parameter from a called module.
e) The variable is a global variable.

Step 2: For each module k, identify the set T_k of all interface variables in module k. The set T_k is created by scanning the source code of module k and adding variables which satisfy any of the following criteria to T_k.

a) The variable is a global variable.
b) The variable is an input parameter to a called module. Each utilization of a variable as an input parameter to a called module is regarded as a unique interface variable. Thus, if variable x is utilized as an input parameter in two module invocations, then each occurrence of x is regarded as a unique interface variable.
c) The variable is an output parameter of module k.

Step 3: For each variable definition i in every module k, compute the set Z_{ki} of interface variables in T_k which are affected by a modification to variable definition i of module k by intramodule change propagation [7], [8].

Step 4: For each interface variable j in every module k, compute the set X_{kj} consisting of the modules in intermodule change propagation as a consequence of affecting interface variable j of module k.

Step 5: For each variable definition i in every module k, compute the set W_{ki} consisting of the set of modules involved in intermodule change propagation as a consequence of modifying variable definition i of module k. W_{ki} is formed as follows:

$$W_{ki} = \bigcup_{j \in Z_{ki}} X_{kj}.$$

Step 6: For each variable definition i, in every module k, compute LCM_{ki} as follows:

$$\mathrm{LCM}_{ki} = \sum_{t \in W_{ki}} C_t$$

where C_t is the McCabe's complexity measure of module t.

Step 7: For each variable definition i in every module k, compute the probability that a particular variable definition i of module k will be selected for modification, denoted by $P(ki)$, as follows:

$$P(ki) = 1/(\text{the number of elements in } V_k).$$

Step 8: For each module k, compute LRE_k and LS_k as follows:

$$\mathrm{LRE}_k = \sum_{i \in V_k} [P(ki) \cdot \mathrm{LCM}_{ki}]$$

$$\mathrm{LS}_k = 1/\mathrm{LRE}_k.$$

Step 9: Compute LREP and LSP as follows:

$$\mathrm{LREP} = \sum_{k=1}^{n} [P(k) \cdot \mathrm{LRE}_k]$$

where $P(k) = 1/n$, and n is the number of modules in the program. Then

$$\mathrm{LSP} = 1/\mathrm{LREP}.$$

V. Applications of the Logical Stability Measures

The logical stability measures presented in this paper can be utilized for comparing the stability of alternate versions of a module or a program. The logical stability measures can also be normalized to provide an indication of the amount of effort which will be needed during the maintenance phase to accommodate for inconsistency created by logical ripple effect as a consequence of a modification. Based upon these figures, decisions can be made regarding the logical stability of a program and the modules of which the program is composed. This information can also help maintenance personnel select a particular maintenance proposal among alternatives. For example, if it is determined that a particular maintenance proposal affects modules which have poor stability, then alternative modifications which do not affect these modules should be considered. Modules whose logical stability is too low may also be selected for restructuring in order to improve their logical stability.

The logical stability measures can be normalized by first modifying the computation of the module logical ripple effect measure to include the complexity of the module undergoing maintenance. Let LRE_k^+ denote this new logical ripple effect measure for module k which is calculated as follows:

$$\mathrm{LRE}_k^+ = C_k + \sum_{i \in V_k} [P(ki) \cdot \mathrm{LCM}_{ki}]$$

where C_k is the complexity of module k. This enables LRE_k^+ to become an expected value for the complexity of a primitive modification to module k. Let C_p be the total complexity of the program which is equal to the sum of all the module complexities in the program. Note that $\mathrm{LRE}_k^+ \leqslant C_p$ since the ripple effect is bounded by the number of modules in the program. The normalized logical ripple effect measure for module k, denoted as LRE_k^*, can then be calculated as follows:

$$\mathrm{LRE}_k^* = \mathrm{LRE}_k^+/C_p.$$

The normalized logical stability measure for module k, denoted as LS_k^*, can then be calculated as follows:

$$\mathrm{LS}_k^* = 1 - \mathrm{LRE}_k^*.$$

The normalized logical stability measure has a range of 0 to 1 with 1 the optimal logical stability. This normalized logical stability can be utilized qualitatively or it can be correlated with collected data to provide a quantitative measure of stability.

