A Case History Analysis of Software Error Cause-Effect Relationships

Takeshi Nakajo and Hitoshi Kume

Abstract- **Software errors have been studied from various perspectives; however, most investigations have been limited to an individual section or a partial path of the cause-effect relationships of these errors. The present study analyzes approximately 700 errors in 4 commercial measuring-control software products, and then identifies the cause-effect relationships of errors occurring during software development. The analysis method used was: (i) defining appropriate observation points along the path leading from cause to effect of a software error, followed by gathering the corresponding data by analyzing each error using Fault Tree Analysis, and (ii) categorizing each observation point's data, and then summarizing the relationships between two adjoining points using a cross-indexing table. This paper presents four major cause-effect relationships and discusses the effects of the Structured Analysis and Structured Design methods on these relationships.**

Index Terms- **Software development, cause-effect relationships, case-based error analysis, program faults, human errors, process flaws, structured analysis and structured design, Fault Tree Analysis.**

I. Introduction

Many problems due to human error occur during software development, yet effectively eliminating root causes from design processes to prevent their recurrence is difficult unless the types of human errors and their cause-effect relationships are identified [1]. This identification is especially important for long-term improvement of software-development techniques.

Although there have been many quality investigations of human errors in software development (e.g., [2]–[5]), most have been limited to examining an individual section or a partial path of the cause-effect relationships of software errors, with no previous report systematically investigating these relationships.

This paper examines empirical errors in four commercial, measuring-equipment, control software packages, and then identifies the cause-effect relationships of these errors. Section II describes a method used to analyze the cause-effect relationships based on software-error case data. Applying this case-based method, Section III shows that four major error occurrence mechanisms exist, whereas the effects of Structured Analysis and Structured Design methods [6], [7] on these mechanisms are discussed in Section IV. Finally, Section V compares the presented analysis method with the common empirical methods used to identify software-error cause-effect relationships.

Manuscript received January 2, 1990; revised March 22, 1991. Recommended by M. S. Deutsch.

T. Nakajo was with the Faculty of Engineering, University of Tokyo, Bunkyo-ku, Tokyo 113, Japan. He is now with the Department of Industrial and System Engineering, Chuo University, 1-13-27 Kasuga, Bunkyo-ku, Tokyo 112, Japan.

H. Kume is with the Faculty of Engineering, University of Tokyo, Bunkyo-ku, Tokyo 113, Japan.

IEEE Log Number 9101143.

II. Identification of Cause-Effect Relationships Based on Error Case Data

One method of identifying the cause-effect relationships of software errors is to analyze error case data. Fault Tree Analysis (FTA) accomplishes this by using a logic tree diagram to present each error's cause-effect relationships, and develops a countermeasure for each error [8]. Although this method can handle the complex structure of cause-effect relationships, it is difficult for FTA to extract the relationships common to all errors under investigation, because it analyzes each error individually.

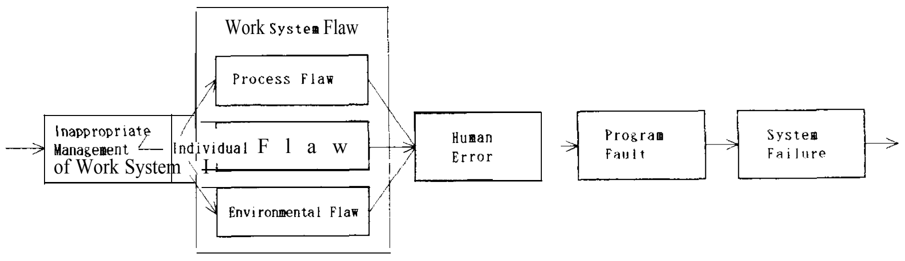
The presented analysis method basically follows FTA, yet facilitates the extraction of the cause-effect relationships inherent in software errors by restricting the logic tree diagram's structure using predefined observation points, and presenting the relationships between two observation points using cross-indexing tables, i.e.,

1) Define observation points along an error's logical path from cause to effect
2) Gather observation point data by analyzing each error using FTA
3) Categorize each observation point's data
4) Identify the relationships between each two adjoining observation points using a cross-indexing table
5) Summarize the major cause-effect relationships based on a series of the cross-indexing tables.

The key elements in the above analysis method are: 1) observation points, and 2) categorization of each observation point's data.

Fig. 1 shows various types of error information along the path from cause to effect. The following considerations must be incorporated when choosing the observation points:

1) A human error occurring during software development is first detected as "system failure"; however, the "system failures" vary widely. Tracing back to the source of a human error reveals that "inappropriate work system management" was ultimately responsible for the error, although the details surrounding this state vary and are strongly dependent on the organization's characteristics.



