# AN ANALYSIS OF ERRORS AND THEIR CAUSES IN SYSTEM PROGRAMS

Albert Endres, IBM Laboratory
Boeblingen, Germany

Keywords

Programming methodology

Program testing

Software reliability

Software error classification

Abstract

Program errors detected during internal testing of the operating system DOS/VS form the basis for an investigation of error distributions in system programs. Using a classification of the errors according to various attributes, conclusions can be drawn concerning the possible causes of these errors. The information thus obtained is applied in a discussion of the most effective methods for the detection and prevention of errors.

Contents



## Introduction

It should be useful for all who attempt to improve the reliability of software to know as much as possible about the types of errors that actually occur in programs. Almost everyone who has ever written a program that did not immediately function as intended - a normal occurence as we all know - has developed his personal theory about what went wrong in this specific case and why. As a result, the programming style is modified the next time, i.e. the tricks which were unsuccessful are avoided, or more attention is directed to typically error-prone areas.

It would be desirable, of course, that this learning process, which is individually experienced by a competent programmer, be expanded to include a larger group of programmers, or even the entire profession. In order to realize this, it is necessary to attempt to generalize the experience accumulated by each individual programmer. This implies that it is essential to identify which errors are made by a large class of programmers.

In my opinion, there are some serious deficiencies in the investigations published to date in this area. As an example, I would cite the work of Moulton and Muller (1). This analysis was based on FORTRAN programs in a university environment (University of Michigan). While a considerable number of programs (about 5000) was analyzed, the average program size was only 38 statements. However, even more characteristic of this kind of study is the fact that the analysis of different types of errors is limited to a purely syntactical classification. The same is true for the statistics of Rubey (2), which he drew from a comparison of FORTRAN, COBOL, JOVIAL, and PL/I. The only conclusion that can be drawn on the basis of these studies is that it is necessary, for example in FORTRAN, to beware of assignment and I/O statements. This, obviously, is not a practical conclusion.

The study by Boies and Gould (3) (among other studies) shows that the problems are normally not found in the syntax of a language. In this study, the conclusion presented is that the proportion of syntax errors is clearly below 15 %. An attempt to include the entire activity from problem definition to coding in a predetermined programming language in the analysis is very well illustrated by Henderson and Snowdon (4). Although only the history of a single error is described here, this type of investigation promises to be the most successful.

The following paper is based on an analysis of errors in system programs. System programs differ from application programs in that they show an exceptionally high degree of parameterization, a broad spectrum of users, and a long life span.

As a result, not only are they subject to exceptionally high quality requirements, but also, due to the growth pattern over several releases, acquire a structure that is no longer clean and cohesive.

The objective of this paper is to investigate what meaningful conclusions, if any, can be obtained from an analysis of errors and also to present the conclusions that can be derived from the specific data of this analysis.

## Object and Method of Investigation

The object of this investigation were the errors discovered during internal tests of the components of the operating system DOS/VS (Release 28), developed in the IBM Boeblingen laboratory. This system was released to customers in the middle of 1973. For a better understanding of this investigation, the following information concerning this project must first be presented. The first version of DOS was released in 1966. Extensions and/or improvements of the system were released first in quarterly and then in semi-annual intervals. The extent of these changes in the different versions (the so-called Releases) varied considerably. The version which is the subject of this discussion includes the most extensive changes ever made to the system. The extensions developed in the Boeblingen laboratory for Release 28 consisted primarily of the following subprojects, all of which are a part of the control program:

(a) support of the virtual storage concept
(b) extension of the system from 3 to 5 partitions, including variable partition priority
(c) support of new card I/O devices
(d) support of an optical display device (CRT) as operator console
(e) several smaller extensions (catalogued procedures, timer per partition, adaptation for VSAM)
(f) adaptation of the spooling subsystem "POWER" to the system changes mentioned above.

Concurrently with these extensions, other additions to the control program were developed, primarily in the Dutch laboratory, and a new assembler, new compilers, and data access methods in several laboratories, including some in the United States.

The code comprising the system is physically divided into macros and modules. Macros are those routines which are contained in the assembler source format when the system is shipped; modules exist in their object form. This distinction is not essential in the following discussion; the term "module" is used for module or macro.

About 500 modules were affected by the activities in the Boeblingen laboratory. The average size of these modules was about 360 lines per module, considering only the executable code, and about 480 lines per module if the comments are counted. The entire project had the following effect on the system:

| | Modules | Instructions old | Instructions new |
|---|---|---|---|
| Completely rewritten | 169 | - | 53K |
| Old and new code | 253 | 97K | 33K |
| Comment change only | 100 | 7K | - |
| Total | 522 | 104K | 86K |

The 190K instructions which are cited represent executable code. Added to this are about 60,000 lines of comments. All modules and macros are written in DOS Macro Assembler language.

As shown in Figure 1, the size of the individual modules varies significantly. The relative magnitude of the change per module also varies greatly. Figure 2 illustrates this for the 253 modules which contain both old and new code. The two figures together lead to the conclusion that the typical project activity consisted of changing or adding about 50 instructions in an existing module of about 200 instructions. These facts should make it evident that many methods recommended for the construction of error-free programs (top-down design, structured programming, and so forth) could hardly be applied in this situation.

