NSG-1267

(NASA-CR-148212) DEVELOPMENT OF A METHODOLOGY FOR CLASSIFYING SOFTWARE ERRORS Final Technical Report, 1 Jan. - 30 Jun., 1976 (Duke Univ.) 93 p HC $5.00 CSCL 09B

N76-26896

Unclas

G3/61 42320

DEPARTMENT OF COMPUTER SCIENCE

DUKE UNIVERSITY

#N76-26896

DEVELOPMENT OF A METHODOLOGY FOR CLASSIFYING SOFTWARE ERRORS

by

Susan L. Gerhart

Grant NSG 1267 to

DUKE UNIVERSITY
Durham, N. C. 27706
Jan.1-June 30, 1976

Investigator: Susan L. Gerhart,
Computer Science Department

Final Technical Report
July 2, 1976

ABSTRACT

Classification of errors in software is an important and difficult problem. Its purpose is to gain insight into the nature of errors in order to develop better methods for preventing and detecting errors and to develop methods for assessing quality and predicting existence of errors in software. The problem is difficult since errors (real or potential) pervade all programming activities and therefore all programming concepts must be utilized in discussing errors.

The problem seems to have a circular character: classification requires definition, but, to a great extent, definition requires classification. To find a way out of this circularity, a mathematical formalization of the intuition behind classification is devised and then extended to a "classification discipline": Every classification scheme should have an easily discernible mathematical structure and certain properties of the scheme should be decidable (although whether or not these properties hold is relative to the intended use of the scheme). Classification of errors then becomes an iterative process of generalization from actual errors to terms defining the errors together with adjustment of definitions according to the classification discipline. Alternatively, whenever possible, small-scale models may be built to give more substance to the definitions.

The classification discipline and the difficulties of definition are illustrated by examples of classification schemes from the literature and a new study of observed errors in published papers of programming methodologies. Several recommendations are made for studies which would further clarify the problem and begin to produce useful classification schemes.

INTRODUCTION

Errors are a fundamental problem in the development and use of high-quality software. As computing systems, especially hardware components, have become more sophisticated, powerful, and reliable, so have the demands on these systems. As computing systems are given more responsibility in life-critical activities, such as flying airplanes and diagnosis and monitoring in medical care, the potential for disastrous results from even a single error has increased immensely. It is also the conventional wisdom, reinforced by various studies, that errors account for a large share of the cost of developing software.

Although errors are an ever-present fact of life for individual programmers, programming projects, and users of programmed systems, there has been relatively little direct study of errors. Studies have concentrated on syntactical (grammatical) errors which impact on the quality of software only in the time consumed by detecting and correcting these errors. Errors which affect the final behavior of a system are harder to study. Many studies have been largely collecting data on errors and analysing these data for some tentative conclusions. The purpose of these studies may be one or all of the following:

1. Insight into activities that give rise to errors or that fail to detect errors
2. Development of better methods for preventing and detecting errors
3. Viewing programming as a dynamic process, modeling the error subprocesses
4. Assessment of quality of software with respect to latent errors

A crucial factor in the execution and validity of these studies is the classification of errors, for it is classification that organizes the error data into meaningful patterns from which abstraction, insight, and conclusions can occur. That is, none of the above purposes can be achieved fully if attention is paid only to errors on a one-at-a-time basis.

The purpose of the research reported on here is the development of a methodology for classification of errors. The research originally proposed included study of two tasks with respect to their errors and an assessment of various methods of preventing and detecting errors. However this soon ran aground on two accounts:

1. A collection of errors in published papers which have been influential in the development of methodologies for preventing and detecting errors, along with additional reading, indicated that the assessment was both premature and too large to be accomplished within the time-span of the research.
2. The specific studies and a reading of previous work showed that there were more fundamental problems that must be addressed.

The most appropriate course of action given these two discouragements seemed to be to attack the problem on a more fundamental level using the insight gained from the failures. This attack used the common approach of factoring the problem into separate subproblems (classification and errors) with the aim of reuniting the subproblems (into classification of errors) after their separate study. The results reported in the rest

of this paper are:

1. Classification can be considered as a general problem independent of errors. This leads to some mathematical definitions which provide the foundation for explicating the structure of classification and evaluation of specific classifications. The mathematics is quite elementary and serves to formalize intuition about classifications.
2. It is not as easy to consider errors separately from classification. The subject of errors brings in the entire realm of programming and computation. The mathematical theory of computation is not fully developed and, even if it were, computation is such a rich area that the mathematics is complicated. Of course, programming is a human activity not easily rendered mathematically. It will be illustrated later that a fundamental difficulty is communication between people whose orientations and experience cover different aspects of computation, say hardware vs. software.
3. When we put the two subjects together, we see that in a way the critical difficulty is circularity: to classify errors requires definitions of errors and definitions require classifications.

The rest of the paper considers first classifications, then errors, then a series of small and incomplete studies of classifications of errors. Our approach is to attack the problem from various angles: mathematical, linguistic, psychological, and technological. The subject

is truly difficult and comprehensive. Our purpose is to expose some of the reasons for the difficulty and to provide a number of small remedies and tentative recommendations for resolving the difficulties. Our claim is that indeed errors can ultimately be classified for the various purposes discussed above but that the task is perhaps harder than one might expect.

Most of the paper focuses on the mathematisation of classification and the appendix, a paper entitled "Observations of Fallibility in Applications of Modern Programming Methodology". That paper illustrates many points about errors and about classification.

I claimed above that errors are discussed relative to individuals with different backgrounds.

Much of this paper must be understood relative to my background: software, rather than hardware; academic, with exposure to, but no direct participation in government and industrial programming activities; programming-in-the-small, rather than programming-in-the-large; theoretical, more than technical or management; verification; more than design, of programmed systems.

2. THE MATHEMATICS OF CLASSIFICATION

When we talk about classification, we mean that there is a set of objects and some means of grouping these objects together into classes such that each class has some common property. We can look at a classification from two different standpoints - the classes and the properties which define (or describe, or characterize) the classes. When we do so, there are some observations we can usually make about the classes: whether they overlap, include all objects, or include any objects. It is also interesting to look at the ways the properties are defined and how properties can be built up from other properties with the concomitant effects on classes.

We will now state some definitions and prove some theorems which attempt to capture these notions about classification. The mathematics is adapted from set theory and logic and is not far beyond the level of the "new math" taught in schools today (A good reference is Stoll, (1)). Someone has said that "mathematics is the science of structure". Our thesis is that these elementary mathematical concepts can make precise for us what we mean when we talk about classification, that is, we can study the structure of classifications. After the definitions, theorems, and a mathematical example, we will examine the definitions in more detail and apply them to classification of errors.

DEFINITIONS AND NOTATION:

Let X be a fixed reference set.

1. A class-defining property is a statement which evaluates to either true or false whenever any reference to a member of X is replaced by an actual member of X. Let c(x) denote a statement referring to x, a member of X. The property c(a) is satisfied by a of X if c(a) evaluates to true.

2. Let c be a class defining property. The class associated with c, denoted $\underline{c}$, is the subset of X which satisfies c, i.e. $\underline{c} = \{x : x \text{ is a member of } X \text{ and } c(x)\}$

3. Two classes $\underline{c}$ and $\underline{c}'$ are mutually exclusive if $\underline{c} \wedge \underline{c}' = \phi$. A class $\underline{c}$ is empty if $\underline{c} = \phi$. (ϕ is the empty set and $\wedge$ denotes intersection, returning exactly the members of both sets.)

4. A classification scheme C is a set of class-defining properties, the class system associated with C, denoted $\underline{C}$, is the set of all classes associated with class-defining properties of C. We will say that C induces $\underline{C}$ on X.

5. A class system $\underline{C}$ is
   a. exclusive if every pair of classes from $\underline{C}$ is mutually exclusive.
   b. inclusive if every member of X is a member of some class of $\underline{C}$.
   c. full if no class of $\underline{C}$ is empty.

6. The product of two classification schemes C and C', denoted C x C', is defined as
   $\{c \xi c' : c \text{ is a member of C and } c' \text{ is a member of C'}\}$
   $(c \xi c')(x)$ is defined as $c(x) \xi c'(x)$. & is the logical and operator, which is true only when both operands are true.

7. A class system $\underline{C}'$ is a refinement of another class system $\underline{C}$ if for every $\underline{c}'$ of $\underline{C}'$ there is a $\underline{c}$ of $\underline{C}$ such that $\underline{c}' \subseteq \underline{c}$ and the union of classes of $\underline{C}'$ equals the union of classes of $\underline{C}$.

8. A classification scheme C' is an extension of a classification scheme C if every property in C' is constructed using the logical operators ξ and $\neg$ (and, negation) on the properties of C.

9. Suppose X and Y are two sets with a relation R on X x Y and that CX and CY are classification schemes defined for X and Y respectively. Then the class relation $\underline{R}$ is defined
   $\underline{cx}$ R $\underline{cy}$ iff for some x and y, members of $\underline{cx}$ and $\underline{cy}$, xRy

Convention: We will say that a classification scheme has a property (exclusive, inclusive, full, refinement, extension) if its associated class system has the property and vice versa.

Example (Mathematical):

Let X= the integers from 1 to 9,
D= {d2, d3}
where d2(x)="x is divisible by 2"
d3(x)="x is divisible by 3"
E= {prime, $\neg$ prime}
where prime (x)="x is prime, i.e. has no divisors except 1 and itself"
$\underline{D}$= { {2,4,6,8,10}, {3,6,9} }
$\underline{\text{prime}}$ ={1,2,3,5,7}
$\underline{\neg\text{prime}}$ ={4,6,8,9}
$\underline{E}$= {$\underline{\text{prime}}$, ($\underline{\neg\text{prime}}$) }

D is not exclusive or inclusive but is full.
E is inclusive, exclusive, and full.

D x E = {d2 ξ prime, d2 $\xi \neg$prime, d3 ξ prime, d3 $\xi \neg$prime}
D x E is a refinement of D and of E.

An extension of D is {d2 ξ d3, d2 ξ ¬d3, ¬d2 ξ d3, ¬d2 ξ ¬d3}
This extension is exclusive, inclusive, and full.

THEOREMS:

1. Any classification scheme C can be extended to an exclusive, inclusive and full classification scheme C'.
   Proof: Suppose C = {c1,...cn}, Let
   C*={c1 ξ c2 ξ ... ξ cn, c1 ξ c2 ξ ...ξ ¬cn, ... ¬c1 ξ ¬c2 ξ ...ξ ¬cn}
   C* is exclusive since every class property has at least one component property which appears both negated and non-negated.
   C* is inclusive since every combination of component properties and their negations is represented and so one must be satisfied by any member of X. Delete the empty properties from C* to get C'.
   For future reference, let us call C* the complete extension of C.

2. If C and D are inclusive and exclusive classification schemes, then C x D is inclusive and exclusive.
   Proof: Consider c1 ξ d1, c2 ξ d2 from C x D.
   Their mutual exclusion follows from mutual exclusion of c1 and c2 if c1 ≠ c2 or the mutual exclusion of d1 and d2 if c1=c2.
   Inclusion follows from the fact that for every x there must be some c and some d satisfied by x, so c ξ d is satisfied by x.

3. If D is inclusive then C x D is a refinement of C.
   Proof: c ξ d ≤ c for every c and d.
   c = union of all cxd such that d is a class of D.
   Therefore the union of all cxd in CxD equals the union of all classes of C.

4. If a classification scheme C is inclusive and exclusive for some set X, then it is inclusive and exclusive for any subset Y of X.
   Proof: Let CX, CY be the class systems induced by C on X and Y, respectively. Then every class cy of CY is a subset of a class cx of CX so CY is exclusive. Every member of Y is a member of X and so satisfies some c of C and therefore is a member of some cy or CY.

Let us now make some observations about the terms and theorems:

a. We have defined class-defining properties relative to a set X. These properties are functions and we are intending that the set X be a subset of the domains of the properties. That is, we assume the properties are well-defined for X. Let us call X the domain.

b. The notion of a classification scheme is more general than the notion of class systems. A class system is specific to the reference set X, but a classification scheme may be defined for many reference sets.

c. We might ask whether there is some kind of "ideal" classification scheme. If we look at X as the largest set of objects to which some classification scheme C might ever be applied, then we can observe:

i. Full means that every class of C is represented by some member of X. This would seem to be a good property to have since otherwise the classification scheme has some superfluous qualities. But it might be that the scheme can only be devised meaningfully as, say, a product of two other classification schemes which just don't completely fit together.

ii. Inclusion is certainly desirable since its absence means that there are some objects which escape discussion in terms of the classification scheme. On the other hand, Theorem 1 shows that an inconclusive scheme can be extended to be inclusive.

iii. Exclusion really depends on the intended usage of the classification scheme and the nature of the class-defining properties. If the purpose is to obtain a precise characterization of each object with respect to certain features, then exclusion should be sought, but the properties may be "fuzzy" or probabilistic so that exclusion is not meaningful. Or it may be highly desirable that the properties not be mutually exclusive, e.g. if there is a probability of failure in determining whether a property is satisfied, but a high overall probability that every object satisfies some property.

Now, if we look at X another way, as a subset of the set of all objects to which C may be applied, full simply means that not every class-defining property has a representative in X; inclusion is probably still desirable, but may mean that X should be pre-classified into the subset which satisfy some property of C, to which C can be applied, and the complementary subset to which some other classification scheme should be applied; exclusion again depends on the context of usage.

d. The notion of extension and the result of Theorem 1 say that there may be ways of building up good classification schemes from a set of properties which initially are satisfied by only a few objects in the reference set. That is, the initial set of properties might be devised by simply generalizing from a few members of X and then extended to some more comprehensive scheme. On the other hand, this approach has the deficiency that some objects are classified only by their failure to have certain properties and this may not give a good description of the properties they do have.

e. The product of two classification schemes is seen to be a well-defined notion and to preserve certain characteristics of classification schemes. Its value comes from the correlations that may be observed from considering what properties from two different schemes are satisfied by an object. Notice that Theorem 2 does not guarantee that full is preserved by product, since it may not be. Thus an empty class in a product scheme may provide useful information.

f. About all we can do with a non-product type of scheme is count the objects, unless the scheme is further refined.

g. The notion of refinement leads to the concept of a hierarchy of classification schemes. We might denote this by

$$C=\{c1:C1, c2:C2, \ldots, cn:Cn\}$$

where ci:Ci means that the classification scheme Ci is well-defined for ci and refines it. Thus all of the subclassification schemes Ci when applied to their respective subclasses induce a refinement on C.

h. Finally, the notion expressed by definition 8 "lifts" a relation between two (possibly very different) sets to a relation between classification schemes over these objects. This might be useful if R were interpreted as, say, "causes" so that a cause-effect relationship between two objects suggests a possible cause-effect relationship for other members of their containing classes.

The mathematical definitions given here could stand some polishing and certainly more theorems could be proved. Our purpose has been to try and capture some of the notions which seem to underly classification activities. We will try to show in the rest of the paper that this endeavor has paid off by allowing us to make specific analysis of various kinds of classification schemes. It also can guide us in deciding what we want from classification schemes and in designing classification schemes. Our claim is that it is extremely important that classification schemes have a clear structure, which structure should be explicated in terms of our definitions (and possibly others) while it is less important that schemes have characteristics such as exclusion, inclusion, and full (or possibly other such characteristics) than that it be possible to decide whether the scheme has these characteristics. That is, our goal is to discipline our thinking about classification by using these mathematically expressed concepts.