System Failure	System behavior mismatched with certain system operation specifications
Program Fault	Incorrect program codes defining software behavior
Human Error	Unintended deviations from work standards or targets caused by the carelessness of designers or programers, such as forgetfulness, mistakes or misunderstanding
Work System Flaw	Inherent characteristics of methods, workers and environments affecting human error occurrence and deviation. Classified into three types: Process Flaws, Individual Flaws and Environmental Flaw
Inappropriate Management of Work System	Inappropriate system managing work system elements, such as methods, workers and environments

Fig. 1. Cause–effect process of software errors

It is therefore important to choose observation points where the most common information can be gathered; i.e., somewhere along the cause-to-result process rather than at either the original cause or final result points

2) The final purpose of identifying cause–effect relationships is to eliminate the root causes that produce "system failure"; thus it is important to choose an observation point which indicates the causes to be eliminated. For example, to improve not only individual product quality but also product development techniques, "program faults" and "human errors" as well as "work system flaws" must be included in the set of observation points
3) In the data analysis case using FTA, the effects and their causes have a one-to-many correspondence; therefore it is important to focus on a specific part of the causes. For instance, to improve software design techniques, "process flaws" must be focused among three types of "work system flaws" which mutually cause a human error.

Another key issue in this analysis method is the categorization of each observation point's data. The following considerations must be accordingly included:

1) There are two data classification methods: one is based on predefined criteria, and the other on data similarity. The latter method is called "clustering" and is effective even when there is no theoretical model of the process which produces the data, with the Affinity Diagram method [9] being a non-numerical data-clusteringmethod
2) When a group of results has theoretically no correlation with a group of causes, it is appropriate to independently classify the causes corresponding to each group of results.

III. Cause-Effect Relationships in Measuring Equipment-Control Software

The method described in Section II was applied to 670 software errors occurring during the software development phase of 4 commercial measuring equipment control packages. This analysis identified the cause-effect relationships of software errors in order to improve the developmental techniques of high-reliability software.

Two of the four software packages investigated were subsystems of an IC measuring system, with one being a set of "intrinsic routines" for users' measurement programs, and the other an application program using these routines. The other two were firmware products which control measuring instruments for electronic components. The four products, respectively referred to as A, B, C, and D, have the following common characteristics:

1) Significantly hardware-specific, since they are designed for initializing, setting, and measuring hardware
2) Consist of a set of unit functions that can be combined to produce various required functions
3) Numerous global variables for memorizing hardware equipment status are required.

The errors used in the analysis were all found during or after the module testing phase.

The three following observation points were selected:

1) Program fault
2) Human error
3) Process flaw.

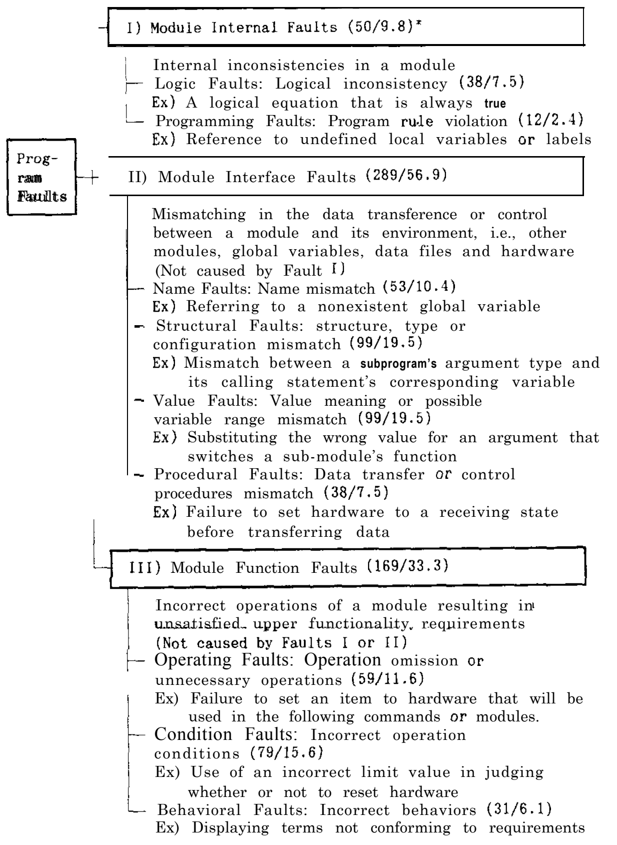
The specifics of each error corresponding to these observation points were collected through source code investigation, design documents and their change histories, and an interview with the designer or programmer who made the error. The Affinity Diagram method was selected for observation point data classification, with Table I showing an example of data collected.

A. Types of Program Faults

Fig. 2 shows the results of "program fault" data classifica-

TABLE I
AN EXAMPLE OF ERROR CASE DATA

Observation Point	Data Collected
Program Fault	*Reading data from a file in the wrong order
Human Error	*Misunderstanding the sequence of the data stored in the file
Process Flaw	*The global variable structure in which the data is stored is different from that of the file -The programmer could not easily refer to the information about the file structure when he was coding the program


Program Faults

I) Module Internal Faults (50/9.8)*

Internal inconsistencies in a module
- Logic Faults: Logical inconsistency (38/7.5)
  Ex) A logical equation that is always true
- Programming Faults: Program rule violation (12/2.4)
  Ex) Reference to undefined local variables or labels

II) Module Interface Faults (289/56.9)