The material used for this study was the record of errors found in the modules mentioned previously during a formal test period of five months. This testing phase was only a part, although the most critical part, of the complete test cycle for the system. It had been preceded by the tests conducted on a decentralized basis by the programmers responsible for each individual module or subproject. Each subproject had been tested to the point that it was "ready for integration". This means that the new functions had been verified to the extent that was possible while not all components were at the same level of development.

Conflicts that may have been introduced by the subsequent integration process had also been resolved, so that this centralized test was begun on an operable system. The objective of this phase was to test the complete system with all its components in as many different configurations and with as many functional variations as possible. To achieve this objective, two groups used two different types of test cases. One group from the development department executed test cases already used on an earlier version of the system, but in component and system configurations that varied as much as possible (Regression test). A second and independent group had developed new test cases, based on the externals of the system. These test cases were designed to simulate an acceptance test as it would be performed by a customer (Beta test).

Additional tests were carried out prior to the delivery of the system to the customers, for example, a performance study, special tests for remote data processing, and a field test in the computer center for selected customers.

The material analyzed here, therefore, cannot provide a complete picture of all types of errors found in the course of the project, but only a subset. Typical errors that would appear in the early stages of a project (completely missing routines), with trainee programmers (syntax errors), or after a hectic period of changes are unusually scarce in this sample. The same applies to errors normally found by methods other than by running test cases.

All irregularities of the system found (or suspected) by both testing groups were documented according to their external manifestation, that is, by the effect they produced in a specific test case. This information we call the problem. It was passed to the original development group which analyzed the pro-

blems and wrote a response on a form designated as the error protocol.

The error protocol first classified the problems in one of the following groups:

(a) machine error
(b) user or operator error
(c) suggestion for improvement
(d) duplicate (of a previously identified program error)
(e) documentation error
(f) program error (not previously identified)

The distribution in these classifications is usually dependent on the nature and also the organization of a project. The number of duplicates, for example, is in inverse proportion to the speed with which a correction of an identified problem is made available to the testing group.

Although relevant information certainly may be contained in the other classifications as well, we will concentrate on classification (f) in the following discussion. These are the problems accepted as program errors by the development department. In this study the entire data base contained about 740 problems of which 432 were classified as program errors.

It should be noted that, from the perspective of a user, this distinction is usually not easily accepted. Errors in the classifications (a), (d), and (e) can be as annoying as the program errors themselves. However, we do not intend to examine them in this study because they do not originate in the programming activity per se.

For the sake of completeness, it should also be noted that all of these errors were corrected prior to the release of the system.

## Possibilities and Limitations of an Error Analysis

The 432 error protocols available for the analysis contain the following information for each error:

- administrative data on discovery of the problem (system version, configuration, test case, date of test run, name of tester etc.)
- description of the problem
- administrative data on the correction made (changed modules, date of change, name of programmer, system version into which the correction is to be integrated, etc.)
- code for the cause of the error; originating subproject
- description of the correction made.

Confronted with such comprehensive data, a series of questions come to mind. The following questions seemed relevant to me:

(a) Where was the error made? What is the distribution of errors by modules? Are there any accumulations, that is, modules which were hit especially hard? If so, what is their function? How are they structured?
(b) When was the error made? Errors can be made in each phase of the development cycle, beginning with the external design of the project, during detailed planning of the logic structure, during the original coding phase, while correcting an error, etc.
(c) Who made the error? This can be evaluated in terms of the responsibility of individual groups during the project cycle (design, implementation), or of individual subprojects or even programmers.
(d) What was done wrong? Which particular programming task has not been solved or solved incorrectly? The classification of errors resulting from this question can then be the basis for the following additional questions:
(e) Why was the particular error made? What caused the error? Closely related to this (as we will show later) is the question:
(f) What could have been done to prevent this particular error?
And finally:
(g) If the error could not be prevented, by which procedure can this type of error be detected?

Of course, this series of questions can be expanded further. A relevant question might be: Which type of test case detected which type of error? Also, combinations of the above questions might be of interest; for example (b) and (d) together, which would then be: When is each type of error made?
It will probably be agreed that this catalog of questions is already quite comprehensive. If we could find complete and valid answers to these questions, we would have fewer problems in the future.

The following remarks are intended to illustrate the difficulties that can occur in such an undertaking and the limitations that must be accepted.

There is, of course, the initial question of how we can determine what the error really was. To dispose of this question immediately, we will say right away that, in the material described here, normally the actual error was equated to the correction made. This is not always quite accurate, because sometimes the real error lies too deep, thus the expenditure in time is too great, and the risk of introducing new errors is too high to attempt to solve the real error. In these cases the correction made has probably only remedied a consequence of the error or circumvented the problem. To obtain greater accuracy in the analysis, we really should, instead of considering the corrections made, make a comparison between the originally intended implementation and the implementation actually carried out. For this, however, we usually have neither the means nor the base material. The implementaion originally intended may be either no longer obvious or no longer valid.