We have been able to characterize classification abstractly. Certainly, classification is a common activity that is carried on in other disciplines, e.g. classification of symptoms according to disease or classification of crimes for demographic studies. Some collegues in medical computing pointed me toward a book on "Clustering Algorithms" (Hartigan, (2)) with the warning that the study of clustering, which is nearly synonamous with classification, is fairly new and has only recently found its way into books. The difference between our needs and this work is that it starts with numerical data and measures. Hartigan does list some purposes of classifications which are worth reviewing:

a. to name a class, presumably with some meaningful name

b. to display related objects in such a way that subtle differences are more apparent

c. to summarize so that it is possible to refer to a class by its property rather than its individual objects, i.e. to abstract

d. to predict since if some objects of a class have a property it is reasonable to expect others also to have it

e. to require explanation since clear-cut and compelling clusters require an explanation of their existence and thus promote the development of theories

He also warns that the clustering techniques are not all based on sound probability models and that it is often difficult to evaluate the results and to determine if the clustering is stable.

There may well be more worthwhile work in the area of pattern recognition which I could not find or understand.

However, I believe the problem of classification of errors is at a more fundamental level because it is first necessary to classify errors by their properties (and to define those properties) after which frequencies can be counted and larger patterns ascertained.

3. ERRORS

Consider the following dictionary definitions (3)

error - "deviation from accuracy or correctness; a mistake"

synonyms: blunder, slip, oversight

fault - "defect or imperfection; a flaw; a failing"

"error or mistake"

synonyms: failing, foible, weakness, vice

mistake-"error in action, opinion, or judgment"

Clarification: an error is an unintentional wandering or deviation from accuracy; a mistake is caused by bad judgment or a disregard of rule or principle; a blunder is a careless, stupid, or gross mistake, suggesting awkwardness, heedlessness, or ignorance; a slip is usually a minor mistake made through haste or carelessness.

What is the best word to use when we talk about errors (or whatever they are) in software? The word "fault" also has meanings in geology and electrical engineering that the other terms do not. It seems best to me to use the work "error" consistently when talking about software, based on the fact that errors are made by people with the effect of deviation from accuracy or correctness. When used in the context of software, "fault" seems to take on the meanings "foible, failing, vice" more than its other meanings which have physical connotations. It also seems appropriate not to spare the use of the words blunder and slip, when their meanings report exactly the reasons for an error; although the attached meanings may cause ill feelings, they may be perfectly appropriate.

The purpose of this discussion on definitions is simply to clarify the word I am using and the reasons I am using it. Another reason is to bring out what I see to be a critical problem in dicussing errors - the communication between people with different backgrounds and therefore different vocabularies. To a completely software-oriented person, a fault is something like the San Andreas and there is nothing physical associated with errors. However, a hardware-oriented person is more accustomed to dealing with physical devices which do have imperfections and failings and therefore may try to ascribe the word fault to certain errors. That is not to say that software cannot be affected by faults since programs ultimately reside and are executed on physical devices, but it seems best to restrict attention separately to software errors, hardware faults, and the interaction between the two.

The following example is worked through to demonstrate some of the linguistic problems associated with discussing software errors.

EXAMPLE:

The following program fragment appears in a book on structured programming (reference 6 of the appendix)

```
I=1;
DO WHILE (I<=N & KEY ¬=TAB(I) );
I=I+1;
END;
```

The language is PL/I. Assume that

a. N,KEY,I, and TAB are declared as integers

b. TAB is an array with one subscript ranging from 1 to N

c. N, KEY, and TAB are initialized to positive integers

d. After execution I is compared with N and further action is taken.

The program fragment linearly searches the "table" TAB for KEY i.e. searching starting at 1 until either KEY is found at TAB(I) or I exceeds N, in which case KEY is not in TAB(1 to N). DO WHILE is the loop construct of PL/I and &,<=,¬= are the logical operators "and", " less than or equal", and"not equal". In PL/I, A & B is defined to be true only when both A and B are true.

The program produces interesting results when executed on a large IBM computer with standard IBM software and four existing PL/I compilers. One compiler was an optimizer, while the others were intended for various degrees of debugging and standard (non-optimized) usage. The test data of interest was N=50, TAB(I)=I for I from 1 to N, and KEY=100. The program produced by the optimizing compiler ran correctly while the other PL/I compilers terminated in DATA INTERRUPT and SUBSCRIPTRANGE errors. The cause for the different results for different compilers is the ambiguous definition of & in the PL/I language. The optimizing compiler produces "short-circuited" code which ceases evaluation of A & B and A is found to be false, while the code produced by the other compilers evaluates both operands. In the above program, when I reaches N+1, KEY¬=TAB (I) is still evaluated with resulting violation of subscript range. The language is defined so that each of these compilers is considered to be a correct implementation of the language.

Clearly, there is an error here. Let us consider some of the different ways we might describe and (subsequently classify) this error:

1. It is a logical (semantic) error
    a. The subscript range of TAB is violated

b. An operand of & is caused to be executed when it is possibly undefined.

c. The terminating condition of the loop is possibly undefined.

2. The error is one of "improper termination", i.e. the program blows up.
3. The error is reported variously in hardware-oriented (DATA INTERRUPT) and software-oriented (SUBSCRIPTRANGE) terms so the error is related to both hardware considerations (finite storage) and software protection (subscript checking)
4. The authors' knowledge of PL/I is incomplete in that they should have been aware of this pitfall in PL/I
5. The authors' knowledge of programming languages in general is deficient, since this condition is carefully considered in other languages, e.g. ALGOL W
6. The error is the implementation in PL/I of a multi-exit loop, one where two or more different actions are appropriate upon termination of the loop.
7. The error is in the implementation of the well-understood linear-search algorithm.
8. The error is in the design of the PL/I language in that the necessity and order of evaluation of operands of logical operators is left undefined and therefore is compiler-dependent
9. The error is in reference materials on PL/I (none of the ones I looked at warns of this possibility)
10. The error is in the authors' publication of a program which had not been tested (since the optimizing compiler is quite expensive, testing would probably be with one of the other compilers and at least two cases, KEY in and not in TAB, would have been run)
11. The error is in the authors' reasoning in an informal proof (with an asserion) that the program is correct; they did not prove proper termination
12. The error is possibly transient, in the sense that PL/I compilers have the option of evaluating superfluous operands and the error might appear and disappear with changes in an installation's PL/I compiler

To complete the example, consider some of the other ways the linear search algorithm could be implemented:

(1)
```
DO I=1  TO N WHILE (KEY¬=TAB(I) );
  END
```

is another PL/I version recommended by the authors of the above book, but is not used because the empty body of a loop disturbs some readers

(2)
```
10  IF (I.LE.N) 20,100
20  IF (KEY.EQ. TAB(I) ) 200 , 30
30  I=I+1
    GO TO 10
100 ...
200 ...
```

is a FORTRAN type of implementation which explicitly separates the two ways of exitting the loop

(3)
```
while I<=N and KEY ¬= TAB(I)  do
    I:=I+1
    end
```

is an ALGOL-like construction where and is defined equivalent to if A then B else false.

(4) An even better implementation when there is room is

```
I=1;
TAB (N+1)=KEY;
DO WHILE (KEY¬=TAB(I) );
  I=I+1;
END
```

since this implementation requires only one comparison per iteration while others require two.

The point is that there are many ways of implementing a linear search and the language does have an effect.

Consider the terms used in describing the nature of the error: structured programming, subscript range, linear search, table, optimizing and debuggingand standard compilers, compiled code, short-circuit, evaluation, logical, semantic, undefined, proper termination, blows up, testing, proof, assertion, transient, algorithm implementation, multi-exit loop, etc. If we gave this list to practically any person associated with computing, I doubt that the person would ascribe the same meaning as I did and that many terms would be either completely unfamiliar or completely misinterpreted (or rather interpreted completely different from my intended meaning).

The error might be surprising to some people and completely natural to others. For example, a hardware oriented person would probably be comfortable with the notion of addressing exceptions but might be surprised that the & was not defined to cover exactly this situation; a FORTRAN programmer might be surprised that subscript checking was such a big deal; a language expert might consider the PL/I treatment of & as perfectly appropriate since optimization is so important to them. In my experience, a prominent computer scientist familiar with IBM ways, several graduate students, and many programmers of various kinds did not recognize an error when the problem was presented as "find the error in this program". The first time I encountered the problem was while introducing logical operators to an introductory programming class. They asked quite naturally whether the second operand of A&B was evaluated if A were false. At that moment I did not know the answer. The only other persons who knew the answers were a computer center director who had received queries and complaints about exactly this situation and two persons with extensive experience in PL/I.

My point is: how are we ever going to talk about errors when the people we must communicate with differ so greatly in their experience and technical vocabulary? I have tried to show that this error is not just a little PL/I anomaly, but that its discussion brings to bear a vast range of computing areas: programming knowledge, e.g. search algorithms and how to construct multi-exit loops; hardware knowledge, e.g. addressing schemes; correctness notions, e.g. proper termination; and programming language knowledge, e.g. the conditional execution of logical operators. In other words, it is impossible to talk about errors in isolation; the subject of errors pervades computing.

Since we have isolated definition as an important problem in discussing errors, it might be worth considering briefly definition as a general probelm of communication. Reference (4) Words and Ideas: A Handbook for College Writing, has a chapter on definition. A formal definition is shown to be of the form "x is a member of class y with the differentiating characteristics z". For example, "an autobiography is the story of a person's life written by himself." Several shortcomings of definitions are enumerated:

a. The defining class may be too inclusive to help narrow down a general area
b. The differentiating characteristics may use the term being defined

c. The differentiating characteristics may not adequately differentiate between things that are similar or closely related
d. A definition may be too restrictive
e. Instead of enumerating characteristics of the term being defined, a definition may offer a synonym
f. A definition may be only a partial description or a tangential observation.

The author of this handbook offers some other advice: (1) Many words are defined sufficiently by the context of their usage and (2) definitions depend for reinforcement on comparison and contrast.

If we accept this brief discussion of definitions as useful, then we can observe the important points that classification is also definition and that definitions are often not adequate to fully explain the meaning of a term.

Consider the following classification of errors that is often seen:

" An error is either syntactic or logical."

Given an error that we want to classify, we first have to define the terms "syntactic" and "logical". Each has a large number of meanings in common and technical usage. We might try to define the terms separately or together. If we observe that the context is classification, then we can apply our classification formalisation to try and ascertain the meanings. It would seem most likely that the persons who proposed this classification meant it to be exclusive and inclusive over program errors (whatever they are). This would mean that syntactic=non-logical and logical=non-syntactic. This assumption leaves a choice of terms to define. Syntactic is most often defined operationally "a syntactic error is one caught by a compiler" using the reasoning that compilers do syntactic analysis of programs for the purpose of translating the program. Of course, compilers can differ in the extent to which they catch errors, e.g. some compilers detect uninitialized variables. "Logical" is so vague a term to me that it is meaningless but "non-syntactic", I understand.

What we are saying is that the technical terms we use everyday in our work are almost completely without standard usage. This means that individuals come to understand terms and groups reconcile their differing use of terms by context and by example. Context can often be in the form of classification and indeed that is often the reason why definition becomes necessary. Put another way, we might expect to get very different answers if we pose the

two different questions:

Is x a y?

and

Is x a y or a z?

Is there some explicit way of defining errors? It seems to me that the answer is "yes there are many formal models of computation or aspects of computation, but the complexity of many of these models may make their use for definition impossible". For example, there is a formal definition of PL/I (5) written in terms of an abstract machine which interprets PL/I programs. The abstract machine is nondeterministic state-transition oriented with additional features which assist in describing programming language concepts, e.g. the environments from which identifiers acquire values and attributes. But this definition is regarded by many people as almost impossible to understand. It comes in several volumes which together are several inches thick. Part of this reflects the nature of the definitional mechanism and how well it is adapted to express PL/I but it also reflects the structure of the language itself, e.g. having to deal with numerous special cases which are inconsistently defined. If we were to use this definition on the above example, we would probably locate abstract machine instructions which show how & is defined and how KEY (I) is evaluated. If so, we could say that the error was associated with one or both of these instructions. Such a definition would lay out the spectrum of PL/I errors but its complexity might be so great that it might be almost impossible to trace down a specific error to the place where it is covered.

So, if programming languages do not have formal definitions suitable for defining errors, are there at least partial solutions to the definition problem? I believe so and will illustrate these in several places in section 5.

Let us come back to the psychology of errors, a subject which simply cannot be ignored. Software errors are caused by people, but whether they are attributed to individuals or not is another matter. Most classifications have not detailed individuals. The study to be discussed in the next section does name names. The point is whether errors can be abstracted completely away from the people who make them. Most people would want that protection, but we will argue that it may not be the best idea. If errors are brought out into the open, it may well be possible to learn more from them.

The study of errors is a strange pursuit. Although errors pervade our everyday life, it is hard to find studies which directly attack errors. A few exceptions are psychologists who study short-term memory or the Freudian slip and historians who study the mistakes of U.S. presidents. The study of errors is considered negative, perverted, and pessimistic. I can testify that the study of errors does affect one psychologically and does affect one's relations with colleagues. As I became interested in errors and took obvious delight in finding an error which reinforced some theory I had as to causes of errors, I could see other people "clam up" in fear that they would be the next victim. I was not interested in destroying their self-image and I had seen enough errors that one more was not going to adversely affect my image of them, but they did not know that. The point is that objective study of errors is hard, if not impossible. It is necessary to view errors as a phenomenon of programming which requires study and, while it is necessary to be sensitive to peoples' reactions when threatened by exposure of errors, it may be healthier to get the errors and the errants out in the open rather than to cover up the human origin of errors.

4. ERRORS IN PUBLISHED PAPERS ABOUT PROGRAMMING METHODOLOGIES

The appendix to this report is a paper which is to be published in IEEE Transactions on Software Engineering in September, 1976. It enumerates and analyzes errors which have occurred in 12 programs or classes of programs in 18 published papers or unpublished theses. The purpose of the paper was to point out these errors and then draw some conclusions about and make some recommendations for improvement in some of the modern programming methodologies. In order to fully understand the content of the rest of this section, it will be necessary to read the paper, but it should still be possible to understand the discussion of classification without reading the paper. We will summarize the errors for reference throughout the rest of this section.

S1, S2, and S3 are errors in specifications, where the specifications do not fully capture the informal purpose of the program and therefore leave open the possibility that the specifications could be satisfied by a program which did not do what was actually intended

S4 is a collection of data structure problems for which adequate specification techniques do not exist and therefore arguments arise as to whether programs are correct

T1 is a simple program to generate certain types of sequences which has a low-level coding error

T2 is a line formatting program which had numerous errors

P2 is an improvement of the line formatter with a proof of correctness, but the improvement has errors not caught by the proof

T3 is basically a word counting problem for which the specifications are inadequate and there are numerous difficulties with the programs. Its history is that authors of one paper detailed how their top-down construction failed and the error was detected and a follow-up paper by another author systematically constructed the program but with more errors.