Mismatching in the data transference or control between a module and its environment, i.e., other modules, global variables, data files and hardware (Not caused by Fault I)
- Name Faults: Name mismatch (53/10.4)
  Ex) Referring to a nonexistent global variable
- Structural Faults: structure, type or configuration mismatch (99/19.5)
  Ex) Mismatch between a subprogram's argument type and its calling statement's corresponding variable
- Value Faults: Value meaning or possible variable range mismatch (99/19.5)
  Ex) Substituting the wrong value for an argument that switches a sub-module's function
- Procedural Faults: Data transfer or control procedures mismatch (38/7.5)
  Ex) Failure to set hardware to a receiving state before transferring data

III) Module Function Faults (169/33.3)

Incorrect operations of a module resulting in unsatisfied upper functionality requirements (Not caused by Faults I or II)
- Operating Faults: Operation omission or unnecessary operations (59/11.6)
  Ex) Failure to set an item to hardware that will be used in the following commands or modules.
- Condition Faults: Incorrect operation conditions (79/15.6)
  Ex) Use of an incorrect limit value in judging whether or not to reset hardware
- Behavioral Faults: Incorrect behaviors (31/6.1)
  Ex) Displaying terms not conforming to requirements

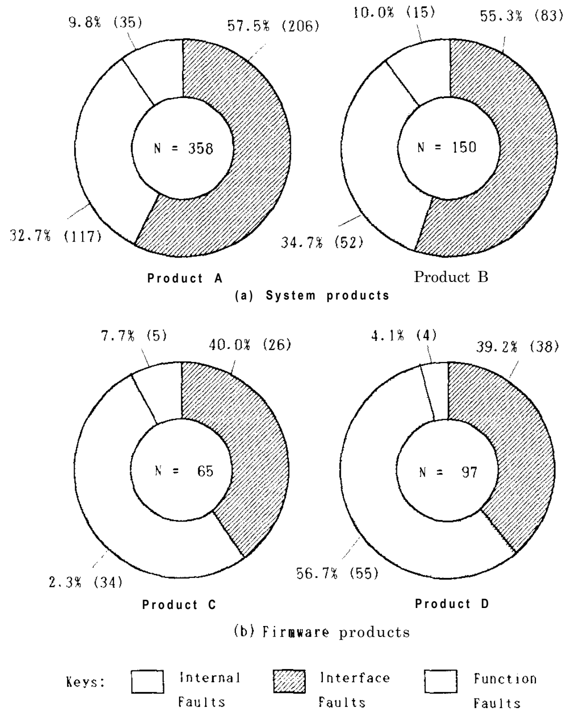
* Numbers in parentheses show the totals and ratios of errors falling under corresponding categories, obtained by classifying 508 program faults in product A and B

Fig. 2. Types of program faults.

tion, where program fault is the observation point nearest to the end of the cause-effect chain.

Fig. 3 and the numbers in parentheses in Fig. 2 show the proportions and total number of program faults corresponding to each of the categories. From these figures the following can be seen:

1) Interface faults and module function faults together constitute more than 90% of the total for all products
2) In the system software programs, interface faults occurred more frequently than module function faults, whereas the latter occurred more frequently in firmware products than the former



Note: (1) Goodness of fit test [10] using a 5% significance level did not reject the hypothesis that the proportion of internal errors is constant for all products.
(2) Goodness of fit test using a 1% significance level rejected the hypothesis that program fault distribution in Products A and B is the same as that in Products C and D.

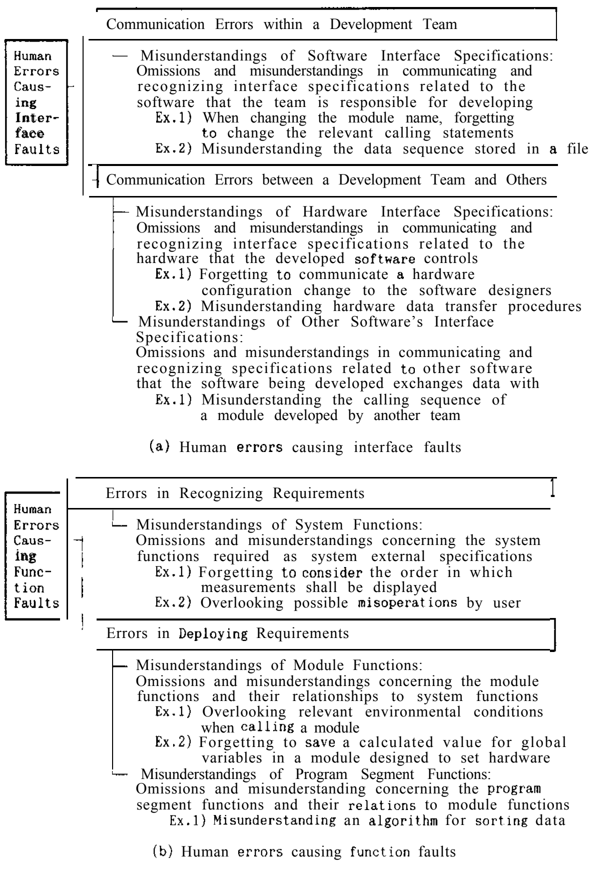
Fig. 3. Proportions of three major types of program faults.

3) In the system software programs, the name, structural, and value faults were the most common types of interface faults, with the operating and condition faults being the most common types of module function faults.