For the same reason, the module where the correction was made need not be the module where the error originated. Often the change is made in the module which has most free space available.

For error analysis in an operating system, it seems typical that the description of the problem rarely contains much evidence for the type and cause of the error. Ignoring cases where, for instance, a library service program produces an output listing in which some errors can be found directly, the effect of an error is normally rather indirect. Typical problem descriptions in an operating system are:

The system is caught in a loop

- The device X could not be started, the data set Y could not be read.
- The system stopped with an invalid operation code, invalid storage, disk, or device address.
- The card reader K runs with only half its theoretical speed, etc.

In counting errors it is also not clear what should be considered as a unit. As a consequence of a problem it can happen that one or 20 constants are changed, one or 20 instructions added, and this in one or 5 places in a module, or in one or 5 modules, etc. It cannot be excluded that when one problem is solved, other problems are solved too, problems which the programmer found as he went through the program again (or which he secretely had been aware of for some time). In this study "number of errors" was equated to "number of problems". Problems, however, which arose from the same error (duplicates) had been deducted previously.

Taking into account the limitations just explained, we believe that we can answer, with some reliability, the questions a, b, c and d from the material available. For some questions, however, we have to take into account other information which does not result directly from the error protocols. If we want, for instance, to pursue the question of who caused an error down to a single programmer, we have to consider information about the history of each module which can be found in the system library. It should be mentioned here that some questions can be very dangerous for the personal relations in the group and should, therefore, be addressed only in a very sensitive and objective fashion.

We will ignore the whole complex of questions (b) and (c) and talk about (a) only briefly. We assume that complex (d) will yield the most interesting information. The categorization according to type of errors which we will develop then serves as a basis for further considerations concerning the questions (e), (f), and (g).

## Presentation of some Results and Conclusions

In the following sections a selection of the information which resulted from the described analysis is presented.

Error Distribution by Modules. This information is summarized in three figures. Figure 3 shows the effects of an error in terms of the number of modules to be changed. It is somewhat surprising that more than 85% of the errors could be corrected by changing only one module per error. It contradicts the preconception we have of the interdependence of the modules in an operating system and the frequently heard observation that interfaces between modules, in particular, are sources of errors.

Figure 4 shows the number of errors found per module, i.e., the inverse of the distribution shown in Figure 3. Here those errors which affected several modules as shown in Figure 3 were counted more than once. The three top items with 28, 19, and 15 errors came as no surprise; they were three of the largest modules of the system (all had more that 3,000 instructions). The fourth place in this negative competition was taken by a relatively small module which was known to be very unstable. In general, it will be noted that out of 422 changed or newly-written modules, only 202 had any errors at all (48%). If we then exclude the modules with only one error each, we find that 78% of the errors (400) are concentrated in 21% of the modules (90).

Figure 5 shows three comparisons of error frequency. In each case, a distinction is always made between modules which contain only new code and modules which contain both old and new code. The following is to be noted: the relation between modules with errors and all the modules, i.e., the error density, is the same (48%). Under "number of errors per module", the modules with only new code seem to come out worse at first glance than the modules with mixed code. The relation is reversed, however, if we consider the size of the newly written code. I doubt that this data is especially conclusive. It does seem to confirm the feeling common among programmers that, after a certain point, it is better to rewrite a program completely than to try to save as much of the old code as possible.

Error Distribution by Type of Error. The information presented in this section constitutes the essence of this study. The figures in this section resulted from an analysis of all program corrections that was made after completion of all tests.

To draw meaningful conclusions from the material, the existing data had to be classified and abstracted. Thus the data becomes comprehensible even for people without prior knowledge of this particluar operating system. A disadvantage is, of course, a loss in precision and exemplatory evidence.

Because of the amount of material, the Figures 6-1 to 6-10 are structured in two levels. Figures 6-1, 6-5 and 6-10 show a major classification which we label here Group A, Group B, and Group C respectively. The other figures show further detail for some of the data in Groups A and B. In Group A this additional explanation is given only for 3 out of 6 subgroups, in Group B for 4 out of 7 subgroups. All numbers in the figures are percentages; all are based on the same set of 432 errors. The division into the main groups A, B, and C was done as an afterthought and is justified below.

The errors in Group A are specific to the problem at hand. They are errors in the understanding of the problem and in the choice of an algorithm to solve it. In other words, often the wrong problem was solved, or the algorithm selected was not adequate for the given problem. The subgroups typify the problems to be solved in an operating system. In a different project, e.g. a compiler, other subgroups would appear.

The errors in Group B are specific to the implementation process used. Here the errors lie in the more or less complete and correct implementation of a given algorithm, in the translation of an algorithm into a programming language, etc. When different problems are solved, the same types of errors might occur, as long as the same procedures and tools are used for programming. If different procedures are used, such as higher level languages or off-the-shelf routines, different classifications would appear.