T4 is a high-level machine language program for sorting which had an initialization error

T5 is the well-known 8-queens program which was incompletely constructed such that any completion of the program led to difficulties

P1 is the linear search program discussed in the last section

P3 is an adaptation of a program into a language currently under development where the program had an error undetected by a proof

P4 is an instance where the specifications and program are each correct but the refinement process went astray

When the study was undertaken, the data on errors consisted simply of the merge of errors I had found with errors found by my colleague and co-author, along with a previous write-up of only a few of the errors. It was clear that some classification scheme was necessary and the most natural one that sprang to mind was by what the errors had to say about specifications, systematic program construction, and program proofs. In terms of our previously discussed classification discipline, this is a rather poor scheme. Errors S3, T1, T3, T4, T5, P1, P3, P4 deal with one program appearing in one paper, while S1, S2, S4, and T3 deal with classes of programs considered in several papers. S1, S2, and S3 are so similar they probably should have been treated together. T2 and P2 are separate papers dealing with the same program and probably should have been classed together. In other words, our domain X was not a consistent set of objects, such as papers, programs, or kinds of errors. Looking back, I am sure that this inconsistency was the cause of some of the difficulties we had writing the paper and that this is a source of confusion to readers, also. Being more precise, our classification scheme was based on the three methodologies: specifications, systematic program construction, and program proofs. Let us abbreviate these S,T, and P, respectively. The classification scheme used in the paper uses the class-defining property form "the main point of error x is with respect to methodology i" where i is one of S,T, P. It was important that the scheme be full and inclusive since we needed to group these known errors in some way, but the classification is artifically exclusive. If we look at the class-defining properties as instead "x says something about i" where again i is one of S,T, and P, we get the much better class system

S: S1,S2,S3,S4,T2,T3,P2

T: T1,T2,T3,T4,T5,P1,P3,P4

P: P1,P2,P3,P4

This is not exclusive which is good because it hints at the inter-relationships of the three areas. This suggests that we might want to look at which errors say something about just one, two of the three, or all three areas. Again illustrating our classification discipline, we investigate what mathematical structure answers this type of question. Consider first the product C x C:

{S&S, S&T, S&P, T&S, T&S, T&T, T&P, P&S, P&T, P&P}

This doesn't work exactly right because S&S contains S&T and S&P and we want to directly construct a class with S&¬T&¬P. Nor does refinement work for the same reason. The other choice is extension, especially a complete

extension, where we use a truth table to show the different combinations

S	T	P	
T	T	T	
T	T	F	T2, T3
T	F	T	P2
T	F	F	S1, S2, S3, S4
F	T	T	P1, P3, P4
F	T	F	T1, T4, T5
F	F	T	
F	F	F	

Note That if P2 and T2 were treated as one, this would appear in the first line as the only comprehensive error.

By the way, this example suggests another theorem: the complete extension of an inclusive classification scheme is not full.

A natural question is whether this scheme and the resulting class system really means anything. My answer is "not much" because in reality all three areas are intertwined: a systematic construction or a proof is dependent on specifications and so on. What the class system displays is more what the papers containing the errors discussed and what we chose to add or delete from that discussion in our paper. However, it does suggest an interesting classification of a large set of papers on programming methodology. I suspect we would find that most systematic construction papers would have no formal specifications while proof papers would and this raises the question of correctness evaluation of systematically constructed programs.

ERROR	CORRECTED BY CHANGING		PREVENTED BY PROGRAM STRUCTURING		COULD BE DETECTED BY PROGRAM	
	SPECS	PROGRAM	YES	NO	PROOF	TESTING
S1	X				X	
S2	X				X	
S3	X				X	
S4	X				X	
T1		X		X	X	X
T2	X	X	X	X	X	X
T3	X	X		X	X	X
T4		X			X	X
T5		X	X		X	X
P1		X		X	X	X
P2	X	X	X	X	X	X
P3		X		X	X	X
P4					X	

TABLE 1

Table 1 shows three other classification schemes for this set of errors with an x indicating that the class-defining property heading the column is satisfied by the error. Let us consider the meaning and implication of these classification schemes in more depth:

a. Supposing we wanted to make a correction for the error, would it be in the program or in the specifications? It is hard to have definite answers to these questions. The four errors S1-S4 did not appear to affect the program. T2, T3, and P2 show necessary corrections in both program and specifications. In fact, the specifications are so vaguely stated it is possible to change them to cover up the program errors. P4 simply reflects the fact that both specifications and program are correct, but the error was in the refinement process. The classification also should show that in some articles there are no specifications to be corrected. If we were to continue this classification scheme to a larger set of errors, we would certainly use the complete extension of the class-defining properties {error in program, error in specifications}. Thinking about a classification scheme such as this abstractly, apart from this set of errors, we might be tempted to dismiss the possibility of an error in neither specifications nor program, but error P4 confirms this possibility. In other words, this set of errors demonstrates that the complete extension is a very reasonable classification scheme.

b. Suppose we ask which errors could be prevented by full use of program structuring, i.e. the principles of goto-less programming and data structuring. T2 and P2 are in both "yes" and "no" columns because there are several errors in the program. One of those errors, an infinite loop, could have been prevented by using a <u>while</u> construct rather than a <u>goto</u>. Error T5 is put in the "yes" column because the program was never actually completed, but if it were, structuring might have shown the error. The errors in the "no" column were in programs that were well-structured but where it didn't help. But remember that this is a classification of errors which indicate fallibility of modern programming methodologies. If we ask which errors could have been prevented by the full use of program structuring in a wider sense, e.g. since the paths are so clearly shown in a well-structured program it is possible to check

out the computations along these paths, then errors T4, P1, and P3 should move over into the "yes" column. That is, program structuring made these errors so easy to see that they should have been detected; that they were not is another matter.

c. Consider whether the errors could have been detected by proving and by testing. The specification errors could be detected by proving because the act of making up assertions in a proof is like making up specifications for little parts of the program and the redundancy could find the error. Or it might be that the proof shows the program could do more than required by the specifications and therefore inadequacy of the specifications becomes obvious. However, testing is less likely to show up the specification errors because the program output would simply be checked with the specifications, The specifications would probably not undergo further analysis since the programs are correct and do satisfy the specifications. All the remaining errors, except P4, are claimed to be detectable by both proving and testing. Closer analysis of the errors shows that they are easily detected by testing whereas proofs and systematic constructions failed. Error P4 is a failure of refinement and of proof but the proof could have caught the failure in the refinement process.

These various classification schemes suggest that there may be a number of standard classification schemes when dealing with errors in programming methodologies. The complete extension of the specification-program class-defining properties is a good starting point. Specification errors can be further refined into consistency, completeness, and definiteness, where our observed errors are of completeness and definiteness and consistency refers to whether the specifications can be satisfied. Top-down, stepwise refinement, and systematic construction failures are much harder to classify because they are so imprecisely defined. Errors in proofs have a nice classification scheme based on the definition of correctness as proper termination (which divides into looping and blowing up) and terminating with a correct result. Further classification comes from looking at the cases where programs and assertions are wrong. One of the purposes of this study was to come up with recommendations that would prevent such errors from re-occurring. The paper shows that we were reasonably successful at this. For example, for incompleteness of specifications we devised a test for

specifications: see if you can find an absurd program which satisfies the specifications as written but not the intent of the specifications. Another was based on the observation that several proof failures dealt with termination, which is usually considered quite easy to prove; the recommendation was simply not to ignore proof of termination. In several cases, our recommendations are on the order of "beware, here is a dangerous spot where the necessary formal techniques have not yet been developed, therefore be especially careful in your informal work". In other words, by looking closely at individual errors, we claim that we are able to produce a sufficiently deep level of understanding of the errors and sufficiently precise recommendations that such errors can be avoided in the future by us, by other authors, by reviewers, and by programmers. The purpose of this comment is to contrast the type of gains made by studying individual errors as opposed to collection of errors. Our conclusion is that the study of individual errors produces immediate gains in understanding and recommendations. Nevertheless, the observation of clusters of errors of the same type reinforces the value of the recommendations and increases the insight.

Finally, we must try to draw some higher level conclusions from this study. An obvious question is: Are these errors typical? The errors were made largely by academics using the traditional mode of academic publishing and in articles that were largely pedagogical or experimental in nature. There is no denying that pressure on academics to publish creates haste and mistakes; the same goes for the reviewers of these articles. But still, these papers have been read by non-academic researchers and developers and probably by a substantial number of programmers. Few of these errors were known before this study and there are no published corrections either by the authors or by readers writing in to the journals. So we must conclude that the errors are not just "academic bungling", that lots of different kinds of people were "taken in" by the errors. We conclude that there must be some kind of mystique which surrounds these articles that lets the errors slip by unnoticed and that our paper should certainly alter, if not destroy, that mystique. However, we must also conclude that if these errors were made in programs which were intended to be used, not just to illustrate a methodology, they would have been caught by testing.

It is also interesting to study the reactions of other readers of this paper with respect to the psychology of errors. One prominent computer scientist

described this study as "morbid, dissecting cadavers rather than devising new and better treatments", but we claim that pathological study does lead to positive measures. Another responded that "this study shows the goofs of the 'best and the brightest' and if that doesn't demonstrate the fallibility of human nature, nothing will." Of course, errors by highly respected individuals may discourage more ordinary programmers from attempting to use the methodologies, but it may also challenge them to top the experts and dispell the stigma of making mistakes. Yet another prominent computer scientist replied simply that "the price of carelessness is embarassment." The only comparable study is that of Kernighan and Plauger in The Elements of Programming Style (6) where they show errors and improvements in programs published in elementary programming textbooks. They chose to protect the anonymity of the authors of those textbooks. That was not possible for us, since it was so important that the errors be seen in the contexts of a single article and of developing methodologies.

Overall this study has not left me pessimistic. It only confirms my suspicion that programming is very hard and that this difficulty leads us to grasp at straws. The study reveals many errors in proofs, but that does not detract from proving as a methodology since testing is also fallible, but its ways of failing have barely been studied. Overall, there are signs of improvement in the design and construction of good programs, but freedom from error is not yet possible. Further study of the prevention-detection aspects of the methodologies is called for and we propose such a study in a later section.

5. EXAMPLES OF CLASSIFICATION FROM THE LITERATURE

There have been several previous studies of errors which required classification or which shed light on the nature of errors. Our purpose in this section is to analyze these studies based on the previous discussions of classification and of errors.

"A study of high-level language features" (2) is an attempt to identify language features and then evaluate them in the context of the design of tactical languages for the Army. The design goals are relevant to any language which requires capabilities for numerical calculations, process handling, and input/output. A language feature is considered to be a very small facet of a language. In this study there are over 1100 features grouped into declaration and storage management, scalar data types and operations, aggregate data types and operations, control structure, and program development aids. The features are evaluated on factors describing properties of programs (efficiency, reliability, understandability, modifiability, reusability, brevity), factors describing properties of notations (naturalness, uniformity, brevity, usability), and factors characterizing a problem domain (application dependence). Each of the factors is further subdivided until there are a total of 33 factors for evaluation of the 1100 features.

With respect to reliability, the factors are error prevention, error detection (compile-time), testability (run-time error detection), and clerical error reduction. Several examples of the types of errors which can be prevented, detected, and reduced are given. We have seen some examples of our own in sections 3 and 4. The PL/I & operator might be evaluated negatively toward error prevention and detection. The error T3 in section 4 occurs in the comparison of two sequences A and B for inequality, i.e. difference in at least one element. The error occurred while setting up a loop for this comparison in an ALGOL-like language. In another language, APL, this operation would be written as a single expression, $V/A \neq B$, not requiring a loop and thus APL might be said to prevent that error.

There are several points about this study which are relevant to our purposes:

1. It is an example of a monstrous classification problem, classifying language features into groups, evaluation factors into groups, and then the evaluation for each factor and feature. The evaluation chart is quite sparse, i.e. the authors were able to evaluate only a small number of

selected features within the scope of their project.

2. It provides a very good framework for a more extensive study of just the reliability factors of languages. For example, given a specific language with the task of evaluating its influence on errors, we might take from the list of 1100 features those which apply to the language. Using the principles of evaluation and the factors which had already been evaluated, we might perform a thorough analysis of each of the features in the language at hand. For those that rated negatively, i.e. did not facilitate prevention or detection of errors, we might forbid their use or devise specific techniques against the associated errors, e.g. conventions or restrictions on their use, specific testing or reading procedures. For features rated positively, we might encourage their use and make sure their error prevention and detection capabilities are used to the fullest. The evaluations however might be somewhat subjective, but this only suggests that the positive and negative subjective evaluations be taken as hypotheses for experiments and data collection as to how the language is used and what errors do occur.

3. Given the complete rating of features in a language, it might be possible to calculate a reliability figure for programs written in the language, e.g. based on the number of poorly rated factors used in the program. Intuitively, a program which uses only good features seems more likely to be reliable than one which uses many poor features, at least with respect to errors associated with language features. But of course this leaves out the measures that might be taken to offset the effects of poor features and the possibility that good features are not fully utilized. In addition, it seems that if the poor features are known it should be possible to eliminate all errors related to the language.

4. One set of features omitted from the list of 1100 were for process handling. A process is loosely defined as a set of actions on an environment, what is often called a task. The implication is that computer systems are composed of many tasks, with the concomitant problems of activation and deactivation, synchronization, and protection. Relatively little is provided by languages for handling processes and therefore an appendix is devoted to discussing functional requirements for process handling language features.

"An experimental analysis of program verification methods" (8) is an elaborate experiment carried out as a Ph.D. dissertation. The goal was to compare three verification methods

a. reading-a disciplined and structured desk check

b. specification testing-devising and executing test cases from specifications without access to source code

c. mixed testing-examination of the code and submission of test cases for execution

on three types of programs (each a few hundred lines long and somewhat complex) which contained known errors under conditions which were better than those usually experienced during verification (e.g. fast turn-around time, other good working conditions, and some training before starting). The results were that specification and mixed testing were about equally effective, with reading significantly inferior to both. None of the methods found much better than half the errors. Other results were that verification ability correlated highest with experience and training in programming, the distribution of time to detect the next error was uniform, and the requirement to execute every path was of little help in detecting errors. Some effort was made to determine a classification of errors by methods which worked best, but these results were rather vague.

There are several points about this study relevant to our purposes:

a. The methodology for experimenting with programming methodologies is highly complex and is not yet well developed. This study is a prototype for other studies.

b. It would be very interesting to perform a classification analysis on the errors used in these programs. The errors are generally well-described and understandable. I made a stab at this, but due to the lack of fundamental understanding of the task and of time, I did not get anywhere. The raw data from this study, i.e. the results of all the verification sessions, were preserved and it is possible that more insight could be gained into the nature of the errors detected and undetected. On the other hand, the relatively poor performance suggests that further experiments could be designed to try and improve the results of the first experiment.

c. If we take the experimental results seriously, it suggests that verification, or at least these verification methods or the conditions for verification, simply did not work well. Therefore, verification cannot be relied upon to detect and remove errors; errors must be prevented.

"A measure to support calibration and balancing of the effectiveness of software engineering tools and techniques" (9) is an attempt to evaluate a

list of about 60 existing tools and techniques as to their effectiveness on a long list of errors which occur during software development. A basic assumption is that the absence of a function is an important as the existence of an incorrect function. The overall classification scheme is requirements, design (subdivided into processing, data base, interface), construction (subdivided into processing, data base, interface, general), verification, and specification. Tools and techniques are classified as to their role in test and design. An assessment of effectiveness (high, medium, low) produces a somewhat sparse product classification. A little model for judging effectiveness is developed, but the model runs into difficulty when tools and techniques are not independent, i.e. exclusivity is a requirement.

There are several points of interest for our purposes:

a. The lists of tools and techniques is comprehensive and their effectiveness ratings provide hypotheses for further study.

b. The author distinguishes faults (causes) from errors (effects) and uses faults as the basis for his classification scheme.

c. The classification scheme, when examined in detail, illustrates all the difficulties of classification and definition previously discussed. For example,

i. The specificity of the fault descriptions ranges from "erroneous data accessing" and "incorrect resource allocation" to "recovery procedures are not implemented or are inadequate for momentary, correctable errors" and "routines are not reentrant where usage so requires" under processing of design faults. What is needed is further classification in order to make a list of 18 processing design faults more comprehensible.

ii. Many of the faults are highly interdependent. For example, "requirements missing" and lots of other aspects of requirements which are missing or inadequate all intuitively imply another class, the separate fault "requirements not testable/verifiable". As mentioned above, the need for exclusivity is dependent on the purpose of classification scheme. In this case, it appears that lack of exclusivity is a symptom of lack of structure of the classification scheme, that in fact the scheme should be given hierarchically. I am unable to revise the classification scheme into a proper hierarchy because the terms are not sufficiently concrete for me.

iii. Consider another classification scheme for data base design faults:

erroneous units

parameters in incorrect format, order, or location

erroneous values

duplicate data variables

missing data variables/values

If we isolate the terms used, we find: data variables, values (or is it data values), parameters, units, erroneous, and missing.