B. Cause–Effect Relationships Between Human Error and Program Fault

This section focuses on the two major program fault types identified in Section III-A: interface fault and function fault, and describes human-error categorization results, followed by examining the relationships between human error and program faults.

An interface fault-mismatching between various software components-is closely related to technical information communication between the various component's development teams. On the other hand, a module function fault is closely related to designing software which satisfies the users' needs. Because it is well known that differences in these types of works greatly affect the types of human errors which occur during the work process, 289 human errors causing interface faults and 169 human errors causing function faults in system software products A and B were independently classified. Fig. 4 shows the human-error categorization results.



Human Errors Causing Interface Faults

Communication Errors within a Development Team

— Misunderstandings of Software Interface Specifications: Omissions and misunderstandings in communicating and recognizing interface specifications related to the software that the team is responsible for developing
Ex.1) When changing the module name, forgetting to change the relevant calling statements
Ex.2) Misunderstanding the data sequence stored in a file

Communication Errors between a Development Team and Others

— Misunderstandings of Hardware Interface Specifications: Omissions and misunderstandings in communicating and recognizing interface specifications related to the hardware that the developed software controls
Ex.1) Forgetting to communicate a hardware configuration change to the software designers
Ex.2) Misunderstanding hardware data transfer procedures

— Misunderstandings of Other Software's Interface Specifications: Omissions and misunderstandings in communicating and recognizing specifications related to other software that the software being developed exchanges data with
Ex.1) Misunderstanding the calling sequence of a module developed by another team

(a) Human errors causing interface faults

Human Errors Causing Function Faults

Errors in Recognizing Requirements

— Misunderstandings of System Functions: Omissions and misunderstandings concerning the system functions required as system external specifications
Ex.1) Forgetting to consider the order in which measurements shall be displayed
Ex.2) Overlooking possible misoperations by user

Errors in Deploying Requirements

— Misunderstandings of Module Functions: Omissions and misunderstandings concerning the module functions and their relationships to system functions
Ex.1) Overlooking relevant environmental conditions when calling a module
Ex.2) Forgetting to save a calculated value for global variables in a module designed to set hardware

— Misunderstandings of Program Segment Functions: Omissions and misunderstanding concerning the program segment functions and their relations to module functions
Ex.1) Misunderstanding an algorithm for sorting data

(b) Human errors causing function faults

Fig. 4. Categories of human errors.

Table II summarizes the relationships between human error and program faults using the categorizations in Figs. 2 and 4. From Table II the following can be seen:

1) More than 50% of interface faults were due to a misunderstanding of software interface specifications; i.e., communication errors within a development team. On the other hand, in the case of communication errors occurring between a development team and others, misunderstanding of hardware interface specifications occurred more frequently
2) Misunderstanding of software interface specifications causes primarily name, structural, or value faults, while misunderstanding of hardware interface specifications causes structural, value, or procedural faults
3) Approximately 45% of the function faults were caused by a misunderstanding of system functions (i.e., an error in recognizing requirements), with about 40% of function faults being caused by a misunderstanding of module functions
4) System function misunderstandings cause mainly condition and behavioral faults, while module function misunderstandings cause operating faults more than condition faults.

TABLE II
PROGRAM FAULT AND HUMAN ERRORS

Program Faults\Human Errors	(a) The Relationships Causing Interface Faults			Total
	Communication within a Development Team	Communication Between a Development Team and Others		
	Misunderstanding of Software Interface Specifications	Misunderstanding of Hardware Interface Specifications	Misunderstanding of Other Software's Interface Specifications	
Name Faults	42	6	5	53
Structural Faults	63	30	6	99
Value Faults	47	42	10	99
Procedural Faults	4	28	6	38
Total	156(54.0)	106(36.7)	27(9 . 3)	289
		133(46.0)		

Program Faults\Human Errors	(b) The Relationships Causing Function Faults			Total
	Requirement Recognition	Requirement Deployment		
	Misunderstanding of System Functions	Misunderstanding of Module Functions	Misunderstanding of Program Segment Functions	
Operating Faults	10	44	5	59
Condition Faults	48	20	11	79
Behavioral Faults	18	4	9	31
Total	76(45.0)	68(40.2)	25(15.8)	169
		93(55.0)		

Note:
(1) These tables resulted from an error analysis on Products A and B.
(2) Goodness of fit test [10] using a 1% significance level rejected the hypothesis that the proportions of program faults due to misunderstandings of software interface specifications are the same as due to misunderstandings of hardware interface specifications, and the hypothesis that the proportions of program faults due to misunderstandings of system functions are the same as due to misunderstandings of module functions.

C. Cause-Effect Relationships Between Process Flaw and Human Error

This section focuses on the following four major human-error types by describing the process flaw categorization results, and then examines the relationships between process flaws and human error.

Error I: Human errors causing interface faults:

1) Misunderstanding of software interface specifications
2) Misunderstanding of hardware interface specifications.