Group C, finally, are not programming errors in the strict sense. They are errors in the code which must

not remain there. After their discovery, at least some can be removed by people who are not programmers or have no detailed knowledge of the project.

What do these figures tell us? Consider first the overall classification: Almost half of all errors (46%) are found in the area of understanding the problem, of problem communication, of the knowledge of possibilities and procedures for problem solving. The other half (38%) are items where we can expect other, presumably better, results if we use different methods. This division in two parts of almost equal size seems to be confirmed in other studies in the same area. This fact is alarming or encouraging, depending on the expectations we had for a hundred percent automation of software production. More specifically, only half of the mistakes can be avoided with better programming techniques (better programming languages, more comprehensive test tools). The other half must be attacked with better methods of problem definition (specification languages), a better understanding of basic system concepts (training, education), and by making applicable algorithms available.

Now to the individual subgroups. What the figures of Group A express can be said as follows: The problems to be solved in an operating system are extremely unstructured. The dependence on machine architecture and configuration details is heavy. Functional demands on the system cannot be formulated precisely. Things are often changed once the programmer has seen their effect on the system. The key problem is the dynamic behaviour of the system. As is well known, the parallelism of processes and events makes the system behaviour difficult to comprehend. We have no good tools to attack this problem.

In Group B we find that typical problems of assembler programming play a major role. Other classifications, for instance the problem of initialization, are such well known phenomena that their appearance in this list is hardly surprising and only the percentage can be of interest. On the whole, the designations chosen for this group convey the impression that trivial mix-ups and omissions occur very often. In the subgroups B2 and B4 this impression may be correct. However, in the subgroups B1 and B3 there is very frequently a more complex and deeply rooted error hidden under the trivial-sounding classification.

Group C, finally, illustrates that there is no way around the technically less attractive tasks in connection with building a large system. They, too, must be performed with pedantic accuracy.

## Cause and Prevention of Errors

The search for the cause of a programming error, for the "Why", must be conducted on several levels. Since programming is a human activity, we really should consider a broad spectrum for our analysis. If we do so, we can distinguish the following causes for errors:

- technological (definability of the problem, feasibility of solving it, available procedures and tools),
- organisational (division of work load, available information, communication, resources),
- historic (history of the project, of the program, special situations, and external influences),
- group dynamic (willingness to cooperate, distribution of roles inside the project group),
- individual (experience, talent, and constitution of the individual programmer),
- other, and inexplicable causes.

It is undeniable that the causes of human erring - and programming errors belong in this category - lie to an important degree in the psychological area (that is in the lower part of the above list). G. Weinberg (5), for instance, has presented many arguments for this point of view.

In the following we will adhere to a very narrow perspective. I interpret as "cause of an error" that which should have been different for the error not to occur. From this viewpoint, cause and prevention of errors are two sides of the same coin. As I will further limit my perspective to the two top groups, the technological and the organisational causes, I will interpret

- cause of error as the discrepancy between the difficulty of the problem and the adequacy of the means applied,
- prevention of errors as all measures capable of reducing this discrepancy.

It is obvious that even in this very limited perspective there are still many degrees of freedom and many uncertainties. In addition, there are always two ways to reduce, in a given situation, the discrepancy described above. One can either increase the means applied to solve a given problem or modify the task so that the available means are more suitable. Both methods lead to certain recommendations which may contribute to preventing errors.

On the basis of these considerations we can once again look at the types of errors (Figures 6-1 to 6-10) and try to associate with each of these groups the technical and organizational causes which may lead to this type of error. Figures 7-1 to 7-7 establish this association for the seven most important types of error which occured in our case. We do not distinguish between error cause and error prevention in these figures, but select the neutral term "error factor" instead.

As we said above, the error factors thus specified indicate at the same time what is relevant to the cause of the error and what could be done in order to prevent the particular error. For example, if we accept as a fact that the cause of an error in device handling (Group A1) is to be found in the lack of clarity of the hardware documentation, then this type of error can be avoided by improving the clarity of the hardware documentation.

This approach is to be preferred because of its constructive aspects. Each result of these studies is a starting point for changes and improvements. The results we get are not always complete and scientifically absolute, but the things we find out can be influenced with technical and organizational means. Proposals based on similar motivations for the prevention of programming errors can be found in the papers of Kosy (6), Elspas (7), and Lowry (8) among others.

The error factors indicated in Figures 7-1 to 7-6 show only the type of technical or organizational

measures by which the particular type of error can be affected. The individual lists are not meant to be complete.

What has been shown by this method is, in any case, the obvious fact that for each type of error there exist different causes and therefore different measures have to be taken to prevent them. In other words, there is not a single cure-all. If, moreover, we think of the levels of possible error causes which were not considered in this study, we find as a result a rather sobering overall picture. The measures we have to take in order to prevent errors are just as varied as the types of errors we can identify, and as complexly structured as are their causes. Any catalog of such measures that we could work out remains by nature fragmentary. This type of study, however, can make that catalog more concrete, and can prioritize the measures within it.

