


THE DECEMBER 1983
VOL. 62, NO. 10, PART 2



BELL SYSTEM
TECHNICAL JOURNAL

COMPUTING SCIENCE AND SYSTEMS

- Theory of Program Testing—An Overview** 3073
R. E. Prather
- Parallel Fault Simulation Using Distributed Processing** 3107
Y. H. Levendel, P. R. Menon, and S. H. Patel
- Two New Kinds of Biased Search Trees** 3139
J. Feigenbaum and R. E. Tarjan
- An Algebraic Theory of Relational Databases** 3159
T. T. Lee
- Generation of Syntax-Directed Editors With Text-Oriented Features** 3205
B. A. Bottos and C. M. R. Kintala
- Performance Analysis of a Preemptive Priority Queue With Applications to Packet Communication Systems** 3225
M. G. Hluchyj, C. D. Tsao, and R. R. Boorstyn

THE BELL SYSTEM TECHNICAL JOURNAL

ADVISORY BOARD

D. E. PROCKNOW, *President*,
I. M. ROSS, *President*,
W. M. ELLINGHAUS, *President*,

Western Electric Company
Bell Telephone Laboratories, Incorporated
American Telephone and Telegraph Company

EDITORIAL COMMITTEE

A. A. PENZIAS, *Chairman*, M. M. BUCHNER, JR., R. P. CLAGETT, B. R. DARNALL,
B. P. DONOHUE, III, I. DORROS, S. HORING, R. A. KELLEY, R. W. LUCKY, R. L. MARTIN,
J. S. NOWAK, G. SPIRO, and J. W. TIMKO

TECHNICAL EDITORIAL BOARD

M. D. MCILROY, *Technical Editor*, A. V. AHO, D. L. BAYER, W. FICHTNER, L. E. GALLAHER,
R. W. GRAVES, M. G. GRISHAM, B. W. KERNIGHAN, Y. E. LIEN, S. G. WASILEW, and S. J. YUILL

EDITORIAL STAFF

B. G. KING, *Editor*, PIERCE WHEELER, *Managing Editor*, LOUISE S. GOLLER, *Assistant Editor*,
H. M. PURVIANCE, *Art Editor*, and B. G. GRUBER, *Circulation*

THE BELL SYSTEM TECHNICAL JOURNAL (ISSN0005-8580) is published by the American Telephone and Telegraph Company; 195 Broadway, N.Y., N.Y. 10007, C. L. Brown, Chairman and Chief Executive Officer; W. M. Ellinghaus, President; V. A. Dwyer, Vice President and Treasurer; T. O. Davis, Secretary.

The Journal is published in three parts. Part 1, general subjects, is published ten times each year. Part 2, Computing Science and Systems, and Part 3, single-subject issues, are published with Part 1 as the papers become available.

The subscription price includes all three parts. Subscriptions: United States—1 year \$35; 2 years \$63; 3 years \$84; foreign—1 year \$45; 2 years \$73; 3 years \$94. Subscriptions to Part 2 only are \$10 (\$11 foreign). Single copies of the journal are available at \$5 (\$6 foreign). Payment for foreign subscriptions or single copies must be made in United States funds, or by check drawn on a United States bank and made payable to The Bell System Technical Journal and sent to Bell Laboratories, Circulation Dept., Room 1E-335, 101 J. F. Kennedy Parkway, Short Hills, N. J. 07078.

Single copies of material from this issue of The Bell System Technical Journal may be reproduced for personal, noncommercial use. Permission to make multiple copies must be obtained from the editor.

Comments on the technical content of any article or brief are welcome. These and other editorial inquiries should be addressed to the Editor, The Bell System Technical Journal, Bell Laboratories, Room 1J-319, 101 J. F. Kennedy Parkway, Short Hills, N. J. 07078. Comments and inquiries, whether or not published, shall not be regarded as confidential or otherwise restricted in use and will become the property of the American Telephone and Telegraph Company. Comments selected for publication may be edited for brevity, subject to author approval.

Printed in U.S.A. Second-class postage paid at Short Hills, N. J. 07078 and additional mailing offices. Postmaster: Send address changes to The Bell System Technical Journal, Room 1E-335, 101 J. F. Kennedy Parkway, Short Hills, N. J. 07078.