The normalized logical stability measure for the program, denoted as LSP*, can be computed by first calculating the normalized logical ripple effect measure for the program, denoted as LREP*, as follows:

$$\mathrm{LREP}^* = \sum_{k=1}^{n} [P(k) \cdot \mathrm{LRE}_k^*].$$

The normalized logical stability measure for the program can then be calculated as follows:

$$LSP^* = 1 - LREP^*.$$

LSP^* has the same range and interpretation as LS_k^*.

VI. Example

In this section the logical stability measures for the program in Fig. 2 will be calculated according to the previously described algorithm as follows:

$$LRE_{MAIN} = 4, \quad LRE_{RROOTS} = 2.9, \quad LRE_{IROOTS} = 2.7.$$

The logical stability of each of the modules is given by

$$LS_{MAIN} = 0.25, \quad LS_{RROOTS} = 0.34, \quad LS_{IROOTS} = 0.37.$$

The potential logical ripple effect of the program is

$$LREP = 3.2$$

and hence the logical stability of the program is given by

$$LSP = 0.31.$$

The normalized logical stability measures for each of the modules and the program are given as follows:

$$LS^*_{MAIN} = 0$$

$$LS^*_{RROOTS} = 0.02$$

$$LS^*_{IROOTS} = 0.06$$

$$LSP^* = 0.0267.$$

These measures indicate that the stability of the program in Fig. 2 is extremely poor. An examination of the program provides intuitive support of these measures since the program utilizes common variables in every module as well as shared information in the form of passed parameters. Thus, the change propagation potential is very high in the program.

```
C MODULE MAIN
C SOLUTION OF THE QUADRATIC EQUATION
C A*X*X+B*X+C = 0
      COMMON HR1,HR2,HI
      READ 100 (A,B,C)
100 FORMAT (3F10.4)
      H1 = -B/(2.*A)
      HR = H1*H1
      DISC = HR - C/A
      CSID = H1*H1 - C/A
      CALL RROOTS (CSID,DISC,H1)
      WRITE 100 HR1,HR2,HI
      END


C MODULE RROOTS
      SUBROUTINE RROOTS (CSID,DISC,H1)
      COMMON HR1,HR2,HI
      IF (DISC.LT.0) GOTO 10
      H2 = SQRT (DISC)
      HR1 = H1 + H2
      HR2 = H1 - H2
      HI = 0.
10 CONTINUE
      CALL IROOTS (CSID,DISC,H1)
      RETURN


C MODULE IROOTS
      SUBROUTINE IROOTS (CSID,DISC,H1)
      COMMON HR1,HR2,HI
      IF (CSID.GE.0) GOTO 10
      H2 = SQRT(-DISC)
      HR1 = H1
      HR2 = H1
      HI = H2
10 CONTINUE
      RETURN
```

Fig. 2. An example program for computing the stability measures.

VII. Validation of Stability Measures

As previously mentioned, an important requirement of the stability measures necessary to increase their applicability and acceptance is the capability of validating them. The previous stability measures [3], [4], [6] failed to satisfy this requirement due to calculations involving subjective or difficult to obtain inputs about the program being measured. The stability measures presented in this paper do not suffer from these limitations since they are produced from algorithms which calculate intermodule and intramodule change propagation properties of the program being measured. Thus, these measures easily lend themselves to validation studies.

The stability measures presented in this paper can be validated either directly through experimentation or indirectly through a discussion of how they are influenced by various established attributes of a program which affect its stability during maintenance. The direct approach to validation requires a large database of maintenance information for a significant number of various types of programs in different languages which have undergone a significant number of modifications of a wide variety. One experimental approach would be to examine sets of programs developed to identical specifications but differing in design or coding. Logical stability measures for each version of the program could then be calculated to determine which possesses the best stability. A set of identical modifications to the specifications of each program could then be performed. For each modification to each program, a logical complexity of modification, LCM, could then be calculated based upon the difficulty of implementing the particular modification for the program. One particular method for calculating an LCM has previously been described [7], [8]. After a significant number of identical specification modifications have been implemented on all versions of the program, an average logical complexity of modification, ALCM, could be computed for each version of the program. This ALCM reflects the stability of the program and, thus, the ALCM can be utilized as a variable in the experiment. After a significant number of sets of programs have undergone their sets of modifications, experimental conclusions based upon a statistical analysis of the ALCM figures and the stability measures could be formulated.