## The Detection of Errors

If we come to the conclusion that possiblities for error prevention can only be incomplete, the question arises whether we can deduce anything from the investigations made so far concerning the possibilities of error detection. It seems indeed useful to contrast the classification of errors by type of error previously presented with the error detection techniques applied today. Corresponding to the division of types of errors in the Groups A and B (Group C will be omitted here) two different sets of procedures are considered.

For Group A the following measures, which are actually applied in practice, seem to be the most important ones:

(a) A careful examination of the functional (external) and the logic (internal) specifications by outsiders (design walk-though). These are usually people who are not directly participating in the project. In the case of an operating system they might be: hardware developers, product planners, sales specialists, application programmers etc.,
(b) Additional or overlapping descriptions of the system by formal methods (e.g. VDL, Petri-nets, Markov models).
(c) Use of analytical or simulation models (which are normally prepared for performance analysis) in order to make the behaviour of the system clear and to understand it better.
(d) Inspection of the written program text by others (code-walk-through). This can be done by other experienced programmers of the project who know the problem to be solved but not the method which was chosen for solution.
(e) Test runs by participants of the project or by outsiders.

For group B, the typical catalog of measures is different. Here most weight has to be put on procedures of program verfication in the strict sense. To these belong:
(a) Program inspection by others (see (d) above)
(b) Proof of programs by the methods of Floyd or Hoare.
(c) Hand simulation of test runs, that is, calculation of examples with pencil and paper.
(d) Test runs by the author of the program.
(e) Test runs by others (i.e. not the author of the program.)

It would certainly be asking too much to try to deduce exact figures on the relative effectiveness of these procedures and measures from the data available here. At best we can attempt a relative evaluation of the different procedures based on experience and subjective judgement. This we ventured to do with Figures 8-1 and 8-2. Although the statements thus obtained may be of a slightly speculative character, they still can throw light on the kind of problems connected with such an assessment. In other words, it is not expected that the particular figures given be taken to be precise, but it is hoped that they give a general indication of how the effectiveness of the measures can be evaluated and expressed.

Figures 8-1 and 8-2 are designed to indicate the suitability of a given error detection procedure for detecting each type of error. This is done by giving the estimated probability (in percent) that a given method would be effective. The figures are not absolute but relative to the set of errors that could be discovered by any of the methods. That means that no information is given about the probability that an error can be discovered at all, but only, if it is detected, which procedure is most likely to be successful. This explains why the sum of the probabilities in each line is 100 %. Of course, the question of absolute probability of success would be of greater interest, but because such an estimate would be even more speculative than the current figures, we have decided to abstain.

## Concluding Remarks

The preceding discussions have - I hope - shown that while the analysis of programming errors is a difficult task, it is also a necessary and useful activity. By means of such an analysis we can derive results to help us find remedies for certain frequently encountered types of errors. The present study was concerned with a specific group of system programs. Therefore, there are still a number of open questions which could not be answered on the basis of the material available. Some of these questions are:
(a) What effect will the application of higher programming languages have on system programming? Which types of errors will appear less frequently?
(b) What general differences are there between control programs and compilers?

Some of the conclusions which were drawn on the basis of our data are certainly open to debate. Perhaps this will stimulate some of my colleagues to think further in the direction indicated and to supplement, support or refute my interpretation.

## Acknowledgements

The author gratefully acknowledges the help received from IBM Böblingen Programming Publication Department in preparing the English version of this paper. Particular thanks are due to Prof. D. Parnas whose interest in the subject was a source of great encouragement to the author.

## Bibliography

(1) Moulton, P.G. and Muller, M.E.: DITRAN - A Compiler Emphasizing Diagnostics, CACM 10, 1, (Jan. 1967).

(2) Rubey, R. et al: Comparative Evaluation of PL/I, ESD-TR-68-150 (Apr. 1968)

(3) Boies, S.J. and Gould J.D.: A Behavioral Analysis of Programming - on the Frequency of Syntactical Errors. IBM Research, Yorktown Heights, N.Y., RC-3907 (June 1972)

(4) Henderson, P. and Snowdon, R.: An Experiment in Structured Programming, BIT 12 (1972) pp. 38 - 53

(5) Weinberg, G.: The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971

(6) Kosy, D.W.: Approaches to Improved Program Validation through Programming Language Design, in W.G. Hetzel (ed): Program Test Methods, Prentice Hall, Englewood Cliffs, N.J. 1973

(7) Elspas, B. et al: Software Reliability, IEEE Computer 4, 1 (January/February 1971)

(8) Lowry, E.: Proposed Language Extensions to Aid Coding and Analysis of Large Programs, IBM Systems Development Division, Poughkeepsie, N.Y., TR 00.1934 (1969)