THE BELL SYSTEM TECHNICAL JOURNAL

DEVOTED TO THE SCIENTIFIC AND ENGINEERING
ASPECTS OF COMPUTING

Volume 62

December 1983

Number 10, Part 2

Theory of Program Testing—An Overview

By R. E. PRATHER*

(Manuscript received January 18, 1983)

In this paper, we provide a detailed survey of the various approaches to program testing that have been proposed in recent years. Particular attention is given to a discussion of the developing theory of program testing and to the decomposition of the testing problem into the program graph construction, test path selection, and test case generation phases. Examples are included to illustrate the different testing strategies. Comparisons are made from one method to another, all in a uniform terminology and notation, to facilitate an understanding of various combinations of strategies that might lead to a more workable testing methodology.

I. INTRODUCTION

The general goal of software testing is to affirm the quality of a program through systematic exercising of the code in a carefully controlled environment. The execution of a program test scheme should validate an expected prespecified behavior, ideally serving to demonstrate the absence of program errors. Considering the difficulty of obtaining actual proofs of program correctness, program testing

* University of Denver, Colorado.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

may be the only effective means for assuring the quality of software systems of nontrivial complexity.

The state of the art in software testing as of a decade ago is broadly surveyed in the book by Hetzel,¹ representing an ad hoc approach at best. During the intervening years, computer programming methodology has made great strides toward improving the quality of our product. And yet, software testing has remained a kind of "black art", only vaguely understood by its practitioners. Happily, this situation is changing. The development of the beginnings of a theory of testing are well under way, and the more recent literature shows great promise for brighter days ahead. Some of these ideas are discussed in a new book by Myers,² and further elaboration can be found in the survey papers by Miller.³⁻⁶

In this overview, we summarize in detail the more recent literature on software testing and present the more important results in a uniform framework, style, and notation. We hope that this perspective will help to focus attention on the more viable alternatives and to point the way toward the most promising directions for future research and development.

II. GENERAL THEORY—THE FUNCTIONAL APPROACH

The first attempt to describe a generalized theory of testing is found in the work of Goodenough and Gerhart,^{7,8} A related study is that of Hamlet.⁹ In the former, a *program* is viewed as a function $F:D \rightarrow R$ over an *input domain* D with values in an *output range* R . The *program specification* can also be viewed as a function $G:D \rightarrow R$, whether completely specified or not. For testing purposes, we must compare $F(d)$ with $G(d)$ for selected inputs d in D . Though such an exhaustive test is not feasible in general, we say that the program F is *correct* if we have

$$F(d) = G(d) \text{ (for all } d \text{ in } D),$$

recognizing that this is simply a theoretical notion, one not necessarily capable of direction verification.

In any practical setting, we will only be able to examine the behavior of the program for a few selected input values. Realizing this, we say that a *test* for the program F is a (finite) subset T of D . Recalling the 'goal of software testing,' T is said to be an *ideal test* (for F) if

$$\text{success}(T) \Rightarrow \text{correct}(F),$$

i.e., if $F(t) = G(t)$ for t in T implies the same for all t in D . Note that the successful execution of an ideal test would constitute a proof of correctness. Given the difficulty in finding proofs of correctness, however, we should not be surprised to learn that ideal tests, in this

sense, are difficult to discover. (We note that the ‘trivial’ ideal test, the exhaustive one with $T = D$, though easily stated is ordinarily unmanageable in size.)

As a matter of fact, we would prefer not to ‘discover’ our tests at all, but to have them ‘selected’ on the basis of some sensible criterion. Formally, a *test selection criterion* (for a program F) is a (true-false) predicate C over the subsets of D . Following Goodenough and Gerhart once again, such a criterion C is *reliable* (for F) if

$$C(T1) \text{ and } C(T2) \Rightarrow \text{success}(T1) = \text{success}(T2),$$

and, on the other hand, C is said to be *valid* (for F) if

$$\sim \text{correct}(F) \Rightarrow \sim \text{success}(T)$$

for some T satisfying $C(T)$. In general, reliability refers to the consistency with which results will be produced within the selection criterion, whereas validity refers to the ability to produce meaningful results, regardless of their consistency.

It is clear that these notions of reliability and consistency are quite strong. Perhaps the most convincing statement to this effect is given by the following:

Theorem (Goodenough and Gerhart): If C is reliable and valid, then $C(T)$ implies that T is an ideal test.