Error II: Human errors causing function faults:

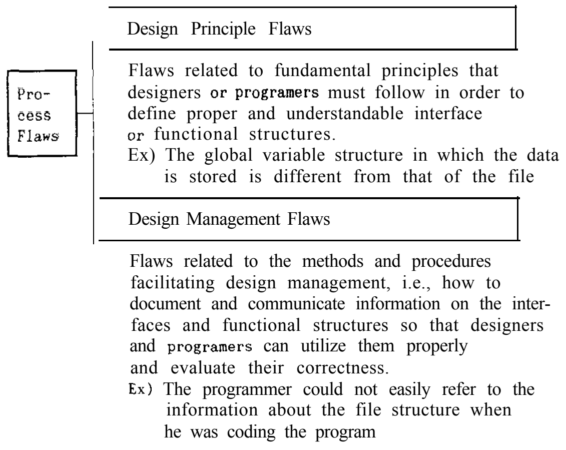
1) Misunderstanding of system functions
2) Misunderstanding of module functions.

Process flaws causing 406 human errors in Products A and B (falling under the 4 above categories) were classified based

TABLE III
HUMAN ERRORS AND PROCESS FLAWS CAUSING INTERFACE ULTS

Huma	Errors\Process Flaws	Design-Principle Flaws	Design-Management Flaws
Errors Causing Interface Faults	Misunderstanding of Software Interface Specifications	**Inappropriate interface definitions prone to be misunderstood and misdeveloped** 1) Definitions inconsistent and distributed (113)* Ex) Parameter structure varies widely for modules when each module is individually defined 2) Complicated correspondence between definitions (53) Ex) Variable structure is different from that of the data to be stored 3) Insufficient discrimination between defined items (37) Ex) Global variables or modules have similar names 4) Ambiguous labels defining items (59) Ex) Flag variable name does not present the system state that it implies	**Lack of methods for recording and referring software interface definitions** 1) Inappropriate communication of module calling information (80) Ex) The necessary operations before calling modules is not clearly documented 2) Inappropriate of communication of global variable or file access information (76) Ex) Global variable value meanings are not clearly documented
	Misunderstanding of Hardware Interface Specifications		**Lack of communication methods between software engineers and hardware engineers** 1) Inappropriate communication of hardware physical configurations (24) Ex) The number of hardware components and the connection between them are shown only by circuit diagrams 2) Inappropriate communication of hardware access information (82) Ex) Methods of reading or writing hardware memories and each memory's meanings are unsystematically documented in hardware specifications

* Numbers in parentheses show the total process flaws of Products A and B in each category. A vital design-principle flaw and a vital design-management flaw are counted for each human error.



Design Principle Flaws

Flaws related to fundamental principles that designers or programers must follow in order to define proper and understandable interface or functional structures.
Ex) The global variable structure in which the data is stored is different from that of the file

Design Management Flaws

Flaws related to the methods and procedures facilitating design management, i.e., how to document and communicate information on the interfaces and functional structures so that designers and programers can utilize them properly and evaluate their correctness.
Ex) The programmer could not easily refer to the information about the file structure when he was coding the program

Fig. 5. Categories of process flaws.

on their similarity. Fig. 5 shows the process flaw categorization results.

Tables III and IV summarize the relationships between process flaws and human errors using the categorizations in Figs. 4 and 5. From these tables the following can be seen:

1) Inconsistent and distributed interface definitions were the major design-principle flaws that cause the misunderstanding of software and hardware interface specifications
2) The design-management flaws that cause the misunderstanding of software interface specifications were inappropriate documentation for module calling methods and/or file and global variable definitions
3) The major design-management flaws that cause the misunderstanding of hardware interface specifications were unclear documentation of hardware accessing methods
4) Dependent/unsymmetrical system unit or module function definitions were the design-principle flaws that cause the misunderstanding of system and module functions
5) The design-management flaws that cause the misunderstanding of system functions were unsystematic requirement documentation and insufficient documentation on how to combine system unit functions to satisfy these requirements
6) The design-management flaws that cause the misunderstanding of module functions were inappropriate documentation of the relationships between system and module functions.

As a result, it is concluded that the error-occurrence mechanisms in the development of these products consist of the four major paths shown in Fig. 6.

IV. Effects of Structured Analysis and Structured Design Methods on Error-Occurrence Mechanisms

This section describes the effects of Structured Analysis (SA) and Structured Design (SD) methods [6], [7] on the previously identified four error-occurrence mechanisms. Although the SA/SD methods are generally used in designing simply structured software, when used as design-management tools they can help suppress these error mechanisms.

A measuring-equipment control system product, similar to Products A and B, was chosen to evaluate the SA/SD methods' effects on error-occurrence mechanisms. For comparison, six

TABLE IV
HU[illegible]N ERRORSAND PROCESS FLAWS CAUSING FUNCTION [illegible]ULTS

Human Errors\Process Flaws		Design-Principle Flaws	Design Management Flaws
Errors Causing Function Faults	Misunderstanding of System Functions	**Complicated correspondence between requirements and the means of realizing them** 1) Functions defined unsymmetrically (91)* Ex) Some commands do not make sense and are unexecutable under certain conditions 2) Functions defined dependently (53) Ex) Functions related to user interface and equipment control are included in a module because module structures are designed so as to satisfy performance requirements	**Lack of systematic methods to describe external system functions** 1) Inappropriate documentation of users' requirements and system unit functions (71) Ex) The conditions where each command can be executed are documented using natural language 2) Inappropriate documentation of relationships between requirements and system unit functions (5) Ex) How to combine commands for function realization is undocumented
	Misunderstanding of Module Functions		**Lack of methods describing module functions and the relationships between module and external system functions** 1) Inappropriate documentation of module functions (15) Ex) The conditions where each module can be executed are not completely documented 2) Inappropriate documentation of relationships between system unit and module functions (53) Ex) The relationships between commands and modules are incompletely documented