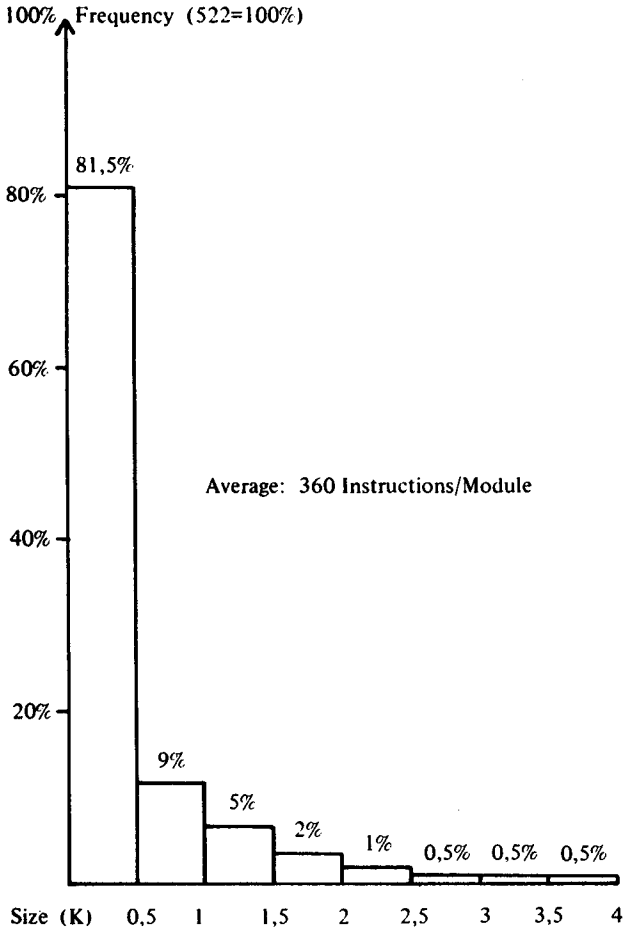


Figure 1. Distribution of modules by module size

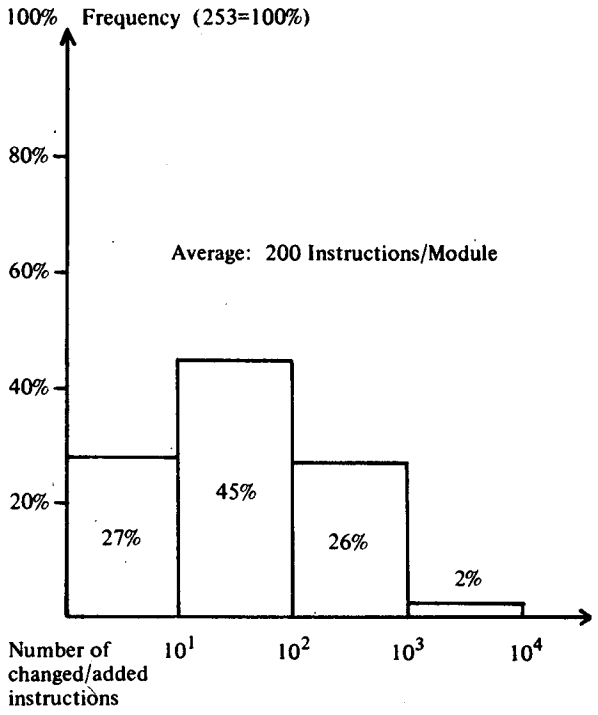


Figure 2. Distribution of modules by extent of change per module

| Number of Errors | Number of Modules Affected |
|---|---|
| 371 | 1 |
| 50 | 2 |
| 6 | 3 |
| 3 | 4 |
| 1 | 5 |
| 1 | 8 |
| 432 | |

Figure 3. Number of modules affected by an error

| Number of Modules | Number of Errors per Module |
|---|---|
| 112 | 1 |
| 36 | 2 |
| 15 | 3 |
| 11 | 4 |
| 8 | 5 |
| 2 | 6 |
| 4 | 7 |
| 5 | 8 |
| 3 | 9 |
| 2 | 10 |
| 1 | 14 |
| 1 | 15 |
| 1 | 19 |
| 1 | 28 |
| 202 | 512 |

Figure 4. Number of errors per module

| Code origin of module | Total number of modules | Number of modules with errors | Percentage of modules with errors |
|---|---|---|---|
| New only | 169 | 81 | 48 |
| Old + new | 253 | 121 | 48 |
| Total | 422 | 202 | 48 |
| | Number of modules | Number of errors | Errors per module |
| New only | 169 | 254 | 1.5 |
| Old + new | 253 | 258 | 1.0 |
| Total | 422 | 512 | 1.2 |
| | Size of new code | Number of errors | Errors per 1K of code |
| New only | 53K | 254 | 4.8 |
| Old + new | 33K | 258 | 7.8 |
| Total | 86K | 512 | 6.0 |

Figure 5. Frequency of errors

| A1 | Machine Configuration and Architecture | |
|---|---|---|
| (a) | Type of device or device feature not considered; valid I/O command taken for invalid | 1.5 |
| (b) | I/O error condition or device status handled incorrectly or not at all | 3.0 |
| (c) | I/O command used incorrectly, or simulated incompletely or incorrectly (in simulating one device by another one) | 4.0 |
| (d) | Error statistics for a device not generated or generated needlessly | 1.0 |
| (e) | External operation mode of a device handled incorrectly | 0.5 |
| | | 10.0 |