What does the author mean by parameters and data variables and values? These all seem like the same sort of thing. And there are several qualities of these things; missing, erroneous, misrepresented, and misplaced. All of these terms suggest that it might be possible to build a little formal model which clarifies what is going on. Suppose we consider the references to the data base as some sequence, r1, r2, ... What can go wrong? The value obtained from the data base on reference ri might be erroneous because the wrong value was placed in the data base or the value that was placed there used a different unit than that assumed in the context of the reference or the reference might somehow be going to the wrong place. The wrong value might become duplicated from another place. References might be made in the wrong order. I have difficulty building such a model, but it seems to me that people familiar with the context should be able to build such a small model and define their terms and the errors more concretely. That is, if we consider the problem of giving meaning to the class-defining properties, we can do so by appeal to a formal model as well as by context in a classification scheme.

The point is that the classification scheme appears very poorly structured and yet it seems that the structure can be assigned with further analysis of the class-defining properties. Another point is the way the classification scheme for faults was used. Several experts were asked to rate the effectiveness of tools and techniques on the faults. How did those experts understand the faults? If the classification scheme were more hierarchically organized could more have been said by the experts? How valid are the ratings the experts did give?

Studies like this one appear to me to be useful, but when one looks at the bottom line, how the errors are described, the value seems to disappear. The absence of a classification discipline and the imprecision of the terms

makes the whole effort a rather mysterious process. That is not to say that the people involved in the study did not know what they were doing, but only that an outsider does not know what they were doing. I am simply trying to point out why I have trouble understanding the study and what might have been some of the problems they encountered during the study. If error classification studies are ever to be valuable, they must be intelligible to a wide range of people and they must be reproducable. The methodology used in the study may be transferrable only down to the point where errors are actually discussed which means that the results might not be reproducible.

We can analyze other classifications from other papers in similar fashion, but before doing so, it is worth re-examining our analysis questions:

What is the structure of the classification scheme in terms of the section 2 of the present paper? Is there any clear structure? Can the structure be improved?

Do we understand each of the class-defining properties? If so, how? By context in the classification scheme, independent knowledge, context in the problem area, example? Given an error which is possible in the context of the known problem area, can we see where to place it in the classification scheme? Can we go down the classification scheme and concoct errors which might fall into each of the classes? That is, can we decide whether the classification scheme has exclusivity, inclusivity, fullness?

What is the purpose of the classification scheme? How did that influence it? Was there some hypothesis to be proved? What characteristics should the classification scheme have?

Another example of a classification scheme appears in "Toward a theory of test data selection". (10) The purpose of classification is to get at the types of errors that testing must deal with. Consider the subclassification of control flow errors:

missing control flow paths
inappropriate path selection
inappropriate or missing action

If we dissect these terms, we find the following components: decisions (selections) and actions, missing and wrong (inappropriate). This suggests a little model based on the idea of a path of a program, viewed as a sequence of decisions and actions with two things going wrong, missing

and wrong. We might represent this formally by the following table

	Action	Decision
Missing	P1,P2/P1,a,P2	P1,P2/P1,a,P3
Wrong	P1,a1,P2/P1,a2,P2	P1,d1,P2/P1,d2,P3

where P, a, d denote paths, actions, and decisions and A/B means that A and B show the form of the erroneous and correct paths.

We can go further with this model. We can add the idea of a state vector and a state-transition function and look at the series of state vectors associated with a computation, identifying errors as places where the state vector contains wrong values. Or we can define a computation function Comp (program statement, state vector) which shows how each program statement defines a new state vector. Then we can isolate points in the trace of Comp which correspond to different types of errors. At another level, we can look at the program text and correlate errors with the type of correction to be made to the text, e.g. inserting statements for missing actions, changing decisions in if-then-else and while statements for wrong decisions, etc. Of course, there are a vast number of models of computation more specific than these. It would be interesting to interject an error aspect, i.e. to define what is meant by error in each of these models and then see how much is said.

The classification schemes used in "Some experience with automated aids to the design of large-scale reliable software" (11) are generally well-structured. For example, Table 1 uses "design" and "coding" a good definition of "design" and "coding" with a good definition of "design" so that "coding" is implicitly defined as "non-design." Figure 2 uses a product kind of classification

$$\left\{ \begin{array}{l} \text{design,} \\ \text{coding} \end{array} \right\} \times \left\{ \begin{array}{l} \text{before acceptance testing,} \\ \text{during or after acceptance testing} \end{array} \right.$$

Table 2 enumerates some error categories which are applied to both design and coding errors with a classification into mostly one or the other. However, the meaning of the class-defining properties is unclear, because the error categories are so vastly different. A subclassification is apparent: interface (user, hardware, data base, software), device handling (tape, disk, card),

communication (output, error message), and computation (computation, bit manipulation, indexing and subscripting, iterative procedure). Again these are vaguely defined terms but some more thought might clarify their meaning. The table must have a typographical error or the scheme is not inclusive since the number of errors adds up to 220 but the total number of errors under consideration is 224. Table 3 is a gross classification of error causes with examples which help to clarify the meaning of the terms and the nature of the causes. It is a product type of classification. Once again in table 4, the difficulty in defining errors is apparent. It looks like a quadruple product {error type} x {software phase} x {origin-found} x {applicable tools} which is useful but somewhat hard to decipher. For the error types, it looks like there is a hierarchical classification something like behavior (first, later cards processed right), relation to storage (internal, mass), data (range, units, accuracy (range, units, values), program action (accept, reject)). It seems to me that it should be possible to devise a clear set of terms and a comprehensive classification scheme for errors like these. The data base interface classification scheme discussed earlier suggests this to be common and yet troublesome to discuss.

"An analysis of errors and their causes in system programs" (12) is an interesting study. One of the first points of interest is the remark that "although only the history of a single error (one discussed under T2 in the appendix) is described, this type of investigation promises to be the most successful." That may be interpreted as "gross collections of errors will be less successful" or "investigations of single errors will provide the most information". Another interesting point is the common-sense approach to reporting errors: Who, Where, When, Why, and How. Again it is impossible to report without classification. Who and When require information like origination, propagation and detection of errors. Where involves classification into modules and statements. What is the general problem of error classification seen so often. Why requires classifying factors, and How gets into prevention and detection methods. The overall classification scheme is well-structured but again at the lowest level, the descriptions of the errors, it becomes difficult both to understand the class-defining properties and to determine exclusivity and inclusivity. For several parts of the classification scheme, a little model of the machine language level with precise usage of terms like addressing would help.

"Types, distribution, and test and correction times for programming errors" (13) relates an extensive data collection effort. Again it is interesting to look at the wide range of descriptions of errors in Figure 3. One should ask what it takes to use the TR/CR forms. Note that these are constructed to be inclusive by using an "other" category and presumably only one box is expected to be checked. Presumably the set of terms was predefined and maybe illustrated by examples so that the originators could check the right boxes.

In summary, I have been trying to bring out some of the problems in previous classification studies by use of the classification discipline and to illustrate how that discipline can be applied. Another point is the difficulty of definition and how that seems to be either unmastered or sometimes tolerated by the use of context and classification assumptions. Finally, I have suggested that there are several places where the definition and classification problems might yield the building of small formal models.

This discussion is not intended to criticize authors of the papers for something that they did not intend to do. Most of the studies are tentative and the classification schemes were never intended to live up to the standards we have applied to them. Nevertheless, I claim that had a classification discipline been followed and had more attention been paid to definitions, the results might have been stronger. Viewed as experiments, we would like the reporting to be such that the conclusions can be evaluated for how well they follow from the data, but when we get down to the lowest level of defining errors we find the imprecision troublesome. We would also like these experiments to be repeatable, and this seems unlikely unless the definition problem is solved and the same classification schemes apply in different situations. The class-defining properties were constructed bottom-up by generalizing from instances of errors. My claim is that this doesn't work since the classification schemes are ill-defined and unconvincing.

6. TWO ABORTED STUDIES OF ERRORS IN SPECIFIC TASKS

As mentioned earlier, the proposed research included studies of the errors in two specific tasks: synchronization of processes and queue data structure management. These studies were not very successful except in causing us to confront the difficulty of the problem and back off to attack the basic principles of classification and definition which are discussed in the rest of the paper. However, there are some more observations which are worth considering.:

a. A process is a sequence of operations carried out one at a time. Processes may be executed concurrently in a computing system, i.e. their executions may overlap in time. Concurrent processes introduce a host of problems: synchronization, whenever two process interact; deadlock,when two processes are waiting indefinitely for an event which can never occur; scheduling, so that work gets done in an orderly fashion; and protection of data of one process from the unrequired operations of another.

This is a subject where errors were recognized early as being of critical importance: the difficulty of reproducing time-dependent errors made testing as a means of verification unquestionably impossible. As a result of the error difficulty and the overall fascinating complexity of the subject, there is a vast amount of literature, perhaps best summarized in Brinch-Hansen [16]. There is considerable work on language mechanisms to facilitate error prevention and detection and mechanisms whose behavior can be provably determined and therefore can be adopted as conventions.

At an abstract level, these problems are well handled, although it is not yet totally understood. While errors may be well understood abstractly, there are few examples of actual errors in the literature and this tends to leave one hand hanging in limbo unless one has studied the problem for years and developed the necessary intuition. I have found this subject particularly difficult to understand and could not find a point at which to attack the problem. It seems that there is something different about this problem than just its complexity. Since the errors are so important, the biggest ammunition has been trained on them and, at least abstractly, adequate mechanisms have been devised for preventing these errors. At least, the difficulty is well enough recognized that extreme care is taken. However, that is not to say that the mechanisms are easy to use or that they are used reliably in practice. It is at this concrete, mundane level that there is no data on errors: What mistakes do programmers usually make? I believe that this is a fruitful area for study, but that it must be undertaken by people more experienced in the area.

b. A queue is a data structure which has a first-in-first-out behavior, (17). The usual operations are insertion and removal and, perhaps, scanning the elements stored in the queue. For example, letting I n denote "insert n into the queue" and R mean "remove the next from the queue," with the following sequence of operations, the queue would have the contents:

	Queue	Element removed
I 1	1	
R	Empty	1
R		Error
I 2	2	
I 3	2,3	
R	3	2
I 4	3,4	
I 5	3,4,5	
R	4,5	3

The programs are short and easily understood. Possible implementations are (1) store the queue sequentially in a fixed sequence of locations and treat these circularly, i.e. let the queue wrap around from the end; (2) sequentially as in (1), except let the queue drift up to the end of storage, then move it back to the front section; and (3) link the elements through some free area of storage. In any case, one constant problem is finite storage, although in (1) and (2) the sizes of the queues are fixed and in (3) they vary with the amount of storage available. There are phenomena which we might call "overflow", too many elements tried to put in the queue, and "underflow", calling for removal when there are no elements in the queue. This suggests a classification of calls on the queue handler by the program using

C1 = {normal, underflow, overflow}

Underflow and overflow might be considered as errors, not of the queue handler, but of the calling program. However, there are possible errors by the queue handler associated with calls. Suppose we construct a product kind of classification with the class-defining property:"the call is i, but the queue handler reports j" where i and j are from C1. Now we have a precise description of some of the possible deviations from accuracy. Another classification scheme applies to the removed elements (call this the output) relative to the inserted elements (call this the input).

c="output elements are the same as input elements."

c defines errors where somehow the queue handler either inputs or outputs

wrong. c may be further refined into where elements are removed are
{lost, duplicated, or reordered}
in the output.

What are we defining here? Our reference sets are calls and input-output lists of elements and we are using the classification discipline to generate classification schemes which we then re-interpret in terms of errors. So what we are doing is focusing on aspects of behavior and then working into defining classes of errors. Now, suppose we look at an actual program which we believe to be correct and inject errors into it. We might do so by systematically changing operators, say + to -, or identifier names; deleting statements, or otherwise altering the flow of control; and modifying various language aspects, such as changing declarations from integer to real. We might want to see how the effects of each of these changes can be classified according to the above. We might ask: does the error manifest itself as an "incorrect report to a class call" or as "an incorrect output" or "both" or "neither"? Neither" is quite possible since the program is likely to blow up. "Both" is also possible. One problem is whether the effects are measured relative to a fixed input stream or to any conceivable input stream. The latter is of more interest, but the former can be experimented with. We can go on classifying into immediate vs. delayed effects relative to the pattern of calls on the queue handler, detection by testing and proof, and prevention by language features and conventions.

Our initial studies of this problem task were unsuccessful since we had only the vaguest notion of the classification task and because error injection was unenlightening. The programs are so simple and well understood that errors were silly and we had no trouble devising assertions and test cases which would immediately detect the errors. Furthermore, the injection of errors is very tedious. However, that is not to say that any real insight was gained. We had only the grossest classification scheme and little understanding as to why errors manifested certain behavior or why they were so easily detected. We also had difficulty finding "devious" errors, ones that would be hard to detect and prevent.

After development of the classification discipline, it now seems like a more feasible project to study the errors that could be injected into this program. However, such a project would still take several days of work. But its purpose is unclear, since natural errors are so much more interesting than artifical ones. We did not have the time or sufficiently defined purpose to carry on the study after development of the classification discipline.

7. SUMMARY

We took apart the problem of classification of errors. Looking at classification abstractly, we devised some mathematical definitions and operations which seemed to characterize the intuition used in classification activities. We then turned to the subject of errors and showed that our vocabulary is all-important and that it is very difficult to express what we need to say about errors. A kind of circularity arose: to discuss errors we needed a classification scheme but to develop a classification scheme we needed definitions of errors. We tried to find our way out of this circularity by considering definitions of errors relative to formal models and definition of errors by adjustment of their meanings to fit classification notions, i.e. by context.

A number of examples of classifications were studied. The study of observed errors on the fallibility of modern programming methodologies suffered from an inconsistent error domain which caused several types of classification schemes to be difficult to construct and to interpret. Several papers from the literature had classification schemes with deficiencies which we could diagnose in terms of our classification discipline. The overall pattern there seems to be: given errors u,v,w,x,y,z, generalize to say that u,v,w are a's; x,y,z are b's; and a's and b's are c's. At the top level, we may agree that a's and b's are c's but a and b describe different kinds of properties of the errors. That is, the criteria for classifying the errors, as expressed by the class-defining properties, are based on different facets of the errors. This suggests that the errors could be better classified, but for the purposes of the studies, it was sufficient to achieve classification from the c level up. Our concern was the intelligibility and reproducability. of the studies when the class-defining properties are ill-defined.

The problem when considering an individual error is that we want to say "error x is an a,b,c, ..." where a,b,c express specific aspects of different facets of x. But then we need a classification of the facets and a further classification of the aspects and of course this is a classification problem.

In our classification study the central point is seen as the class-defining property. In our error study, this translates into defining errors. We claim that the problem must be approached top-down by defining terms within the context of classification, bottom-up by testing the terms on example errors and by generalization, and sideways by building formal models which more precisely define the terms.

8. RECOMMENDATIONS

a. Explore the classification terminology and discipline further.