On the other hand, Weyuker and Ostrand¹⁰ have argued that these notions are not strong enough, referring as they do to a particular program. If the same ideas are extended, however, so as to apply “uniformly” over all programs F , then one obtains the following:

Theorem (Weyuker and Ostrand): If C is uniformly reliable and uniformly valid, then $C(T)$ implies that $T = D$, i.e., T is an exhaustive test.

Surely this carries the original ideas too far. And in fact, the theorem can be understood to say, “If nothing is known about the errors in the program, a test criterion is guaranteed ideal (in the sense of Goodenough and Gerhart) if and only if it selects the entire input domain.” What is probably needed to arrive at a more practical alternative is a weakening of the Goodenough and Gerhart theory. This is the general thrust of Hamlet’s work, but results along these lines thus far are less than satisfactory, showing perhaps more promise toward applications to program maintenance than to testing. The interested reader should consult Ref. 9 for details.

A test selection criterion C should outline the properties of a program that must be exercised to constitute a “thorough” test, ideally one whose successful execution implies an error-free program. Following Goodenough and Gerhart once again, we may suppose that C is described as a finite set $\{c\}$ of *test predicates* (i.e., logical conditions on

the input data), and we then choose T subject to the condition(s):

$$C(T) \Leftrightarrow \begin{array}{l} \text{for all } c \text{ in } C, \text{ there is } t \text{ in } T \text{ with } c(t) \\ \text{for all } t \text{ in } T, \text{ there is } c \text{ in } C \text{ with } c(t). \end{array} \quad (*)$$

In words, every test predicate belonging to C should be satisfied by at least one test datum t in T , and conversely, every t in T must satisfy at least one test predicate.

It is suggested that the test predicates be derived from the program specifications—this is the essence of the *functional approach* (or “black box” approach) to testing. But the claim is made⁷ that to have a reasonable chance of constituting a reliable criterion, C must be composed of test predicates satisfying (at least) the following set of conditions:

Condition 1: Every individual branching condition in the program must be represented by an equivalent test predicate.

Condition 2: Every potential termination condition (e.g., error, overflow, etc.) must be represented by a corresponding test predicate.

Condition 3: The range of every variable appearing in a test predicate must be partitioned into classes that are “treated in the same way” by the program.

Condition 4: Every condition relevant to the proper functioning of the program that is implicit in the program specification or of one’s knowledge of the program must be represented by a corresponding test predicate.

Condition 5: The test predicates must be “independent,” in that all data satisfying a particular test predicate must exercise the same path in the program and must test the same branch conditions.

We note that only the second and fourth of these conditions are of a “functional” nature. The others are “structural,” that is, relating more to the topology of the underlying flowchart. It would seem, therefore, that any reasonable testing strategy should address both points of view.

Consider the following example, the often cited problem of classifying triangles:

Specification:

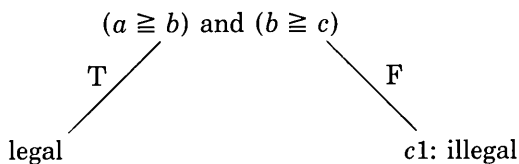
Input: Three positive integers $a \geq b \geq c$.

Output: An indication as to whether:

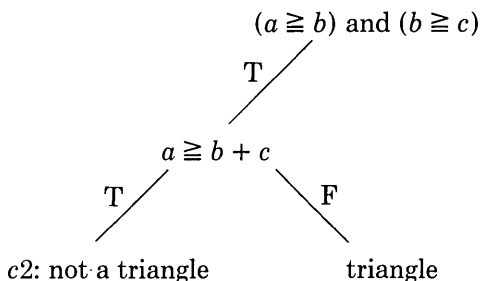
1. They do not represent the sides of a triangle
2. They are the sides of an equilateral triangle
3. They are the sides of an isosceles triangle
4. They are the sides of a scalene right triangle
5. They are the sides of a scalene obtuse triangle
6. They are the sides of a scalene acute triangle.

This problem is especially well suited to the “functional” approach.

Since the whole purpose of the problem is to classify its input domain, there is an obvious specification-based derivation of test predicates. We may first divide the universe of triples (a, b, c) into legal and illegal forms:



For the legal entries, we may further distinguish two cases:



and the triangles may then be subdivided into six subclasses:

- c3 : $(a = b)$ and $(b = c)$ equilateral
- c4 : $(a = b)$ and $(b > c)$ isosceles
- c5 : $(a > b)$ and $(b = c)$ and $(a < b + c)$ isosceles
- c6 : $(a > b)$ and $(b > c)$ and $(a*a = b*b + c*c)$ right scalene
- c7 : $(a > b)$ and $(b > c)$ and $(a*a < b*b + c*c)$ acute scalene
- c8 : $(a > b)$ and $(b > c)$ and $(a*a > b*b + c*c)$ and $(a < b + c)$ obtuse scalene.