Figure 6-2. Types of Errors-Group A1

| A3 | Functions Offered | |
|---|---|---|
| (a) | Functions are completed, as necessary for the intended use of the system | 2.0 |
| (b) | Functions are added, or generalized, although not orginally intended | 1.5 |
| (c) | Functions are completed in order to handle extreme cases and other exceptional situations | 1.5 |
| (d) | Functions are changed in order to improve usability, security, compatibility etc. | 2.0 |
| (e) | Changes caused externally (e.g product strategy) | 2.0 |
| (f) | Defaults for omitted parameters changed | 1.0 |
| (g) | Message for operator/user added | 1.5 |
| (h) | Function is eliminated, because no longer needed | 0.5 |
| | | 12.0 |

Figure 6-4. Types of errors-Group A3

| | | |
|---|---|---|
| A1 | Machine configuration and architecture | 10 |
| A2 | Dynamic behaviour and communication between processes | 17 |
| A3 | Functions offered | 12 |
| A4 | Output listings and formats | 3 |
| A5 | Diagnostics | 3 |
| A6 | Performance | 1 |
| | | 46 |

Figure 6-1. Types of errors - Group A

| A2 | Dynamic Behaviour and Communication between Processes | |
|---|---|---|
| (a) | System state which was entered dynamically not identified exactly enough | 2.0 |
| (b) | In case of sequential transition to another process (especially in forced termination) status not cleared up. The next process does not find the expected parameters (e.g. register contents) | 3.0 |
| (c) | Registers and control blocks used repeatedly were not saved. Interrupt destroys information which is still needed. | 4.0 |
| (d) | Interrupts were enabled which could have been masked out. Other interrupts were masked out and thus ignored, although they were important for the functioning of the system | 3.0 |
| (e) | Logically necessary steps (such as opening a file) were missing, wrong sequence, wrong return branch | 3.0 |
| (f) | Incorrect resource allocation; deadlock, allocation of non-existing or of previously assigned resources. | 1.5 |
| (g) | If one function was not generated (or not activated) a related subsequent function was not eliminated at system generation time (or not switched off at run time) | 0.5 |
| | | 17.0 |

Figure 6-3. Types of errors-Group A2.

| | | |
|---|---|---|
| B1 | Initialization (of fields and areas) | 8 |
| B2 | Addressability (in the sense of the assembler) | 7 |
| B3 | Reference to names | 7 |
| B4 | Counting and calculating | 8 |
| B5 | Masks and comparisons | 2 |
| B6 | Estimation of range limits (for addresses and parameters) | 1 |
| B7 | Placing of instructions within a module, bad fixes | 5 |
| | | 38 |

Figure 6-5. Types of errors-Group B

| B1 | Initialization | |
|---|---|---|
| (a) | Control block, register, switch not cleared or reset before transition from one routine, process, job etc. to another | |
| (b) | I/O area, buffer, etc. not cleared before usage | 5.0 |
| (c) | Fields declared as "Define Storage" (without initialization) instead of as "Define Constant" (with initialization at program loading) | 1.0 |
| (d) | Cleared only part of a field or table | 1.0 |
| (e) | Initialization at wrong time or with wrong value | 0.5 |
| | | 0.5 |
| | | 8.0 |

Figure 6-6. Types of errors-Group B1.

| B3 | Reference to Names | |
|---|---|---|
| (a) | Field has meaning different than assumed (e.g. pointer does not contain address but address of address) | 2.0 |
| (b) | Reference to wrong register or wrong field name (possibly because similarity of abbreviations) | 2.0 |
| (c) | Correct control block found, but reference to wrong relative entry. Table addressed with wrong search argument. | 1.5 |
| (d) | Mix-up of system constants (partition number, SVC number) | 1.5 |
| | | 7.0 |

Figure 6-8. Types of errors-Group B3.

| C1 | Spelling errors in messages and commentaries | 4 |
|---|---|---|
| C2 | Missing commentaries or flowcharts (standards) | 5 |
| C3 | Incompatible status of macros or modules (integration errors) | 5 |
| C4 | Not classifiable | 2 |
| | | 16 |

Figure 6-10. Types of errors-Group C.

| A1 | Machine Configuration and Architecture |
|---|---|
| 1. | Number of different device types and device features. |
| 2. | Device specific properties and variations in error treatment. |
| 3. | Availability and clarity of hardware documentation. |
| 4. | Contact to and communication with hardware developers. |
| 5. | Central or decentralized handling of I/O devices in the system. |
| 6. | Experience in operation of a device. |

Figure 7-1. Error factors-Group A1.

| B2 | Addressability | |
|---|---|---|
| (a) | Assignment, loading, or saving of address registers forgotten (especially when code increases) | 2.0 |
| (b) | Effects of changes in the length of constants, messages, etc. on adjacent storage areas overlooked | 1.0 |
| (c) | Code, areas, and data overwritten since storage was used twice (or more) | 1.0 |
| (d) | Mix-ups between absolute and relocatable, real and virtual addresses (especially when accessing the lower storage areas) | 1.0 |
| (e) | Alignment to word boundaries incorrect | 1.0 |
| (f) | ORG, LTORG added or changed | 0.5 |
| (g) | Splitting of a phase that exceeded defined storage limits | 0.5 |
| | | 7.0 |