This direct approach to validation of the stability measures will be difficult due to the number of programs and modifications necessary to produce significant statistical results. Thus, this direct approach to validation will be performed utilizing the maintenance data base which will be created in conjunction with the validation of our program maintainability measure which is currently under investigation.

The stability measures presented here can also be indirectly validated by showing how the measures are affected by some attributes of the program which affect its stability during maintenance. One program attribute which affects maintainability is the use of global variables. The channeling of communication via parameter-passing rather than global variables is characteristic of more maintainable programs [11]. Thus,

an indirect validation of the stability measures must show that the stability of programs utilizing parameter passing is generally better than that of programs utilizing global variables. This can be easily shown since the calculation of LS_i is based upon the LCM of each interface variable in module i. Since global variables are regarded as interface variables and since the LCM of an interface variable is equal to the sum of the complexity of the modules affected by modification of the interface variable, LS_i will be small for modules sharing the global variable. Thus, the logical stability of the program will also be small. On the other hand, if communication is via parameter passing instead of global variables, the LCM of the parameters will generally be small, and hence LS_i and LSP will generally be improved. Thus, the stability measures indicate that the stability of programs utilizing parameter passing is generally better than that of programs utilizing global variables.

The stability of a program during maintenance is also affected by the utilization of data abstractions. Data abstractions hide information about data which may undergo modification from the program modules which manipulate it. Thus, data abstraction utilization is characteristic of more maintainable programs. An indirect validation of the stability measures must, therefore, show that the stability of programs utilizing data abstractions is generally better than that of programs whose modules directly manipulate data structures. This can easily be shown by examining the stability measures of a program that utilizes data abstractions and comparing those measures to that of an equivalent program in which the modules directly access the data structure, i.e., data abstractions are not utilized. The modules which utilize a data abstraction to access a data structure will have fewer assumptions about their interface variables and hence have higher stability than that of the modules directly accessing the data structure and hence having many assumptions about it. For example, consider a data structure consisting of records where each record has an employee number and a department number. Assume that module INIT initializes the data structure and orders the records by the employee number. Also, assume modules X, Y, and Z must access the data structure to obtain the department for a given employee number. In this design, if module INIT is modified so that the records in the data structure are ordered by the department instead of the employee number, then modules X, Y, and Z must also be modified. This potential modification is reflected in the calculation of LS_{INIT} and, consequently, LSP. If, however, modules X, Y, and Z access the data structure through a data abstraction, then the same modification to module INIT will affect the data abstraction algorithm, but not modules X, Y, and Z. Consequently, LS_{INIT} and, consequently, LSP will be larger in the program which utilizes the data abstraction than the measures for the program which does not. Thus, the stability measures proposed in this paper indicate that the stability of programs utilizing data abstractions is generally better than that of programs which do not.

Another attribute affecting program stability during maintenance is a program control and data structure in which the scope of effect of a module lies within the scope of control of the module. This implies that the only part of a program affected by a change to a module, i.e., its scope of effect, is a subset of the modules which are directly or indirectly invoked by the modified module, i.e., its scope of control [12]. An indirect validation of the stability measures must, therefore, show that the stability of programs possessing this type of control and data structure are better than that of programs which do not possess this attribute. Now a program which exhibits this scope of effect/scope of control property has a logical stability which is calculated from the logical stability of its modules, each of which is bounded above by the sum of the complexity of the modules which lie within its scope of control. If the scope of effect of a modification to a module does not lie within the scope of control of the module, the logical stability of the module is only bounded above by the complexity of the entire program. Thus, the stability measures indicate that the stability of programs possessing the scope of effect/scope of control attribute are generally better than that of programs which do not possess this attribute.

Another attribute affecting program stability during maintenance is the complexity of the program. Program complexity directly affects the understandability of the program and, consequently, its maintainability. Thus, an indirect validation of the stability measures must, therefore, show that the stability of programs with less complexity is generally better than that of programs with more complexity. This is readily apparent from the calculation of the logical complexity of modification of an interface variable. Thus, complexity is clearly reflected in the calculation of the stability measures.