If we set $C = \{c(i) : i = 1 \text{ to } 8\}$ and choose one triple from each input subdomain, we may obtain the test set:

- $t1 = (1, 2, 3)$
- $t2 = (14, 6, 4)$
- $t3 = (1, 1, 1)$
- $t4 = (2, 2, 1)$
- $t5 = (3, 2, 2)$
- $t6 = (5, 4, 3)$
- $t7 = (6, 5, 4)$
- $t8 = (4, 3, 2).$

Such a test set will automatically satisfy (*) and the test selection criteria will more than likely meet Conditions 2 and 4 above. But we have no guarantee that the “structural” conditions 1, 3, 5 will be met, since we haven’t looked at the program!

Weyuker and Ostrand^{10,11} have made the cogent suggestion that the input domain be partitioned *both* on the basis of the specification-driven, program-independent properties mentioned above, and on the structural properties of the program as well. It seems that this is the only way to meet all five of the Goodenough and Gerhart conditions, and to thus have a chance of approaching a reliable test selection criterion $C = \{c\}$ defined by a set of test predicates.

Suppose we add a sixth (implicit) condition to the five that are outlined above, namely:

Condition 6: The test predicates must be “complete” in that every input of the domain D must satisfy (exactly—see condition 5) one of the test predicates.

Then Conditions 5 and 6 ensure that $C = \{c\}$ defines a partition

$$\kappa = \{C\}$$

on the input domain. When we concentrate only on the problem specifications, as above, we obtain the *problem partition* consisting of *problem domains* C . Having a completed version of the program in hand, we may speak as well of a *path partition*

$$\pi = \{P\}$$

of the same domain D , where each *path domain* P comprises a class of inputs that traverse the same path through the program. Thus, the path partition separates D into classes of inputs that are treated the same way by the program, whereas the problem partition separates D into classes that should be treated the same.

There is no assurance that these two partitions will coincide, nor is it necessary that they do. But ultimately (or at least, hopefully), the program and its algorithm all derive from the original problem specification, so we should not expect the two partitions to differ markedly. On the other hand, those differences that do exist are fruitful places to look for errors! Recognizing this, Weyuker and Ostrand have suggested that the problem and path partitions be intersected, yielding a finer partition

$$\sigma = \kappa \wedge \pi = \{C \cap P\} = \{S\}$$

of nonoverlapping subdomains S of the domain D , and they further suggest that this be used as the ultimate test selection criterion, choosing one test case from each subdomain as before.

In the terminology of Weyuker and Ostrand,^{10,11} a subdomain S of D is said to be *revealing* (of errors) if

$$\text{success}(s \text{ in } S) \Rightarrow \text{correct}(F, S),$$

i.e., if the successful execution of any input from S implies correctness

of the program over the whole subdomain. Since the inputs of a subdomain S (in the intersection above) should be and in fact are treated the same by the program, the hope is extended that these are, in essence, the revealing subdomains. A successful execution of one test datum from each of the subdomains S then implies (or at least suggests) the correctness of the program over the whole domain.

Consider the “triangle classification problem” once again, and suppose we are presented with the program (flowchart) of Fig. 1 as