It requires polishing of terminology and there are certainly more definitions and theorems which would be useful. The classification concepts are really quite simple and therefore it should be easy to teach them with some examples appropriate to the students at hand. However, its simplicity could cause the whole subject to be ignored. The terms and theorems are so obviously just a formalization of intuition and normal practice that it is tempting to simply bypass them and go on with normal practice. However, our studies of the literature have shown that, in fact, classifications often turn out confused, perhaps because intuition gets overwhelmed with detail. Our claim is that the mathematical formalization of intuition leads to a discipline which thereafter guides intuition to better and more easily achieve structures. The classification material may be further developed in two ways. One is simply abstractly building on what is there and the other is by applying it to several examples and extending and adjusting the definitions and theorems until it fully explains these examples.

b. Develop criteria for good classification schemes.

We have put forth two criteria

I. A classification scheme should have a clean, immediately discernible structure expressible in our classification terminology.

II. It should be decidable whether a classification scheme has characteristics such as exclusivity, inclusivity, fullness.

We have also claimed that whether a classification scheme has these properties depends on its intended use. Therefore, there should be additional criteria specialized to usage.

c. Develop an exemplary classification scheme.

The size of the classification scheme need not be large. Its purpose is to show the difficulties and benefits of a good scheme. We have suggested something like the data accessing which appeared in two classification schemes discussed in section 5. The idea is to get at all the different things that can go wrong whenever a program calls for data, in both device-independent and dependent terms. Another possibility is addressing errors at the machine language level. Both of these can be tied to little formal models which should be developed to test out and refine the classification.

d. Develop an exemplary set of errors.

Explore possible statements about the errors: what should be said and how it should be said. Experiment with different individuals with different orientations and how they percieve and express the errors and how they understand and react to perceptions and expressions of other orientations. The purpose would be to obtain a complete analysis of a set of errors.

e. Decide whether to pursue errors individually or in collections.

It may well be more fruitful, at least initially, to study errors individually. For example, suppose we take a single error and trace back all the ways it could have been prevented and can be detected. This might well provide a large number of specific recommendations that could then be generalized as the same process is repeated with other errors. The alternative approach of considering collections of errors might be less fruitful simply because there is too much to consider at one time and/or classification does not help. In other words, iterating on a set of data may be better than handling them in parallel. However, the collection of error data probably would point to the best errors to start with.

f. If error data is going to be collected, this should be done under some firm hypothesis. (15)

The problem is that there is so much data that can possibly be collected that it is necessary to select in some fashion. A good selection procedure is to adopt a relevant hypothesis. For example, Endres (12) observes that many errors occur in "understanding the problem" which suggests collecting data which traces all errors sufficiently far back that it can be determined whether they are of this nature and also suggests predefining classifications under this general rubric so that the errors can be meaningfully classified as they occur. As another example, there is a vague feeling that structured programming prevents errors, but there is little real data on what types of errors are prevented. Perhaps the reason errors are prevented, if they are, has nothing to do with the actual techniques, but instead is due to increased carefulness or improved expectations. This might lead to collection of error data which asks about the psychological reactions of persons associated with the error to it, as well as trying to pin down the exact cause of the error.

g. Interject error processes into formal models or re-examine formal models from the standpoint of errors.

We mentioned the PL/I formal definition in section 4. While this formal model of PL/I program execution is probably too complex, it might be interesting to enumerate all it says about errors. We have also suggested that

it might be possible, and it certainly would be useful, to build some small specialized models of situations where errors occur, e.g, data accessing. The goal would be to abstract away from languages, devices, and problems, to get at the nature of data accessing errors and obtain a general classification. That model might then be specialized for languages, devices and problems with added clarity.

h. Construct probabilistic models based on the classification mathematics. Section 2 provides a way of classifying classification schemes. This seems like a good place to start building up a probabilistic structure. What type of model corresponds to each kind of classification scheme? What kinds of questions does each scheme suggest and answer? How much does the model depend on the class-defining properties and their characteristics? What kinds of schemes are best for developing probabilistic structures?

It seems unlikely that reliability assessment methods can ever be devised unless they have some abstract structure related to the classification structure.

i. Continue some of the studies described in section 5. We suggested that some of the classifications given there could be redone using our classification discipline. There are more specific continuations.

I. Helzel's errors could be classified and his data used to explore them.

II. The language features in the Goodenough study could be specialized to a single language, evaluated, and studied with respect to actual errors.

III. The effectiveness of tools and techniques is a promising way of getting at immediate gains. A much simpler study might be to propose three modes of verification:

reading - superficial analysis for the purpose of finding gross errors or inconsistencies,

testing-case analysis with detailed exploration at the case level

proving-statement of conjectures followed by deductive reasoning, either mathematical or argumentative

and three types of programming activities

specification, design, and construction of programs

and apply all three modes to all three activities, trying to determine what types of errors are differently detected. Another question is what type of errors are undetected in one activity, but detected by later activities and how.

However, remarkably little is known about how well testing works for different types of errors. Both formal and experimental analyses are called for. A few examples are seen in references (10) and (14). It would seem best to cast a critical eye on the most prevalent activity before looking at the others.

REFERENCES

1. Stoll, R. Sets, Logic, and Axiomatic Theories, Freeman, 1961 .
2. Hartigan, J.A. Clustering Algorithms, Wiley Interscience, 1975.
3. American College Dictionary, 1966.
4. Guth, H. Words and Ideas: A Handbook for College Writing, Wadsworth, 1961.
5. PL/I definition, Technical Report Series, IBM Labatories, Vienna, Austria.
6. Kernighen, B. and Plauger Elements of Programming Style, McGraw-Hill, 1974.
7. Goodenough, J. and Shaffer, L. A Study of High Level Language Features, ECOM-75-0373-F (2 volumes), February, 1976.
8. Hetzel, W. "An analysis of methods of program verification", Ph.D. dissertation, University of North Carolina, April, 1976.
9. Curry, R. "A measure to support calibration and balancing of the effectiveness of software engineering tools and techniques," MRI Symposium on Software Engineering, April, 1976.
10. Goodenough, J. and Gerhart, S. Toward a theory of test data selection", IEEETSE, June, 1975
11. Boehm, B., McClean, R., and Urfrig, D. "Some experience with Automated Aids to the Design of Large-Scale Reliable Software", Proc. Intl Conf. on Reliable Software, April, 1975.
12. Endres, A. "An analysis of Software Errors and their Causes", Proc. Intl. Conf. on Reliable Software, April, 1975.
13. Shooman, M.L. and Golsky, M.I. "Types, Distribution, and Test and Correction Times for Programming Errors", Proceedings Intl. Conf. on Reliable Software, April, 1975.
14. Howden, W. "Reliability of the path testing strategy", IEEETSE, September, 1976.
15. Goodenough, J. Personal Communication, May, 1976
16. Brinch-Hansen, P. Operating Systems Principles, Prentice-Hall, 1973
17. Knuth, D. The Art of Computer Programming, Vol. I.

APPENDIX

OBSERVATIONS OF FALLIBILITY IN APPLICATIONS OF MODERN PROGRAMMING METHODOLOGIES

Susan L. Gerhart*
Computer Science Department
Duke University
Durham, NC 27706

and

Lawrence Yelowitz
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

May 1976

Keywords: Program correctness, program specifications, programming methodologies, program errors, correctness proofs, testing, reliability, fallibility.

*This work was partially supported by NASA Grant NSG 1267 and NSF Grant MCS 75-08146. The views and conclusions expressed in this paper are those of the authors and do not necessarily reflect those of the granting agencies.

To appear in IEEE Transactions on Software Engineering, September, 1976.

ABSTRACT

Errors, inconsistencies, or confusing points are noted in a variety of published algorithms, many of which are being used as examples in formulating or teaching principles of such modern programming methodologies as formal specification, systematic construction, and correctness proving. Common properties of these points of contention are abstracted. These properties are then used to pinpoint possible causes of the errors and to formulate general guidelines which might help to avoid future errors. The common characteristic of mathematical rigor and reasoning in these examples is noted, leading to some discussion about fallibility in mathematics, and its relationship to fallibility in these programming methodologies. The overriding goal is to cast a more realistic perspective on the methodologies, particularly with respect to older methodologies, such as testing, and to provide constructive recommendations for their improvement.

1. INTRODUCTION

It is well-known that programming is an error-prone process. As a result, the last decade has seen the development of new programming methodologies aimed at reducing the frequency and severity of errors during the programming process. Briefly, we might label some of these newer methodologies:

Formal specification - Expression of program requirements in unambiguous and complete terms;

Program structuring - Use of a restricted set of reliable control and data structures;

Systematic construction - Development of programs through successive refinements where correctness is argued informally based on the simplicity of each step;

Program proving - Development and use of mathematical systems for presenting proofs of program correctness.

Common to all these methodologies is the application of mathematical reasoning to programming, the goal being a sufficiently high level of mathematical rigor so that errors will occur infrequently and be easily detected when they do occur.

In this paper, we show that the new programming methodologies are still quite fallible. Our approach is, in part, to point out errors, inconsistencies, or confusing points in a variety of published algorithms. Many of these algorithms are being used as examples in teaching new programming methodologies, and it is important for such points of contention to be discussed openly. We go beyond merely listing the points of contention by trying to abstract common properties of them. These common properties are used to help us conjecture some "reasons" for the errors

47

and to assist us in formulating general guidelines which might prevent reoccurrence of the errors. Our goal is to cast a more realistic perspective on the methodologies and to make constructive recommendations to improve them.

The errors are classified as:

1. Specification Errors, where something is wrong with the specifications for a program, making the programming and verification process fallacious;
2. Systematic Construction Errors, where errors contaminate the process by which a program is developed and the resulting programs are incorrect;
3. Proved Program Errors, where errors remain undetected even though a "proof" has been given.

This tripartite categorization is largely a matter of convenience in exposition and should not be construed too rigidly. Several of the errors are, in fact, discussed in relation to more than one of the above categories.

In the next three sections the error categories are discussed individually. Each section begins with a short introduction, followed by a listing of the points of contention, followed by a conclusion which generalizes over the errors of each class and provides recommendations for preventing and detecting errors of the class. Each point of contention is presented as a miniature case study, in which the following are described: the context of the algorithm in relation to the publication; a description of the algorithm and its point of contention; and an analysis. Section 5 presents conclusions and recommendations generalized from the three error classes. In Section 6, we discuss some relations between fallibility in programming methodologies and fallibility in mathematics. The errors are listed in a Table for reference.

48

We realize that this paper deals with a sensitive subject and that the material can be interpreted in many ways. Therefore, we wish to present our views of the subject and to repeat our purpose:

1. We did not search for these errors; once we became aware of the potential for error and developed some intuition about causes and effects of errors, the observations appeared naturally in the course of our normal reading and research. This is discouraging in that it signals a lack of awareness and/or critical reading on the part of reviewers, but at the same time it is encouraging in that it shows that errors can be identified once awareness and critical reading skills have been developed.

2. We are convinced that the errors do not destroy credibility of the modern programming methodologies. Perhaps we should split the purpose of the methodologies into design and verification.

   a.) Design has been continually emphasized in the programming methodology literature. The programs mentioned in this paper are, for the most part, well-designed; that is, even though errors are present, the programs are substantially correct. There should be little doubt of the value of the methodologies for design.

   b.) It is at the verification level that the methodology failures have been observed. As mentioned above, the programs are substantially correct through good design but still contain errors, most of which are minor and easily fixed. However, even minor errors can have serious consequences and be costly to fix. One of the most serious consequences is to cast doubt on the usefulness of the methodologies.

We believe that the analysis of errors and the recommendations we present can lead to prevention and early detection of most of the types of error that so frequently occur.

3. We do not believe that the errors reflect negatively on the skills of the persons who committed them. Instead, mistakes are inherent in the difficulty of the programming task and the early stage of development of the methodologies. Each article mentioned here makes a significant contribution to that development. Often the erroneous examples are tangential to the main point of the paper. The errors may only increase that contribution, albeit in an unplanned way. If blame is to be laid anywhere, it should go to the reviewers of the papers and other readers who have missed the errors. We also believe that it is far healthier to discuss these errors openly than to ignore or cover up their existence Perhaps what we need is more "egoless publishing."

4. To some extent, we are playing the role of Monday morning quarterbacks. Many of the errors are "old" in the sense that the papers are very early and much progress has been made since their appearance. However, many of the errors have only recently been detected and the errors are still occurring in contemporary papers. This forces us to conclude that the analysis is necessary.

5. There is the additional aspect that the errors provide a way of studying the programming methodologies which yields unique insights into the processes. We have learned much about how to write specifications, assertions, and programs from our study of the literature from the unique viewpoint of errors. Perhaps, others will, also.

6. That the testing methodology is fallible is so well known that we did not attempt to include errors of this kind. Analyses of testing fallibility are presented in (17) and (31). Examples occur regularly in the algorithms section of the Communications of the ACM.

7. The common characteristic of mathematical rigor and reasoning in these examples leads one to question the effectiveness of "The Mathematical Approach", not only in Programming Methodology, but in Mathematics itself. The frequency of errors in mathematical theorems, proofs, and applications of theorems is well-recognized and documented. Mills (5) provides a cogent argument for the use of mathematics in programming, a subject we will return to in section 6.

Readers are, of course, free to draw their own conclusions about the significance of the errors and the implications about modern programming methodologies. We only ask that the material be considered in the spirit in which it is offered.

2. ERRORS IN SPECIFICATIONS

2.1 Introduction

An early stage of the program development process should involve the rigorous specification of requirements for a program in terms of expected input, required output, constraints on storage and time, and actions in response to invalid inputs or run-time storage or time limitations. In practice, it seems that such specifications are used more frequently in large multi-person software projects and are often skipped in pedagogical articles on the new programming methodologies. In articles on proving program correctness, however, at least formal input and output requirements must be specified, although errors and algorithm constraints are

5/

usually ignored. Our list of specification errors concentrates on articles on program correctness, not necessarily because errors occur more frequently here, but because there is a lack of other published material on program specification.

Liskov and Zilles (1) discuss specifications as the media which translate a concept in someone's mind of what a program should do to solve a problem into a formal written statement of exactly what the program should do. One value of this step is that it becomes possible to formally prove consistency of programs with such formal specifications. However, the complementary step of verifying that a program specification implements the underlying concept must necessarily remain informal. Most of the error observations relate to deficiencies in the concept-to-specification step. But we shall see that these errors suggest guidelines which can make this necessarily informal step more reliable.

2.2 Examples of Specification Errors

S1: Prime Test

King (2) p. 190, Wegbreit (3) p. 106, Deutsch (4)

Context: The example is used in [King] to show the power of a mechanical verifer using the inductive assertion method, and in [Wegbreit] to illustrate a mechanical assertion generator.

Description: The informal specifications are "set the flag variable J to 0 or 1 as A is or is not a prime". The formal specifications are

Input: $A \geq 2$

Output: $[J=0 \Rightarrow (\forall k)(2 \leq k < A \supset A \bmod k \neq 0)] \wedge$

$[J=1 \Rightarrow (A \bmod I=0)]$

The program is (rewritten from flow charts to text):

```
I:=2;
while (A mod I)≠0 do I:=I+1;
if I=A then J:=0 else J:=1;
```

The formal specifications are inadequate, as shown by the following programs which are equally "correct" with respect to these specifications.