The stability measures presented here can, thus, be indirectly validated since they incorporate and reflect some aspects of program design generally recognized as contributing to the development of program stability during maintenance.

VIII. Conclusion and Future Research

In this paper, measures for estimating the logical stability of a program and the modules of which the program is composed have been presented. Algorithms for computing these stability measures and for normalizing them have also been given. Applications and interpretations of these stability measures as well as an indirect validation of the measures have been presented.

Much research remains to be done in this area. One area of future research involves the application of the logical stability measures to the design phase of the software life cycle. An analysis of the control flow and the data flow of the design of the program should provide sufficient information for calculation of a logical stability measure during the design phase.

Another area of future research involves the development of a performance stability measure. Since a program modification may result in both a logical and a performance ripple effect, a measure for the performance stability of a program and the modules of which the program is composed is also necessary [7], [8].

Much research also remains to be done in the identification of the other software quality factors contributing to maintainability. Suitable measures for these software quality factors must also be developed. These measures must then be integrated with the stability measures to produce a maintainability

measure. This maintainability measure must be calculatable at each phase of the software life cycle and must be validated.

Another area of future research involves the development of automated restructuring techniques to improve both the stability of a program and the modules of which the program is composed. These restructuring techniques should be applicable at each phase of the software development. Restructuring techniques must also be developed to improve the other quality factors contributing to maintainability. These restructuring techniques must automatically improve the maintainability of the program at each phase of its development. The net results of this approach should be a significant reduction of the maintenance costs of software programs and, consequently, a substantial reduction in their life cycle costs. Program reliability should also be improved because fewer errors may be injected into the program during program changes due to its improved maintainability.

REFERENCES

[1] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48–59, May 1973.

[2] M. V. Zelkowitz, "Perspectives on software engineering," *ACM Comput. Surveys*, vol. 10, pp. 197–216, June 1978.

[3] N. L. Soong, "A program stability measure," in *Proc. 1977 Annu. ACM Conf.*, pp. 163–173.

[4] G. J. Myers, *Reliable Software through Composite Design*. Petrocelli/Charter, 1979, pp. 137–149.

[5] J. A. McCall, P. K. Richards, and G. F. Walters, *Factors in Software Quality, Volume III: Preliminary Handbook on Software Quality for an Acquisition Manager*, NTIS AD-A049 055, Nov. 1977, pp. 2-1–3-7.

[6] F. M. Haney, "Module connection analysis," in *Proc. AFIPS 1972 Fall Joint Comput. Conf.*, vol. 41, part I, pp. 173–179.

[7] S. S. Yau, J. S. Collofello, and T. M. MacGregor, "Ripple effect analysis of software maintenance," in *Proc. COMPSAC 78*, pp. 60–65.

[8] S. S. Yau, "Self-metric software–Summary of technical progress," Rep. NTIS AD-A086-290, Apr. 1980.

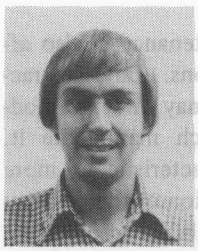
[9] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.

[10] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977, pp. 84–91.

[11] L. A. Belady and M. M. Lehman, "The characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 106–139.

[12] E. Yourdon and L. Constantine, *Structured Design*. Yourdon, 1976.

Stephen S. Yau (S'60–M'61–SM'68–F'73), for a photograph and biography, see p. 434 of the September 1980 issue of this TRANSACTIONS.



James S. Collofello (S'78–M'79) received the B.S. and M.S. degrees in mathematics/computer science from Northern Illinois University, Dekalb, in 1976 and 1977, respectively, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1978.

After graduating, he was a visiting Assistant Professor in the Department of Electrical Engineering and Computer Science, Northwestern University. He joined the faculty of the Department of Computer Science, Arizona State University, Tempe, in August 1979 and is currently an Assistant Professor there. He is interested in the reliability and maintainability of computing systems and the development, validation, and application of software quality metrics.

Dr. Collofello is a member of the Association for Computing Machinery and Sigma Xi.