* Numbers in the parentheses show the total process flaws of Products A and B in each category. A al design-principle flaw and a vital design-management flaw are counted for each human error

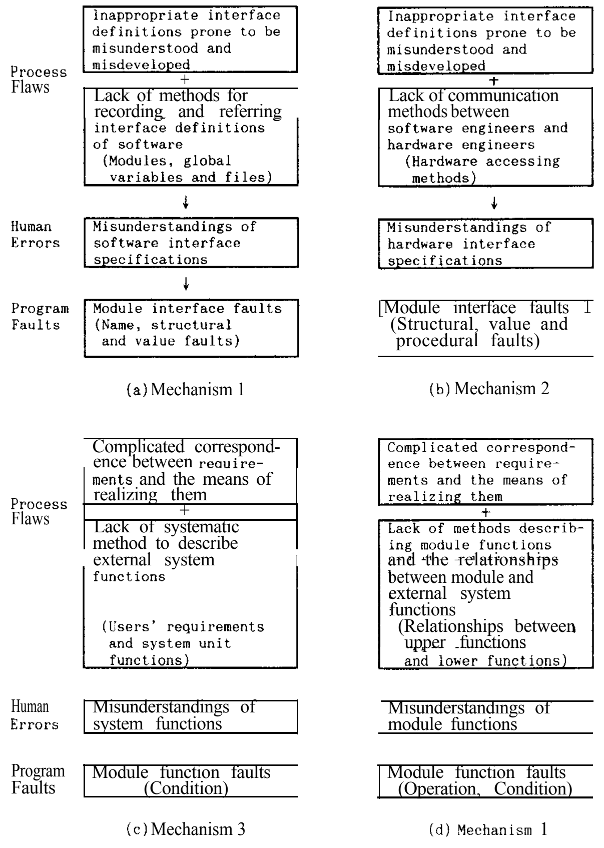

Process Flaws
Inappropriate interface definitions prone to be misunderstood and misdeveloped
+
Lack of methods for recording and referring interface definitions of software (Modules, global variables and files)
Human Errors
Misunderstandings of software interface specifications
Program Faults
Module interface faults (Name, structural and value faults)
(a) Mechanism 1
Inappropriate interface definitions prone to be misunderstood and misdeveloped
+
Lack of communication methods between software engineers and hardware engineers (Hardware accessing methods)
Misunderstandings of hardware interface specifications
Module interface faults (Structural, value and procedural faults)
(b) Mechanism 2
Process Flaws
Complicated correspondence between requirements and the means of realizing them
+
Lack of systematic method to describe external system functions (Users' requirements and system unit functions)
Human Errors
Misunderstandings of system functions
Program Faults
Module function faults (Condition)
(c) Mechanism 3
Complicated correspondence between requirements and the means of realizing them
+
Lack of methods describing module functions and the relationships between module and external system functions (Relationships between upper functions and lower functions)
Misunderstandings of module functions
Module function faults (Operation, Condition)
(d) Mechanism 1

Fig. 6. Error-occurrence mechanisms of measuring equipment control software.

TABLE V
CHARACTERISTICS AND DEVELOPMENT CONDITIONS OF THE COMPARED SUBSYSTEMS

Characteristic Development Condition\Subsystems	Subsystems Developed using SA/SD Methods	Subsystems Developed without using SA/SD Methods
Number of Subsystems*	6(including 5 intrinsic subsystems)	6(including 4 intrinsic subsystems)
The Size of Codes	24.9 KNCSS**	27.0 KNCSS
The Size of Reused Codes***	1.7 KNCSS	12.0 KNCSS
Program Language	C	C
Number of Programers	5 persons	6 persons
Experience of Programers	3–4 years	3–4 years
System Specification	No difference	
Development Environment		
Other Development Processes		

* Subsystems of a single system were selected and compared.
* * KNCSS = 1000 Noncommented source statements.
* * * All codes reused were reviewed and modified if necessary.

product subsystems were developed using SA/SD methods (SA/SD group), and six other subsystems of the same product were developed without using SA/SD methods (Control group). Table V shows the characteristics and development conditions for both groups. All errors discovered during and after the module testing phase were analyzed and compared.