(1) J:=2

(2) I:=1; J:=1

(3) I:=A; J:=1;

(4) J:=1; A:=0; I:=1

(5) J:=0; A:=1

Analysis: A source of difficulty is that neither I nor J is sufficiently constrained in the output specifications; also, A is not constrained to have the same value it had upon input. More complete output specifications are:

$[J=0 \Rightarrow (\forall k)(2 \leq k < A \supset A \bmod k \neq 0)] \wedge$

$[J=1 \Rightarrow (\exists k)(A \bmod k = 0 \wedge 2 \leq k < A)] \wedge$

$[J=0 \vee J=1] \wedge [A = A_0]$

(where A_0 denotes the input value of A).

The error might have been detected by noting that the given program can be proved without using the input specification. Such a phenomenon would probably be noticed by a person performing a hand proof but possibly not by a machine proof that was not carefully inspected.

Comparing the informal and formal specifications, we see the following inconsistencies:

a. Informally, J is to be set to either 0 or 1.
   Formally, this is omitted.

b. The condition for A being a prime is translated correctly in the implication for J=0, but the condition for A being a non-

53

prime is not. This shows a failure to abstract the notion "prime", such as

prime (A) $\triangleq$ $(\forall k)(2 \leq k < A \supset A \bmod k \neq 0) \wedge A \geq 2$

$\equiv$ 'A is greater than 1 and has no divisors except 1 and itself'.

which may be used to better express the formal specifications as

$[J=0 \wedge \text{prime}(A) \vee J=1 \wedge \sim \text{prime}(A)] \wedge [A=A_0]$

c. The informal specifications clearly refer to the input value of A, but this is not reflected in the output specifications.

S2: Sorting and Searching

King (2, p.208), Deutsch (4), Mills (5), McGowan and Kelly (6, p. 33)

Context: The examples illustrate program proving techniques.

Description: Let A be a real-valued array indexed from 1 to N, $N \geq 0$. The output specification of a sorting program is that:

$\text{Sorted}(A,A_0) \triangleq \text{Permutation}(A,A_0)$ and Ordered(A)

where Permutation formally expresses that A is a permutation of A_0 (the input value of A) and Ordered expresses that A is in (usually nondescending) order. In the examples, the Permutation conjunct is often omitted and ordering alone is used as the specification.

As pointed out by London(7) and Hoare(8), if this occurs the following program may be said to "sort A into nondecreasing order".

for I:=1 to N do A[I]:=0.

Specifications for the example searching programs usually look like:

Input: TAB(1::N) and KEY, where $1 \leq N \leq$ declared subscript limit of TAB and TAB and KEY are of compatible types.

Output: KEY = TAB(I) and $1 \leq I \leq N$ or KEY is not in TAB(1::N).

54

The output specification should require TAB, N, KEY to be the same as on entry. If this is not required the following programs may be said to "search":

(1) I:=1; TAB(I):=KEY

(2) N:=0; I:=0

Analysis: The Permutation property is messy to state and prove, especially using the inductive assertion method. A common, and certainly reasonable, proof technique is to use the fact that if the only operations performed on a vector are swaps of two elements, then the vector is always a permutation of the input vector. The Ordered property is better adapted to the inductive assertion method of proof. There is nothing wrong with splitting the proof into two parts as long as it is explicitly stated that Ordered is only part of the specification and does not correspond by itself to Sorted.

The following properties of Permutation are often used:

(i) Permutation (A,A)

(ii) Permutation (A, swap (A,I,J))

(iii) Permutation (A,B) $\wedge$ Permutation (B,C) $\supset$ Permutation (A,C)

Notice, however, that it is insufficient to formally characterize Permutation by only these facts, which are also satisfied by

Permutation (A,B) = "the sum of the elements of A= the sum of the elements of B"

However, it is still fair to use these facts within a proof.

We commonly understand that searching does not destroy the initial values of KEY, N, or TAB (1::N), although it might be that TAB(N+1) is used as a terminating value in a search loop. It is just a convention by which programmers abide when writing search algorithms. (Search and

insert algorithms are another matter, though). From the standpoint of formal specifications, however, it is hard to argue that the programs (1) and (2) are not "correct". Like the permutation property of sorting algorithms, the fact that input variables retain their values at output is more easily shown by simply inspecting the program for absence of assignment or side effects in procedures than by the cumbersome method of incorporating these statements into inductive assertions. And again this should either be adopted as a convention or explicitly stated as a separate aspect of the specifications and proof.

S3: Magic Square Generator

Gerhart (9, p. 194)

Context: Proof techniques for APL programs are illustrated.

Description: The program, written in APL, is proved correct with respect to the specifications (informally stated):

Input: $N \geq 1$ and N is an odd integer

Output: M is an N x N matrix and the sums of the rows, columns, and main diagonals of M are the same.

An equally correct program with these specifications sets every element of M equal to 0. The usual definition of a magic square adds the requirement that every element of the matrix M should be an element of the initial sequence of integers $1...N^2$, and that each element of that sequence is an element of M. Of course, N should not be changed by the program.

Analysis: Since the committer of this error is one of the present authors, we can testify that the omitted requirement was simply forgotten. The proof of a real magic square generator is difficult, using several number theoretic results, and the author simply became absorbed in that

56

proof and failed to complete the output specification. Once the example had made the point in the context of the thesis, it was considered complete. Like error S2, this implicit output condition is easily seen to be valid by inspection of the program. Nevertheless, it should be stated.

This error first came to our attention when a story was related to the authors about a computer science professor who assigned the magic square problem as a programming assignment and gave the incomplete specifications above as the problem requirements. One student submitted the program which set every element of M to 0. The furious professor was then faced with the dilemma that the program was consistent with the given specifications, but not a magic square generator. This characterizes the problems described in the last errors: specifications must be complete enough to capture the concept involved, and sufficiently constrained so as not to be satisfiable by trival programs like those we have been giving.

S4: Assertions about data structures

Oppen (10), Cook and Oppen (11), Knuth (13, Alg. 2.3.5D), Bertziss (14, Alg. 10.8)

Note: These are points of contention, not necessarily errors.

Context: Two opposite approaches to discussing data structures are taken, formal in the first two and informal in the last two references above. The purpose of the formal papers is to develop the theoretical notion of expressibility of languages for stating assertions about programs. The two well-known books are sources of algorithms and techniques for data structures.

Description: The main example of the Oppen papers is reversal of a list

57

by reversal of pointers. The underlying notation of datagraphs is too complex to describe here. There are two difficulties in the output assertion for the program: (1) existential quantification over nodes and arcs leads to incomplete specification, as in previous examples and (2) the assertion seems inconsistent with respect to the last node of the reversed list.

The point of contention in the [Knuth] and [Bertziss] books is the precise specification of list structures, namely the constraints on how nodes point to each other. The specific algorithms, which involve marking in preparation for garbage collection, assume constraints on pointers to list heads. It was difficult for us to elicit these assumptions from the books. If the assumptions did not hold, some reachable cells might be left unmarked.

Analysis: In private correspondence where we queried whether these errors exist, Knuth responded that "Rlinks never point to list heads" but the algorithm itself makes a test to see if a node accessed by an Rlink is a list head. Several readings later we decided that this case occurred if the list was circular and that pg. 408 implied that heads of sublists were pointed to by special sublist nodes. Reference to similar algorithms in [Bertziss] did not resolve the assumption. Similarly, private correspondence with Oppen did not resolve the question of whether the informal statement "reverse a list" was faithfully described in assertions in his assertion language.

These careful readings and correspondences arose from a research effort on the development of formalisms for data structures which would support understandable and precise proofs of properties about data structures, Yelowitz (12). There are several possible explanations for

the contention over the possibility of error in these examples:

(1) We may not have read the material carefully enough or we may simply have confused ourselves.

(2) The articles and books may have left out critical assumptions which are only revealed when our attempts to state and prove correctness placed higher demands on precision than the usual reader.

(3) There may actually be errors.

Two things are certain: it is difficult to develop complete, precise, and readable notation for discussing data structures, but it must be done before correctness proofs can be given for data structures.

Our claim is not that the cited papers are wrong and that we are right, but that when it takes multiple rounds of correspondence to resolve issues such as these there is clearly a failure in the specification process. It may well be that specifying data structures is so difficult that we will have to get along for a while with unsatisfactory approaches. Our point is that we should be aware of this problem and emphasize specifications and verifications of specifications. Put another way, we suggest that if it is not possible to determine whether a program or specification is wrong, then indeed something is wrong.

2.3 Conclusions About Specification Errors

There are several explanations for these errors:

1. In some of the examples, there was no intention of making the specifications complete. This occurs often in program proving where the output specification is split into two or more parts which are proved separately because different proof techniques or levels of detail in proofs are applicable. Thus a proof that a program meets some given set

59

of specifications is not meant to imply that the specifications are complete. 'Correctness' in this case is a purely technical matter.

2. There is often an implicit understanding in the use of some terms in specifications that constrains certain variables to be unaltered in the program, and others to be created to report the result of operating on the input. Examples are "searching", which implies that the table and the key are unaltered, and the "testing for a property, such as primeness", which says that the variable being tested is unchanged. It is debatable whether specifications should be explicit on these points, but in the formal world which starts from a set of specifications it seems fair that anything not designated as unalterable should be treated as a program variable, Perhaps an explicit convention should be adopted for this situation.

3. Confusion as to context and assumptions does not explain errors S1 and S3; these are slips in translation from concepts to formal specifications. This points to failure to confirm that the specifications implement the concept completely and correctly, failure to recognize the need for and therefore to attempt such a confirmation, or lack of tools for verifying specifications.

4. There are cases where specifications are exceedingly difficult, e.g., the line editor problem (16) to be discussed in T2.

Perhaps it is useful to view specifications as consisting of the following three components: relations between input and output, assertions about input (independently of output), and assertions about output (independently of input). There are several suggestions for devising specifications that arise from these observations.

1. Check that assumptions have been made explicit:

60

a. If the specifications are not intended to be complete, then state what is omitted, why it is omitted, and how it can be handled.

b. State which brand of correctness is being sought: "partial" where termination is not considered; or "total" which does require termination, and state whether termination includes eventually halting and/or halting "cleanly" (i.e. no run-time errors).

2. Structure the specifications using abstraction to capture the important aspects of the concept and write the formal specifications to read like the informal specifications.

3. Apply some tests to the specifications:

a. (The absurd program test) Try to find the shortest program which satisfies the specifications, instead of starting with a preconceived solution. If the program obviously does not satisfy the informal concept, the formal specifications are inadequate.

b. Break the specifications into cases and determine whether in each case the specifications match the informal concept.

c. Formulate the specifications in a different way or at a different point in time and prove consistency of the two sets of specifications.

3. Get an independent verifier for the specifications who will be naive (with respect to the problem), critical, and knowledgeable as to the above techniques for testing specifications and eliciting assumptions.

61

All in all, these errors point to the critical need for a better specification methodology. Without proper specifications, the verification process is fallacious and program design is substantially more difficult.

These examples clearly show that specifications must be 'tested' in much the same way that programs are tested, by selecting data with the goal of revealing any errors that might exist.

3. ERRORS IN SYSTEMATICALLY CONSTRUCTED PROGRAMS

3.1 Introduction

The goal of the methodology which is called by the various names "structured programming", "systematic programming", "stepwise refinement", "topdown programming", etc., is to factor the programming process into small enough steps and programs into small enough parts so that each step or part can be seen to be "correct", and so that each step or part fits together with others to give correctness at a higher level. This is not an easy concept to describe, teach, or grasp, so examples have been the main pedagogical vehicle.

The examples cited here have errors which illuminate the fact that this methodology is not yet fully understood. We hope that the examples point out pitfalls where those learning to apply the methodology should be wary and where further development of the methodology is required.

3.2 Errors in Systematically Constructed Programs

T1: Sequence Generation

Wirth (15)

Context: This is the culminating example in the chapter on stepwise program development. It is stressed as an example of a heuristic algorithm, using the important technique of backtracking.

Description: The specific problem is to "generate a sequence of N characters, chosen from an alphabet of three elements such that no two immediately adjacent subsequences are equal". The algorithm has three fundamental operations for extending, changing, and checking a candidate sequence.

The error occurs in refining the statement

$$good := (S_{m-2L+1} \ldots S_{m-L}) \neq (S_{m-L+1} \ldots S_m)$$

The Boolean variable 'good' should be set to true if the two sequences of length $L > 0$ differ in at least one pair of corresponding positions, false otherwise.

The refinement is

```
i := 0;
repeat
    good := S(m-L-i) ≠ S(m-i); i := i+1;
until ∿ good ∨ i =L
```

63

The variable 'good' should not be negated. As a counter example consider m=4, L=2, and the sequence S=3,2,1,2. The above loop forces 'good' to be false by finding the two 2's, but in fact the sequence 3,2 is not equal to the sequence 1,2 so good should be true.

Analysis: Since there is a difference of only one symbol, it might seem that this is simply a typographical error, but it is hard to interpret the insertion of the "~" character in that way.

This error seem to indicate failure to check the final step of the program construction. Here is where program proofs enter the picture because in being forced to write down a definition of "good" and to check the until test the error would probably be found. For example, an assertion to hold right before the until test is

$$[good \equiv (\exists j \mid 0 \leq j < i)(S(m-L-j) \neq S(m-j))] \wedge [1 \leq i \leq L]$$

and then it is easily seen that terminating with ~good will not give the right result. The error was actually discovered while studying the program in preparation for proof. It was later discovered that 'good' is used elsewhere in the example with similar errors.

T2: A Line Editor

Naur (16)

Contest: The article presents a view of systematic construction based on identifying important actions which are organized to meet the overall requirements.

Description: The problem requirements are "Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to

64

a line-by-line form in accordance with the following rules: 1) line breaks must be made only where the given text has BLANK or NL; 2) each line is filled as far as possible as long as 3) no line will contain more than MAXPOS characters". There are numerous problems with the specifications that lead to different interpretations of the problem, e.g. should two successive blanks be treated as ending one or two words? How should the text end?

The program also has numerous problems: it doesn't show any explicit provision for termination; if the program does terminate, the last word is left in the buffer unless followed by a BLANK or NL; there are conditions under which extra line breaks and blanks are output at the beginning; there is confusion between the two symbols NL and LF representing the line break or new line character. These errors have been extensively discussed and analyzed in Goodenough and Gerhart (17) in an example illustrating test data selection techniques.

Analysis: At one point in the paper, there is an assertion "the input character preceding the one held in BUFFER(1) was a BLANK or a NL. This has not been output."

For this assertion to be true the first time it is reached, it is necessary for the text to start with NL or BLANK, but the specifications do not state this requirement.

The point is that the action cluster methodology appears systematic, but the resulting program fails to accomplish even the ill-defined task. However, we believe that this failure can be traced back to the specifications, which are definitely inadequate. The specifications were somewhat elaborated on in Goodenough and Gerhart (18), retaining the prose format, but the authors finally concluded that there was no way to ever

get the full problem stated in English without some ambiguity or excessive length. A specification technique for this class of program has been proposed by Noonan (18).

See also error P2.

T3: A Telegram Processor

Henderson and Snowden (19), Ledgard (20)

Context: Systematic development of a program to count words and format a stream of telegrams is considered.

Description: [Henderson and Snowden] found when they ran their stepwise-constructed program that it miscounted words. They trace their error history through the steps of the program development process. [Ledgard] develops a new solution, which contains the following errors:

1. Each output line of the program begins with a blank. There is nothing in the specifications requiring or prohibiting this, but it effectively reduces the line length by 1 and seems to contradict the specification that extra blanks should be removed from the telegram on output. Careful reading of the bottom-level program was needed for the present authors to determine this.
2. The instruction "CHAR ← next-char(BUFFER)" might lead to unpredictable results. The meaning of this (predefined) instruction is not given, but apparently is to set CHAR to Λ if there are no more characters in BUFFER, and otherwise to set CHAR to the next (possibly blank) character in BUFFER and logically delete that character from BUFFER. Although Ledgard does not show the implementation of BUFFER, a standard approach to implementing a buffer of length N is to allocate an array A of length N+1, in which A[N+1] = Λ ; this $N+1^{st}$ element is analogous to an "end-of-file" marker. The problem

66

is that whenever the last word of the buffer is not followed by a blank, it is possible to execute "CHAR ← next-char(BUFFER)" twice before refilling the buffer. So the above common implementation will not satisfy the assumptions on the behavior of next-char. Again, a careful reading at the bottom level is necessary to determine this. Without knowing the behavior of next-char and the other primitives, it is not possible to justify the correctness of the final program.