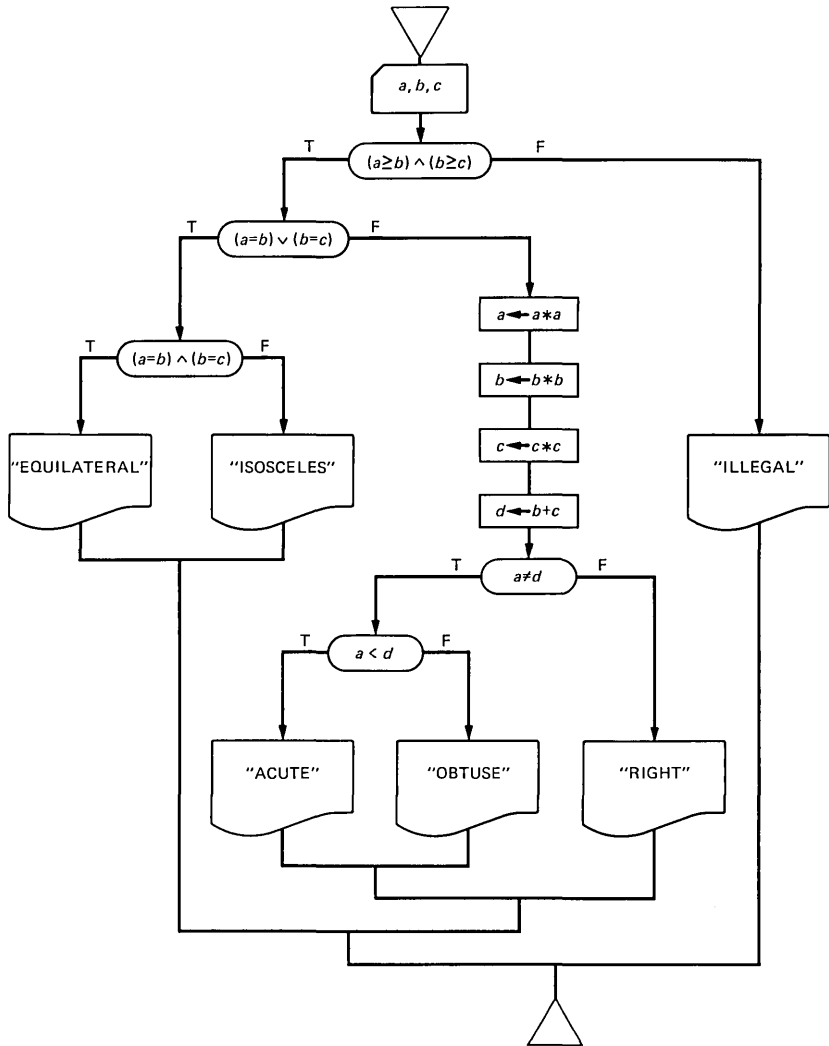


Fig. 1—Flowchart for classifying triangles.

representing a solution to the problem. There are six paths through the program, as described by the conjunctions of branch conditions defined by each path, as follows:

- $p1 : \sim[(a \geq b) \text{ and } (b \geq c)] = (a < b) \text{ or } (b < c)$
- $p2 : (a > b > c) \text{ and } (a*a = b*b + c*c)$
- $p3 : (a > b > c) \text{ and } (a*a < b*b + c*c)$
- $p4 : (a > b > c) \text{ and } (a*a > b*b + c*c)$
- $p5 : (a \geq b \geq c) \text{ and } [(a = b) \text{ or } (b = c)] \text{ and } \sim[(a = b) \text{ and } (b = c)]$
 $= (a = b > c) \text{ or } (a > b = c)$
- $p6 : a = b = c.$

Intersecting the six corresponding path domains $P[i]$ ($i = 1$ to 6) with the eight earlier problem domains $C[i]$ results in a partition $\{S\}$ of nine subdomains characterized as follows:

- $S1 = C1 = C1 \cap P1 : (a < b) \text{ or } (b < c)$
- $S2 = C2 \cap P4 : (a > b > c) \text{ and } (a \geq b + c)$
- $S3 = C2 \cap P5 : (b = c) \text{ and } (a \geq b + c)$
- $S4 = C3 = C3 \cap P6 : a = b = c$
- $S5 = C4 = C4 \cap P5 : a = b > c$
- $S6 = C5 = C5 \cap P5 : (a > b = c) \text{ and } (a < b + c)$
- $S7 = C6 = C6 \cap P2 : (a > b > c) \text{ and } (a*a = b*b + c*c)$
- $S8 = C7 = C7 \cap P3 : (a > b > c) \text{ and } (a*a < b*b + c*c)$
- $S9 = C8 = C8 \cap P4 : (a > b > c) \text{ and } (a*a > b*b + c*c)$
 $\text{and } (a < b + c)$

in very close agreement with the problem partition $\{C\}$ obtained earlier.

The problem we are discussing has a rather precise functional specification so that we would expect that the problem and path partitions might nearly coincide. Nevertheless, there is a slight discrepancy, and in place of the test datum $t2 = (14, 6, 4)$ we would now have to choose two, say $(14, 6, 4)$ and $(3, 1, 1)$. A test of the resulting nine data points would then reveal two errors, as shown in Table I below.