Figure 6-7. Types of errors-Group B2.

| B4 | Counting and Calculating | |
|---|---|---|
| (a) | Incorrect calculation or counting of field or record lengths, area sizes (discrepancy not specified) | 2.0 |
| (b) | Like (a) (with discrepancy of 1 byte) | 1.5 |
| (c) | Incorrect displacement (relative address) | 1.0 |
| (d) | Incorrect testing of a loop condition (too early stop, infinite loop) | 1.0 |
| (e) | Programmed counter of records, lines etc. gives wrong values | 1.0 |
| (f) | Transformation from decimal to hexadecimal is missing; binary to decimal transformation is wrong | 0.5 |
| (g) | Calculation of disk addresses, number of tracks, or track capacity is wrong | 1.0 |
| | | 8.0 |

Figure 6-9. Types of errors-Group B4.

| A2 | Dynamic Behaviour and Communication between Processes |
|---|---|
| 1. | Representation of process information in the system (clarity, security) |
| 2. | Structuring of process hierarchy. |
| 3. | Description of interfaces and communication needs of all processes<br>(a) explicit parameters (sequence, meaning, format)<br>(b) shared data areas (implicit parameters) |
| 4. | Standardized routines (macros), supporting process monitoring at a higher level, forcing clearance of system status, etc. |
| 5. | Description techniques for dynamic events, interaction between processes, etc. |
| 6. | Description of resources, their properties, their status. |
| 7. | Central or decentralized handling of supervisory functions, resource allocation, etc. |

Figure 7-2. Error factors-Group A2.

| A3 | Functions Offered |
|---|---|
| 1. | Quality of specifications |
| 2. | Experience with similar systems |
| 3. | Statistical information on user profiles, data volumes, operating modes. |
| 4. | Clarification of and concentration on worst cases, exceptional situations. |
| 5. | Self-discipline with one's own "bright-ideas" and external suggestions. |

Figure 7-3. Error factors-Group A3.

| B1 | Initialization |
|---|---|
| 1. | Forced initialization or warning message by language translator if initialization is missing |
| 2. | Automatic adaptation of operations to field length (e.g. clear the whole field) |
| 3. | Analysis of routines for effect on control blocks, registers, and data fields. |
| 4. | Specification of explicit and implicit parameters, of allowed and expected value ranges. |

Figure 7-4. Error factors-Group B1.

| B2 | Addressability |
|---|---|
| 1. | Extension of symbolic addressing |
| 2. | Extendability of address space |
| 3. | Delineation of address spaces for each routine ("need to know") |

Figure 7-5. Error factors-Group B2.

| B3 | Reference to Names |
|---|---|
| 1. | Syntax of names |
| 2. | Possibility of qualification for names |
| 3. | Representation of the role of a field and indication of routines with access rights |
| 4. | Associative addressing of tables |

Figure 7-6. Error factors-Group B3.

| B4 | Counting and Calculating |
|---|---|
| 1. | Self-decribing data and areas |
| 2. | Powerful loop commands to be linked to data descriptions (e.g. reiterate for all entries of a table) |
| 3. | Tables or easily-accessible transformation routines for the calculation of disk addresses or track capacities |
| 4. | More regularity in the addressing structure for all devices; symbolic addressing |

Figure 7-7. Error factors-Group B4.

| Type of Error \ Detection Method | Examination of specs by others | Formal description methods | Simulation, model building | Program inspection by others | Test runs |
|---|---|---|---|---|---|
| A1 Machine configuration + architecture | 30 | 10 | 10 | 20 | 30 |
| A2 Dynamic behaviour + communication | 10 | 20 | 20 | 20 | 30 |
| A3 Functions offered | 40 | - | - | 20 | 40 |
| A4 Output listings and formats | 30 | 10 | - | 10 | 50 |
| A5 Diagnostics | 20 | 20 | - | 20 | 40 |
| A6 Performance | 10 | 10 | 30 | 10 | 40 |

Figure 8-1. Error detection-Group A.

| Type of Error \ Detection Method | Program inspection by others | Floyd/Hoare method of proof | Simulation of test runs | Test runs by the programmer | Test runs by others |
|---|---|---|---|---|---|
| B1 Initialization | 30 | 10 | 10 | 20 | 30 |
| B2 Addressability | 20 | - | 10 | 30 | 40 |
| B3 Reference to names | 20 | 20 | 10 | 20 | 30 |
| B4 Counting and calculating | 20 | 20 | 20 | 20 | 20 |
| B5 Masks and comparisons | 20 | 10 | 20 | 20 | 30 |
| B6 Estimation of range limits | 30 | 10 | 10 | 20 | 30 |
| B7 Placing of code | 30 | - | - | 30 | 40 |

Figure 8-2. Error detection-Group B.