3. Indentation is used as a bracketing device, rather than begin..... end. While not strictly an error, it may confuse other readers, as it did us. The use of two labels A also confused us at one point.

4. Termination conditions differ between the Final Program in Figure 6 (containing gotos) and the Final Program in Figure 7 (without gotos). For the input stream "ZZZZ HELLO DOLLY ZZZZ ZZZZ", the Figure 6 Program will print "HELLO DOLLY", whereas the Figure 7 Program will not print any telegram words. The specifications are vague on this point, which forces us to ask how the program could have been proved correct at any level.

Analysis: This problem, like the line editor problem (T2) is hard to specify completely. There are surprisingly many potential sources of error, and Henderson and Snowden warn against being lulled into a false sense of security based upon systematic program development. Ledgard provides some general guidelines on a program development methodology at the beginning of his paper, and cites the need for formalizing and debugging each of the levels. The above points of confusion show that there is still a gap in the guidelines which permits programs to be implemented without precise specifications and therefore without the

67

basis for insuring correctness at each level. In such cases, systematic construction should be expected to be quite fallible.

T4: A Sorting Algorithm in PL360

Wirth (21, p. 53)

Context: The purpose of the PL360 language is to "...further the state of the art of programming by encouraging and even forcing the programmer to improve his style of exposition and his principles and discipline in program organization" (from the abstract).

Description: The error is in procedure sort. The purpose of the procedure apparently is to sort an array a, indexed from 0 to n in increments of 4, into decreasing order. The incrementation by 4 is due to the IBM 360 architecture -- 4 bytes comprise a word, and incrementation by 1 would simply be a byte at a time.

In an outer loop, R1 goes from 0 to n in steps of 4. In an inner loop, the procedure checks if there is some index greater than R1, say R3, such that

(1) $a(R3) > a(R1)$

and (2) $a(R3) = \max \{a(R1+4), \ldots, a(n)\}$

If such an index R3 exists, then for definiteness let R3 be the smallest possible value satisfying (1) and (2). For such an R3, the appropriate logic is to swap a(R1) with a(R3) so that right after the swap $a(R1) = \max \{a(R1), a(R1+4), \ldots, a(n)\}$. Then the outer loop should continue. If no such R3 exists, then a(R1) is already the maximum of a(R1), ..., a(n) and the outer loop can continue immediately. The error is that the swap occurs even if no such R3 exists; thus R3 might be undefined (if this is the first swap), or R3 might be "left over" from a previous iteration. In programming terms, R3 is assigned

a value in the then-part of an if-then, but at the conclusion of the if-then, it assumed that the then-part has been executed.

This error has continued to appear in later reports and manuals on PL360.

Analysis: Failure to initialize a variable is a common error, e.g., see error P4 below. One virtue of structured programming is that all paths leading to a given statement can be discerned relatively easily, making it routine to verify that every variable is initialized prior to being referenced. Apparently, that verification was not performed.

The error was discovered in a classroom exercise which involved reformatting the program text.

T5: The 8-Queens Problem

Wirth (22)

Context: The "stepwise refinement" method is explained and illustrated.

Description: The "8-queens" problem is "find a way of placing 8 hostile queens on a standard 8 x 8 chessboard so that no queen may attack another". The point of contention is one of programming style and robustness rather than an actual error. When attention is restricted to only the 8-queen problem, no error will arise. If, however, we wish to generalize the solution to the N-queens problem, for arbitrary $N \geq 1$, then an error will arise for each N in which there is no solution (e.g., N=2,3). Since it might not be known in advance of running the program if a solution exists for the 8-queens problem, it is fortuitous that the error does not occur here also. The same error occurs when all solutions to the 8-queens problem are sought.

The specific error is a possible out-of-bound array reference. The x-array is indexed from 1 to 8, and represents the current board configuration; x[p]=k if a queen is present in column p, row k, where

$1 \leq p \leq j$ (j is a variable used to move left or right across columns). When the program regresses out of the first column (as will occur when no solution exists or all solutions have been produced) the following code will be executed with j=1, (according to our interpretation discussed below)

```
        j := j-1;
        i := x[j];
```

Analysis: Actually, it is somewhat ambiguous what the final program should be. After completing the stepwise refinement, Wirth observes that x[j] can be replaced by a variable i, saving several subscript computations. The proper modifications to coordinate i with x[j] are mentioned and then the affected procedures, except reconsiderprior column, are rewritten. If one constructs the complete concrete program from the latest versions of the procedures, the adjustment for i does not occur because reconsiderpriorcolumn is out of date. But if one constructs, as we did, the program with the obvious recommended change to reconsiderpriorcolumn

```
        j:=j-1; i:=x[j]
```

the subscript error occurs. A third possibility is to rewrite regress to read

```
begin   j:=j-1
        if   j ≥ 1
        then  i:=x[j]; removequeen, ...
```

but this is a major deviation from the preceding refinements.

Our conclusion is that a seemingly safe optimization did not preserve correctness and should have been checked more carefully. We are not sure how this type of program rewriting fits into the stepwise refinement method.

3.3 Conclusions About Errors in Systematically Constructed Programs

It is hard to pinpoint the exact places of failure in the systematic constructions since there are always many assumptions in effect and the reasoning is informal. Most errors seem to occur when the bottom-level code is written. It is as if the systematic construction is performed as a series of refinement steps where every step except the last, in which concrete code is produced, is carefully checked. This leads to the obvious recommendations:

1. Be especially careful to verify that the concrete program parts do exactly what the abstract parts intended.
2. After completing a systematic construction, put all of the pieces of program together and recheck, using standard methods of testing and/or proving, that the program does what was initially specified.

Some amount of formalization would probably benefit the systematic construction methodology. Care must be taken to avoid overformalization, since a point of diminishing returns can easily be reached, and passed. For example, S4 and P4 (below) fail to detect errors despite a great deal of formalism. One practical approach is to treat data reference and program structure with more symmetry. In many articles, "structure" is claimed for a program based upon the use of only well-known control structures, but mention is seldom made of the degree of locality or globality of data reference. If a variable is referenced and modified at every level of a program, then the difficulty in understanding the purpose of that variable might become inordinate, and the fact that gotos have been avoided becomes somewhat academic. More recent work (23) concentrates on the data structure aspect of systematic con-

41

struction.

The following recommendations might be useful:

In addition to the standard refinement process, keep a list of important program variables (or more general data structures). The list should explain the purpose of the variable at a problem-solving, or goal-oriented level, including its initialization, updates, and relation to other variables; a check then can be made that the purpose of the variable corresponds to the pattern of references and modifications as used by the program. Such a list might have caught the errors in T4 and P4.

It is also important to note that some errors were easily discovered by hand simulation on test values. Finally, we note an alternative viewpoint; systematic construction should expose various facts about the program which then can serve as a basis for a proof, but the systematic construction alone is insufficient to guarantee correctness.

4. ERRORS IN 'PROVED' PROGRAMS

4.1 Introduction

Testing cannot guarantee in a practical sense that a program is correct, although, in theory, testing can be viewed as a basis for an induction proof which does demonstrate correctness (17). However, program proving based on testing is not yet well-understood. The approach to program proving which has been advocated over the past few years stresses the construction of theorems (verification conditions) to express program correctness, and various mechanical techniques for proving these theorems. Other work has concentrated on proof styles, ranging from the loose arguments for correctness seen in articles on stepwise refinement to much more rigorous proofs, some of which have been mechanically produced.

The overall goal of the work on proving program correctness is to show convincingly that programs do not contain errors. The following examples demonstrate that proofs of correctness do not always discover errors, even though the proofs may be persuasive, and perhaps even "formalistic". We will have more to say about the nature of errors in proofs in mathematics at the end of this article. For now, the reader should bear in mind that there are two aspects to program proving: (1) What to prove; and (2) How to prove it. Most of the errors are best viewed as failures in defining what to prove.

4.2 Errors in Proved Programs

P1: A Linear Search Program

McGowan and Kelly (6, p. 33)

Context: The example occurs in a section intended to help readers convince themselves "that careful reasoning about programs is a better guide to correctness than extensive testing." (6, pg. 30).

Description: Suppose that a table TAB has been declared to have N elements with 1-origin subscripting and that KEY and TAB are declared of the same or compatible types. The language in use is PL/1. (The example in the book uses structures, but we are simplifying to arrays without losing the general idea). The following program is given to search an initialized TAB for an initialized value of KEY:

```
I=1;
DO WHILE (I <= N & KEY ¬=TAB(I));
     I=I+1;
END;
```

with the loop invariant

KEY $\neq$ TAB(j) for $1 \leq j \leq I-1$

73

The claim is that on exiting the loop, either I=N+1 and KEY is not in TAB or KEY=TAB(I). (In fact, the invariant needs the conjunct I $\leq$ N+1 in order to conclude I=N+1 at loop exit, but that is not the main problem here.)

The specific problem is that if KEY is not present in TAB, the final while test will be executed with the value I = N+1, making the first conjunct false. In all but the optimizing PL/I compiler, however, the second conjunct is evaluated (even though it is logically superfluous), giving rise to DATA INTERRUPTS and SUBSCRIPT RANGE errors. (This experiment was performed on an IBM 370/168 with standard IBM software in December, 1975.)

Analysis: The undefined order of evaluation of operands of logical operators is a well-known pitfall of PL/1. Left-to-right, non-superfluous evaluation is often assumed, but the PL/1 reference manual is vague on this point. Other languages, e.g. ALGOL W, make it explicit that the and operator in A and B is sequentially defined as if A then B else false.

The error shows that ignoring control within expressions and inexecutable operations can invalidate a correctness argument or a careful reasoning process. Elsewhere in reference (6), attention is paid to logical operators in assembly language macros. The authors point out that the preferred code for this problem is

```
DO I=1 TO N WHILE (KEY¬=TAB(I));
END;
```

which avoids the problem of order of evaluation of operands for this program.

P2: Line Editor

London (7)

Context: The Line Editor program has been discussed in error T2. [London] corrected one error and proved several properties of the corrected program. The goal was to illustrate the methods and some results of the approach of proving programs correct and to suggest that the approach at least be considered as a means of attaining software reliability.

Description: The program provided by London has the following abstract structure:

```
'initialize program variables';
while 'more characters to be read' do
begin 'input a character';
      'process that character' (putting it in the buffer or out-
          putting the buffer with a preceding blank or line feed,
          as required by the line specifications)
end
```

The 'more characters to be read' action is simply expressed as 'halt if no more characters'. The problem with this action is that when there are no more characters, there may still be a word in the buffer. In this version of the program, the buffer is not emptied.

Analysis: Several lemmas for properties of the program are proved: Variable types are consistent; subscript errors do not occur if the words are not oversized; the buffer array contains only legal parts of words; and the words output on a line are done so correctly. The proof line 'the output of each entire word (possibly null) after the first word must be and is preceded either by a line feed...or a BLANK...' comes close to hitting the point of error in the program, but it concentrates on showing

that the words which are output are done so correctly, not that all the words are output (and in the same order).

As in T2, the proof missed a common and well-known pitfall of this type of program, namely, failure to empty the buffer at the end of processing. The error probably was not caught because the program specifications, and hence the correctness requirements, were so loosely stated. It should also be noted that this is one of the earliest published attempts at proving a realistic program.

P3: Prime Sieve

Wulf (23)

Context: The language ALPHARD is being designed to provide, among other features, the facility for handling abstractions in both control and data structures. The prime sieve (sieve of Eratosthenes) program previously developed and proved by Hoare (24) was reworked to display the abstractions in the final text of the program. It is claimed that program proving should be factored into proofs of high level algorithms (which may often be omitted when they are well known, as in this example, or obvious) and proofs that the representations correctly reflect the high level algorithm. The intended proof style is used on the example.

Description: The high level algorithm is

```
while ~ empty(sieve) do
    (include(prime,min(sieve)); removemultiples(sieve, min(sieve)))
```

where 'prime' is declared of type powerset of the integers 1..N and initialized to empty and 'sieve' is declared of type powerset of the integers 2..N and initialized to {2,...,N}. The ultimate representation of both is bits within an array of machine words.

The error is that the 'min' routine does not return the minimum

element of the sieve, as specified by the algorithm, but instead returns the index of the minimum element as a pair of integers representing an element in an array of words and a bit in that word. The index of the least possible element of 'sieve', that is, 2, corresponds to 0. There are two effects of this error:

(1) 'include (prime, min(sieve))' causes min(sieve)-1 to be placed in'prime'.

(2) the operation removemultiples(sieve, min(sieve))' corresponds to a loop

```
for I := X step X until N do
    'remove the element with index I from sieve'
```

which is executed with X being the index of min(sieve) in sieve, thus causing an infinite loop when X is 0.

Analysis: The proof shows that the bit-word pair and powerset forms are correctly defined and attempts to show that an integer-set form is correct. The latter part of the proof states "removemultiples(n) removes the elements at indexes n, 2n, 3n, ..., size of powerset" but this cannot be true when n=0 and, even if that worked, the sieve would be emptied when n=1. It is not proved that the element which is included in 'prime' is actually the minimum element of the sieve. The error seems to have occurred because the data representations do not actually correspond to the algorithm, with a resulting confusion between the minimum element of the sieve and its index. Note that usually the initialization is stated in the algorithm but that in Alphard, initialization is distributed to the data structure forms.

The original claim that program proving can be factored into algorithm and data representation

is probably justified, but that there is still a substantial proof step in showing that the representations are faithful to the intent of the algorithm. It should be noted that this is the first description of Alphard and a more recent description (30) uses better defined language constructs and takes a more rigorous approach.

P4: Maximum of a Series of Powers and Matrices

Lanzarone and Ornaghi (25)

Context: The paper presents a variation of the usual correctness formalism to describe the stepwise refinement method.

Description: The example is specified: 'A symmetric matrix with positive or null elements has to be multiplied by itself until the maximum of its elements is greater than or equal to an assigned positive real number alpha'. Let · represent matrix multiplication, $||M||$ represent the value of the maximum element of matrix M, and x denote the input matrix.

The top level program is:

$(a,b,c) \leftarrow (x,x,||x||)$

while $c \leq$ alpha do

$(a,c) \leftarrow (a \cdot b, ||a \cdot b||)$

The error occurs in the concrete code refined from the body of the loop. As each element, say e, of the new product matrix is computed, a variable d is set to max(d,e). However, d is not initialized at the start of each matrix product computation, but only at the beginning. This causes d to contain not the maximum of the current matrix, but the maximum of all matrices computed so far. It might be thought that the historical maximum always equals the current maximum, but for the matrix

$$x = \begin{pmatrix} .95 & .15 \\ .15 & .95 \end{pmatrix}$$

78

the successive maxima are .95, .925, .921, .936, .969. Nevertheless, the program will still work correctly because if the current maximum is less than a previous maximum, and termination has not occurred, then the current maximum is less than alpha.