Table I—Test of nine data points

Domain	Test Data	Correct Output	Actual Output
S1	(1, 2, 3)	Illegal	illegal
S2	(14, 6, 4)	Not a triangle	obtuse
S3	(3, 1, 1)	Not a triangle	isosceles
S4	(1, 1, 1)	Equilateral	equilateral
S5	(1, 1, 1)	Isosceles	isosceles
S6	(2, 2, 1)	Isosceles	isosceles
S7	(3, 2, 2)	Right	right
S8	(5, 4, 3)	Acute	acute
S9	(6, 5, 4)	Obtuse	obtuse
	(4, 3, 2)		

The programmer has failed to take account of those situations where $a \cong b + c$ (not a triangle). And our test criteria are able to detect such errors. In fact, so detailed is the specification for this example that a test set based on the problem partition alone would have served equally well.

In spite of the obvious relevance of the ideas presented here, particularly those of Weyuker and Ostrand, a good deal of work remains to be done to apply the theory to a wide class of programs. One of the more important tasks is to find more systematic methods for constructing the problem partition. This will not be easy, since finding a good problem partition is quite similar to the task of creating the program itself. It is suggested, however, that the development of formal specification languages would be helpful here, particularly if such developments are made with a specification-driven testing methodology in mind, along the lines presented here.

An equally important consideration when thinking of applying the above theory to larger programs is that of obtaining the domains of the path partition. How are the paths to be described, generated, and selected with programs of increasing size and complexity? It is certainly clear that our one example is misleading in this respect. We had only a small number of paths to consider, whereas a typical program of any size will have a very large number of paths, most likely an infinity of paths, owing to the presence of loops. If our test is still to be finite, how do we then choose paths judiciously? How are they described? And most importantly, how do we generate test cases that will traverse these paths, if indeed this is possible? These are some of the questions that we begin to address in the following sections.

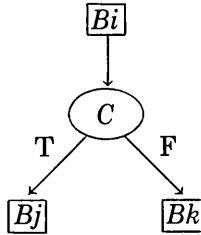
III. GENERAL THEORY—THE STRUCTURAL APPROACH

In a *structural approach* to the theory of testing, a program F is represented by a “skeleton” of its underlying flowchart, a directed graph symbolizing the flow of control. This point of view is advanced most effectively in the extremely lucid survey paper by Huang.¹² We should keep in mind, however, that the flowchart graph must be an accurate representation of the program flow in the code itself. Ordinarily, this is ensured through the use of a “tool” that automatically generates the flowgraph from the source program listing.

Using Huang’s terminology,¹²⁻¹⁴ a *program block* is a maximal sequence of program statements having the property that if the first member of the sequence is executed, then all other statements in the sequence will also be executed. A *program graph* $F = (V, E)$ is then a directed graph with vertex set V and edge set E , where each vertex is associated with a program block and in which there are pairs of edges:

- (i, j) labeled by the condition C
- (i, k) labeled by the condition $\sim C$

according as the flowchart segment:



encountered for blocks B_i , B_j , and B_k . (For convenience, we permit an empty block as a vertex in good standing, e.g., for treating an “if . . . then . . .” statement with vacuous “else” clause.) It is further assumed that the graph has a single entry point, the *start vertex*, and a single exit point, the *stop vertex*, and that every vertex lies on some path from ‘start’ to ‘stop.’

A *path* in a (program) graph is defined in the usual way, as a sequence of edges

$$p = e_1, e_2, \dots, e_n,$$

though we ordinarily assume as well that we begin the sequence at ‘start’ and end at ‘stop.’ Each such path has an associated *path predicate*

$$P = P_1 \wedge \dots \wedge P_n$$

written as a conjunction of the individual interpreted branch condition labels on the edges e , as discussed below (see Section V). The path predicates P are to be identified in one-to-one correspondence with the path domains of the previous section. Thus we may write (somewhat ambiguously):

$$D = \cup P \quad \text{for} \quad P = \{d \text{ in } D : P(d)\}$$

so that the program function $F : D \rightarrow R$ is a union of functions $F(P) : P \rightarrow R$ restricting F to the individual path domains P .