Fig. 7 shows the occurrence rates and proportions of the three program fault types for both groups, with Table VI giving the relationships between human error and program fault for the SA/SD group. This information, combined with that of Fig. 3 and Table II, yielded the following:

1) The internal fault proportion was constant for the four test groups; i.e., SA/SD, Control, and Products A and B
2) The internal fault-occurrence rate in the SA/SD group was the same as in the Control group
3) The interface fault proportion in the Control group was

TABLE VI
PROGRAM FAULTS AND HUMAN ERRORS IN THE SUBSYSTEMS DEVELOPED USING SA/SD METHODS

(a) The Relationships Causing Interface Faults

	Communication within a Development team	Communication between a Development Team and Others		
Program Faults\Human Errors	Misunderstanding of Software Interface Specifications	Misunderstanding of Hardware Interface Specifications	Misunderstanding of Other Software's Interface Specifications	Total
Name Faults	1	0	0	1
Structural Faults	7	2	1	10
Value Faults	1	2	0	3
Procedural Faults	1	0	0	1
		4(26.7)	1(6.7)	
Total	10(66.7)	5(33.3)		15

(b) The Relationships Causing Function Faults

	Requirement Recognition	Requirement Deployment		
Program Faults\Human Errors	Misunderstanding of System Functions	Misunderstanding of Module Functions	Misunderstanding of Program Segment Functions	Total
Operating Faults	3	10	0	13
Condition Faults	4	7	1	12
Behavioral Faults	4	0	0	4
		17(58.6)	1(3.5)	
Total	11(37.9)	18(62.1)		29

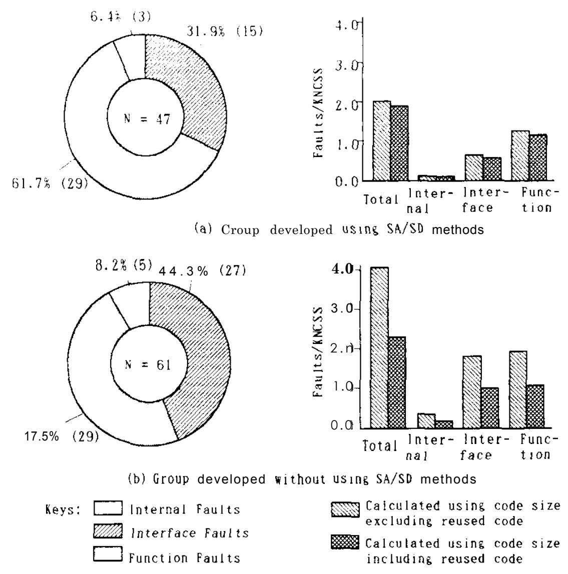
the same as in Products A and B, whereas its proportion in the SA/SD group was smaller than in Products A and B
4) The interface fault-occurrence rate in the SA/SD group was about a half of the Control group's
5) The function fault-occurrence rate in the SA/SD group was approximately 35% smaller than that in the Control group
6) Although the amount of data was not sufficient to determine the relationships between human error and program faults, for the SA/SD group the proportion of structural faults due to software-interface specification misunderstandings increased, and the proportion of condition faults due to system-function misunderstandings decreased.

It is concluded that in this experiment the SA/SD methods suppressed the four error-occurrence mechanisms, and also that the effects on 1 and 2 were greater than those on 3 and 4.

V. DISCUSSION

The metric-based method is a common method of empirically analyzing software error cause-effect relationships [5], [11], [12]. This is an application of the method used to analyze hardware cause-effect relationships, and primarily uses correlation coefficients between two metrics measuring results and their causes. The method's general outline is:

1) Divide the products under investigation into appropriate-sized units for analyzing cause–effect relationships; i.e., systems, subsystems, or modules
2) Define the metrics which measure the unit characteristics and the unit-producing processes and then gather data
3) Calculate the correlation coefficients between the unit's metrics and the processes' metrics and then evaluate them.



Note: (1) Goodness of fit test [10] using a 5% significance level did not reject the hypothesis that internal fault proportion is constant for four test groups. i.e., SA/SD, Control, and Products A and B, the hypothesis that the internal fault occurrence rate in SA/SD group is the same as in Control group, and the hypothesis that Interface fault proportion is constant for Control group and Products A and B.
(2) Goodness of fit test using a 1% significance level rejected the hypothesis that interface fault proportion is constant for SA/SD group and Products A and B, and the hypothesis that the interface fault occurrence rate based on the code size excluding reused code in SA/SD group is the same as in Control group. Even including the reused code, the test statistic value was near to the 5% significance paint.
(3) The statistic value of goodness of fit test for the hypothesis that the function fault occurrence rate in SA/SD group is the same as in Control group was near to the 5% significance point when excluding reused code.

Fig. 7. Effects of SA/SD methods on program faults.

The key elements of this metric-based method are the size of the units used in Sten 1 and the metric types defined in

Step 2, especially the process characteristic measurements. Method effectiveness is highly dependent upon these elements (e.g., large units decrease the amount of data for calculating correlation coefficients and do not isolate the effects of various causes, whereas small units increase data variance and the respective correlation coefficient variance). Usually trial-and-error techniques determine the appropriate unit sizes, and similarly the process characteristic metrics are primarily selected based on either specific engineering models or experience.