Analysis: The point is that the final program is not a refinement of the top level program because the variable d is not reinitialized every time a new matrix is computed. The proof does not catch this discrepancy nor did the proof give any indication that the final result is correct nevertheless. It is debatable whether this should be considered an error, since the final program is correct (assuming there are not other errors which we have not found). However, it could have just been fortuitous that everything worked out in this example, and in other examples the luck might give out. The overall flaw in the approach seems to be that the interfaces between refinements were not carefully checked. For example, the input assertion about the section in error permitted 'd' to be any real value, not necessarily 0.

4.2 Conclusions about the errors in proved programs

There are several common features of these four errors:

1. The inductive assertion method is used informally. It is difficult to apply the assertion method to the line editor problem, lacking a suitable assertion language. We believe the assertion method could have caught the error in P4 since the property of d being the maximum of the current matrix would have been in the loop assertion, as well as the error in P3 since the relation between elements of prime and sieve must be stated.

All of these programs have a loose notion of the required verification task. The presentation in [6] is deliberately informal in order to

introduce correctness concerns. P2 uses an informal approach which is dictated by the informal nature of the specifications. P3 skips a crucial aspect of the proof, namely that the representation corresponds to the algorithm. P4 seems to skip the interface steps to concentrate on proofs for the individual refinements, although such interfaces play an important role in the theory and practice of program development.

2. Three of the errors are related to proper termination: error P1 relates to the value of the conjunction at the time the loop exists; error P2 occurs at the end of the text; error P3 results in a nonterminating loop; error P4 is related to initialization.

It is common in program proving to treat the termination task informally since termination in most of the examples is relatively obvious and easily checked. These errors suggest that perhaps more effort should be concentrated on termination, especially since it is well known that many programming errors occur at boundary points, which includes initialization and termination.

3. Ironically, each of the first three errors are easily discovered by the standard methods of hand simulation and testing. For example, test cases for P1 would undoubtedly include the two subcases of KEY present, and not present, in TAB, and the error would be revealed on any but the optimizing PL/1 compiler. Testing of error P2 might show the last word left in the buffer, depending on type of input device. Hand simulation on the prime sieve program quickly revealed the problem at the first loop iteration when 2 is the minimum element of the sieve. (We had previously been told that an error exists in this program, but we were not told the details.)

Based on these generalizations, we make the following recommendations

for increasing the value and credibility of program proofs:

1. Do not ignore the "standard" methods of verification. London (7) gives the "hint that one should be fairly confident the program is correct before starting to prove it so. This confidence may, for example, arise from the standard testing/debugging process."

2. Check the proof and program especially closely at known pitfalls and problem areas of the programming language and the programming task. One goal of programming language design is to minimize the number of such trouble spots. There does not appear to exist a well-documented, widely distributed and suitably general catalog of trouble spots in programming, but there is certainly informal communication of a large amount of bitter experience.

3. Adopt a cautiously skeptical attitude toward proofs, as one of several possible means of persuasion, in which formalization and abstraction might provide some new insights and documentation. Keep in mind, however, that there are usually at least some parts of the program that are better-explained informally, and it is pointless to attempt subverting these parts to fit a particular formalism. Formalism should supplement, definitely not replace, common sense and programming experience and intuition. See Redish (26) for various types of common sense questions to supplement the assertion method.

4. Even though a challenging aspect of a proof has been solved, one should not let one's guard down on the more mundane aspects of the program.

5. First concentrate a large amount of effort on stating what should be proved in order to guarantee the program is correct, and then set about proving it. It seems fair to say that in most of the above errors the proving task was not well-understood. Therefore some things

81

which should have been proved were ignored, resulting in failure to catch errors.

5. OVERALL CONCLUSIONS AND RECOMMENDATIONS

We have identified and discussed some common features for each of the three classes of errors. We can now elaborate on some common features of all three classes.

Observation 1: The tasks were not well defined: it was not recognized that formal specification must be shown to capture the underlying informal concept; there were gaps in the statements of what should be proved about programs, especially proper termination; systematically constructed programs were not checked closely to confirm their correctness.

Recommendation 1: Identify more carefully the complete task, for example, by including those parts which cover the errors we have discussed here. Make sure the task is well understood and precisely stated before undertaking the time consuming and absorbing process of verifying that the task was accomplished.

Observation 2: The errors are not deep. The standard methodologies and everyday programming knowledge are sufficient to reveal most of them. The errors seem to have been overlooked because the authors were concentrating on pedagogical points and therefore looking at the program from restricted viewpoints.

Recommendation 2: Apply as many techniques as possible to the task: perform testing as well as proving; look for known difficult and error prone language constructs; obtain an independent verifier to read and check the results. The greatest confidence arises from consistent positive results from different methodologies applied to the same task,

because different methodologies often have compensating strengths and weaknesses.

Observation 3: There is a tendency to concentrate more effort on the harder parts which require sophisticated techniques and less effort on the "obvious" and easier parts. It is often claimed that the methodologies are even more essential in multiprocessing programs than in sequential programs. The errors show they do not yet work reliably for sequential programs.

Recommendation 3: Do not bring to the task preconceived notions of hard and easy, e.g. "termination is always trivial to prove" or "inductive assertions are always hard to formulate". The apportionment of effort must be somewhat tailored to the specific task. Do not get so bogged down in formal proofs that some aspects of the task are ignored completely.

Observation 4: Most of the erroneous programs were also well-structured, according to current criteria. It is often claimed that good structure makes it easier to detect errors, but these errors show that it is no guarantee.

Recommendation 4: Do not confuse good structure with correctness. If the structure is good, then make use of the clarity thereby gained to verify the program, at least informally.

Observation 5: The methodologies proposed to increase software reliability are still in their early stages of development: the tasks are not easily taught or learned; old habits make it hard to take seriously the importance of some tasks, e.g. the common practice of writing the specifications after writing the program, or worse, never writing the specifications at all; there is a tendency to believe that

following the techniques will automatically bring favorable results, e.g. systematic construction will lead to correct programs.

Recommendation 5: Do not view new methodologies as panaceas, especially when one has little experience in applying the methodologies or is unaware of the pitfalls. Just as with any other skill, it will take considerable training and experience before the new programming methodologies are mastered. Part of that experience will undoubtedly be committing and recovering from errors.

6. SOME RELATIONSHIPS BETWEEN MODERN PROGRAMMING METHODOLOGIES AND MATHEMATICS

Earlier we claimed that the common feature of the new methodologies is the emphasis on the use of mathematical reasoning in programming. A natural question to ask is, How well does mathematical reasoning work in mathematics?

Here are a few documentations of error processes in mathematics.

1. The Mathematical Games section of the December, 1975, issue of Scientific American (27) reports an interesting instance of error. A proof had been submitted that a particular algorithm produced all solutions to a given problem. A counterexample in the form of a missed solution was later submitted. The nature of the proof error was not given. The author of the original "proof" was quoted from a book he had authored to the effect that there is no "magic formula for a proof which makes it immutable and unarguable henceforth and forevermore."

2. An interesting paper by an eminent mathematician P.J. Davis, (28) relates many instances of errors in mathematics. It concludes that "a derivation of a theorem or a verification of a proof has only probabilistic validity" and that mathematics, as a somewhat experimental science, is "saved from chaos

by the stability of the universe...and the self-correcting features of usage."

3. Schwartz (29) relates the following anecdote: "I think here of a case that became famous a few years ago, in which after certain statements in algebraic number theory had been proved by three independent methods in published papers (an algebraic proof, an analytic proof, and an elementary proof), a counter-example was published."

Another point to consider is the purpose of a proof. In addition to the obvious one of certification, Davis also points out the "discovery" aspect of proofs. A mathematical proof of a given statement helps to elicit the hypotheses under which the statement holds and perhaps induces minor alterations in the statement. Analogously, a program proof can help to discover conditions on input under which the program will or will not execute completely and provide the required output. These conditions may or may not be subsumed by the program specification, which may need to be altered.

Yet another aspect is that a proof should reveal clearly why a theorem holds. Likewise, a program proof should reveal why the program works and thus serve as a form of documentation. All in all, mathematical reasoning leads to a deeper understanding of the subject being studied, if not to certainty in manipulating the understanding.

The certification aspect of mathematical proofs has an obvious carryover to program proofs. It is recognized in mathematics that a proof does not become a proof until 'there has been a consensus of experts that the proof is right' (28). In program proving, we would like one of our experts to be mechanical proof checker, but of course this leads to the question of correctness of the proof checker, as well as the immense

difficulty of constructing and the expense of running such a checker. It should be observed that many of the above errors occur in papers which have undergone a supposedly rigorous review process before publication. It is a reasonable expectation that each article which had not been reviewed had nevertheless been read by at least one other competent person. Yet the errors persist. The conviction from a proof that a statement or program is correct is only meaningful if the person being convinced is critical and trained to detect proof failures.

These are similarities between mathematical reasoning in mathematics and in programming. There are diffferences, also.

1. Mathematical theorems are often stated and proved for their elegance or their role within a theory. It is not necessary that there be an immediate, or even an eventual, application of the theorem. In programming, we are more immediately concerned with correctness since program errors may be costly or dangerous.

2. There is usually an established and well known theory in which a mathematical theorem is embedded, wheras in programming, each program proof is usually isolated. A mathematician does not start from scratch, but instead builds upon a body of theorems with the result that the task is easier and the theorem can be shown to be consistent or inconsistent with other theorems in the theory. Currently, each program proof starts from scratch and must be examined in isolation. This state will probably change as a more mathematical theory of programs is evolved from present work on program correctness and from the abstraction and organization of programming knowledge.

3. Studies in the mathematical foundations of computer science lead to advances in machine and language design. A current premise is

that languages should facilitate mathematical reasoning in programming, be semantically defined in a mathematical fashion, and be subjected to rigorous mathematical analysis.

This discussion leaves us with the fundamental question:

What is the role of formalism and mathematical reasoning in programming methodologies? Based on our study of errors, we conclude that it is one, but not the only, or necessarily best, tool for verifying programs. It provides evidence of a logical nature that programs are substantially correct, the degree of certainty being somewhat related to the depth of logical analysis and the skills of the analyser(s), but never absolute. On the other hand, testing provides empirical certainty of at least some correctness aspects of a program. Experience with both testing and mathematical reasoning should convince us that neither type of evidence is sufficient and that both types are necessary.

There are two important roles, other than verification, for formalism in programming methodologies: (1) they provide the training in rigorous thinking which is essential for good programming and (2) they provide the most effective language for organizing and expressing knowledge about programs. Of course, this is what the leading programming methodologists have been saying for years. We hope that this paper provides new, and more realistic, insight into the mathematical foundations of these methodologies. One unfortunate aspect of that reality is that "mathematics is a human activity subject to human fallibility." (5) This statement should not be interpreted to say that the mathematical approach should be abandoned, for it will always be a necessary tool. Nor should it be construed to mean that mechanical tools are the only solution, for these must ultimately be evaluated by mathematical means. We simply must learn to live with fallibility.

87

TABLE OF ERRORS

Reference Code	Name of Program(s)	Classification(s)
S1	Prime Test	Specifications & proofs
S2	Sorting and Searching	Specifications & proofs
S3	Magic Square Generator	Specifications & proofs
S4	Data structure algorithms	Specifications & proofs
T1	Sequence Generation	Systematic Construction
T2/P2	Line Editor	Systematic Construction/proofs
T3	Telegram Processor	Systematic Construction & proofs
T4	Sorting Algorithm	Systematic Construction
T5	8-queens	Systematic Construction
P1	Linear Search Program	Proof
P3	Prime Sieve	Systematic Construction & proofs
P4	Powers of matrices	Systematic Construction & proofs

ACKNOWLEDGEMENTS

Error T2 was originally analyzed with the help of John Goodenough. Ted Linden alerted us to error P3. Harlan Mills is the source for error T4. Error T5 was independently confirmed by Leon Stucki. Parts of errors S2, T2, and T3 have been previously discussed in the cited articles and first sowed the seeds of doubt in our minds. We appreciate the comments of many colleagues, especially Harlan Mills, Bernard Elspas, and Ralph London, on an earlier and the present version of this paper. However, the interpretations of the errors and the conclusions are our own. The paper was distributed to the authors who committed errors and we have tried to incorporate their replies, where received.

BIBLIOGRAPHY

1. Liskov, B., and Zilles, S., Specifications techniques for data abstractions, IEEETSE 1, pp. 7-19.

2. King, J.C., A program verifier, Ph.D. Dissertation, Carnegie-Mellon University, 1969.

3. Wegbreit, B., The synthesis of loop predicates, CACM 17, pp. 102-112.

4. Deutsch, L.P., An interactive program verifier, Ph.D. Dissertation, University of California, Berkeley, June 1973.

5. Mills, H.D., How to write correct programs and know it, Proceedings of International Conference on Reliable Software, April 1975, Los Angeles, pp. 363-370.

6. McGowan, C.L., and Kelly, J.R., Top-Down Structural Programming Techniques, Mason Charter, New York 1975.

7. London, R., Software reliability through proving programs correct, 1971 IEEE Conference on Fault Tolerant Computing, pp. 125-129.

8. Hoare, C.A.R., Proof of a program: Find, CACM, 14, 1971, pp. 39-45.

9. Gerhart, S.L., Verification of APL programs, Ph.D. Dissertation, Carnegie-Mellon University, 1972.

10. Oppen, D.,On Logic and Program verification, Ph.D. Dissertation, University of Toronto, April, 1975. (Technical Report No. 82).

11. Cook, S. and Oppen, D., An assertion language for data structure, 2nd Symposium on Principles of Programming Languages, 1975, pp. 160-166.

12. Yelowitz, L., Assertions about data structures (unpublished manuscript).

13. Knuth, D.E., The art of computer programming, vol. 1, Fundamental algorithms, Second edition, Addison-Wesley Reading, Mass., 1973.

14. Berztiss, A.T., Data structures: Theory and practice, Second edition, Academic Press, New York, 1975.

15. Wirth, N., Systematic programming, Prentice-Hall, 1972. (1st printing)

16. Naur, P., Programming by action clusters, BIT 9, 1969, pp. 250-258.

17. Goodenough, J., and Gerhart, S., Toward a theory of test data selection, IEETSE, 1, no. 2, 1975, pp. 156-173.

18. Noonan, R., Structured programming and formal specifications, IEEETSE 1, 1975, pp. 421-425.

19. Henderson, P., and Snowden, R., An experiment in structured programming, BIT 12, pp. 38-53.

20. Ledgard, H., The case for structured programming, BIT 13, pp. 45-57.

21. Wirth, N., PL360, A programming language for the 360 computer, JACM 15, pp. 37-74.

22. Wirth, N., Program development by stepwise refinement, CACM 14, pp. 221-227.

23. Wulf, W.A., ALPHARD: Toward a language to support structured programs, Report AFOSR-TR-74-1434, Carnegie-Mellon University, April 1974.

24. Hoare, C.A.R., Notes on data structuring, in Dahl, et al., Structured Programming, pp. 83-174.

25. Lanzarone, G.A. and Ornaghi, M., Program construction by refinements preserving correctness, The Computer Journal 18, pp. 55-62.

26. Redish, K.A., Comments on London's certification of algorithm 245, CACM 14, pp. 50-51.

27. Dec. 1974 Scientific American, Mathematical Games Section.

28. Davis, P.J., Fidelity in Mathematical Discourse: Is one and one really two?, Am. Math. Mo., 1972, pp. 252-263.

29. Schwartz, J.A., An overview of bugs, in Debugging Techniques in Large Systems, Prentice-Hall, 1971, pp. 1-16.

30. Wulf, W., London, R. and Shaw, M., Verification and abstraction in ALPHARD, unpublished.

31. Howden, W., Reliability of the path testing strategy, this issue.

91