Another approach in determining the cause-effect relationships of software errors is to individually analyze each error. FTA is one such method, where unit sizes and process characteristic metrics are not required to be determined in advance. FTA is flexible enough to handle each error's complex cause-effect relationships; however, this flexibility makes it difficult for FTA to extract the relationships common to various errors. Additionally, analyzing more than one error at a time in order to obtain a fault tree diagram presenting common cause-effect relationships often results in a subjective analysis unrelated to individual errors. Another FTA method weakness is the number of man-hours necessary to collect data. In the metric-based method, data can be automatically collected once the metrics are established. Contrastingly, the FTA method is difficult and inappropriate to limit the data range in advance, and as a result requires more data-collection time. This degree of difference significantly depends on the development environment, especially on the error-tracking system.

As in FTA, the presented case-based analysis method does not require the unit sizes and process characteristic metrics to be determined in advance and can be applied without specific engineering models of the cause-effect relationships. The case-based method overcomes the FTA weakness by using both predefined observation points and a phased approach which individually tracks each error's case data to extract the common cause-effect relationships inherent in the data. On the other hand, this case-based analysis method still requires a considerable amount of man-hours, because it basically falls under the category of the FTA method. The man-hours necessary for data collection are more critical than those necessary for data analysis; however, the data collection time can be fairly reduced once appropriate observation points are established.

VI. Conclusions

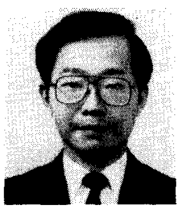
The paper focused on measuring-equipment control software and identified cause-effect relationships of software errors. Four major error-occurrence mechanisms were identified-two are related to hardware and software interface specification misunderstandings, and the other two are related to system and module function misunderstandings. The effects of the Structured Analysis and Structured Design methods on software errors were also evaluated using these mechanisms. The structured methods could suppress the error-occurrence mechanisms, being more effective on hardware and software interface mechanisms than on system and module function mechanisms. The presented case-based cause-effect relationship analysis method can be expected to supplement the metric-based and FTA methods. These conclusions were drawn from specific software packages and are thus limited to them. Given the similarities of software development procedures, however, the presented results can be generalized and applied to the development of other software products.

Acknowledgment

The authors wish to thank the following employees of Yokogawa-Hewlett-Packard Co. Ltd.: M. Mori (Instrument Division Manager) for his beneficial advice, and K. Sasabuchi, T. Akiyama, and I. Azuma (Product Assurance Section), and F. Turuda, M. Tada (Research and Development Sections), and many other software engineers for their help with the collection and analysis of software-error information.

References

[1] W. S. Humphrey, ***Managing the Software Process.*** Reading, MA: Addison-Wesley, 1988, pp. 363-388.
[2] A. Enders, "An analysis of errors and their causes in system programs," IEEE Trans. Software ***Eng.***, vol. SE-l, pp. 140–149, June 1975.
[3] N. F. Schneidewind and H. M. Hoffman, "An experiment in software error data collection and analysis," ***IEEE Trans. Software Eng.,*** vol. SE-5, pp. 276–286, May 1979.
[4] E. A. Young, "Human errors in programming," ***Int. J. Man-Machine Studies,*** vol. 6, no. 3, pp. 361-376, May 1974.
[5] V. R. Basili and T. Perricone, "Software errors and complexity: an empirical investigation," ***Commun. ACM,*** vol. 27, no. 1, pp. 42-52, Jan. 1984.
[6] T.DeMarco, *Structured Analysis and System Specification.* Englewood Cliffs, NJ: Prentice-Hall, 1979.
[7] C. Gane and T. Sarson, ***Structured System Analysis: Tools and Techniques.*** New York: Improved System Technologies, 1979.
[8] N. G. Leveson and R. R. Harvey, "Software fault tree analysis," ***J. Syst. Software***, vol. 3, no. 2, pp. 173-182, June 1983.
[9] S. Mizuno, ***Management for Quality Improvement: The 7 New QC Tools.*** Cambridge, MA: Productivity Press, 1988, pp. 115-142.
[10] I. Guttman and S. S. Wilks, ***Introductory Engineering Statistics.*** New York: Wiley, 1965, ch. 12.
[11] V. R. Basili and D. Rombach, "Tailoring the software process to project goals and environments,"in ***Proc. 9th Int. Conf. Software Eng.,*** Mar. 1987, pp. 345-357.
[12] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," ***IEEE Trans. Software Eng.,*** vol. SE-12, pp. 733-743, July 1986.



Takeshi Nakajo received the B.S., M.S., and Ph.D. degrees in engineering from the University of Tokyo, Japan, in 1979, 1981, and 1986, respectively.

He was a Research Assistant at the University of Tokyo from 1987 to 1990, and has been a Lecturer at Chuo University since 1990, where his special field is in quality improvement, especially the prevention of human errors, with his current research interests being in software-error analysis and design and testing methods. Dr. Nakajo is a member of IEEE Computer Society.

Hitoshi Kume received the B.S., MS., and Ph.D. degrees in engineering from the University of Tokyo, Japan, in 1960, 1962, and 1965, respectively.

He was an Associate Professor at the University of Tokyo from 1974 to 1980, and has been a Professor since 1980, where his special field is in quality control. Since 1960 he has been engaged in consulting work for the improvement of quality systems in Japan's various manufacturing industries.

Dr. Kume is a member of the American Society for Quality Control and a Japan Delegate to TC176 (Quality Management and Quality Assurance) of the International Organization for Standardization.