In structured testing, we examine the program (as a digraph) and we seek to choose a finite set of paths that will cover the program with a certain degree of thoroughness. It is then hoped that test data causing the program to be successfully executed when traversing these paths are sufficient to warrant our confidence in the program’s correctness. The theoretical underpinnings of such a testing plan have been studied by Howden,¹⁵⁻¹⁸ who relates his work to the earlier study by Goodenough and Gerhart.⁷ Rather than speaking of a “test criteria,” however, Howden refers to a *testing strategy*, as a uniform computable function

$$(F : D \rightarrow R) \xrightarrow{H} (T, \text{subset of } D)$$

that associates with each program F a finite test set T of D . H is said to be an *ideal strategy* if each $T = H(F)$ is an ideal test (for every program F). As is so often the case with testing theory, the first result is of a negative character:

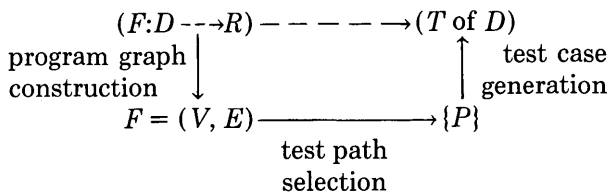
Theorem (Howden): An ideal testing strategy does not exist.

Nevertheless, Howden was able to show that “path testing” can be a reliable approach, at least for detecting certain types of errors. He takes the view that the program being treated is a member of a class of programs differing only as to whether they are correct, and for which the incorrect programs have errors of various (known) types. His objective was then to find, if possible, a restricted set of programs for which certain forms of structured testing (i.e., path testing) would be reliable. Typical of Howden’s results is that which assumes that the error in a program does not change its control flow, i.e., that the set of path domains is not affected.

Theorem (Howden): Path testing is a reliable method for distinguishing correct from incorrect programs, as long as the errors of incorrect programs do not affect the path partition.

Of course, there are theoretical limitations in applying results such as this, since Howden has in mind our choosing one test datum from each path domain P , and these may be infinite in number. On the other hand, he has also devised a classification of error types that can be expected to lead to new insights into the testing problem generally. The reader should consult Howden’s work (particularly Refs. 16 and 17) for further detail.

In a practical test setting, we require that the subset T of D be finite. Moreover, if we are speaking of a “path testing strategy,” the above schema will be decomposed into the three-stage process,



summarized as follows:

1. Program graph construction
2. Test path selection
3. Test case generation.

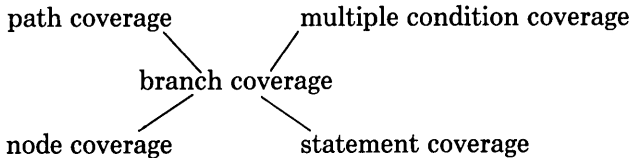
The first phase of the process, to construct the program graph from a source code listing, is fairly straightforward, and for most of the conventional programming languages, e.g., FORTRAN, Pascal, etc., such implementations are already in existence.

As a matter of fact, implementations of testing tools are in various stages of development for treating the entire process outlined above (e.g., see Clarke¹⁹). But, as we shall see, there are serious problems associated with the latter stages of any proposed implementation along these lines. There are many alternative strategies to choose from, and seemingly, none of these is best for all situations. All we can do at this point is to outline the several alternatives and comment on their general suitability. We begin by introducing the various path selection criteria, continuing this discussion in the next section. The last, and perhaps the most difficult, of our three subprocesses, the generation of test data, is treated in Section V.

There are, as we have indicated, a number of path selection criteria that can be used in attempting to devise a testing strategy that will provide a reasonable coverage of a program graph. Among these criteria are:

1. *Statement coverage*: Execute all statements (blocks) in the graph.
2. *Node coverage*: Encounter all decision node entry points in the graph.
3. *Branch coverage*: Encounter all exit branches of each decision node in the graph.
4. *Multiple condition coverage*: Achieve all possible combinations of condition outcomes at each decision node of the graph.
5. *Path coverage*: Traverse all paths in the graph.

These five strategies are related in their strength of coverage as shown below:



with the weaker criteria at the bottom and the stronger criteria at the top.

As an example illustrating the differing requirements of these criteria, consider the flowchart segment (program graph) shown in Fig. 2. In order to achieve node coverage, the single test:

$$abe: A = 2, B = 1, X = 1$$

will suffice (but it will not achieve statement coverage because the assignment $X \leftarrow X/A$ will not have been executed). On the other hand, the single test:

$$ace: A = 2, B = 0, X = 3$$

will be sufficient for complete statement coverage (and node coverage

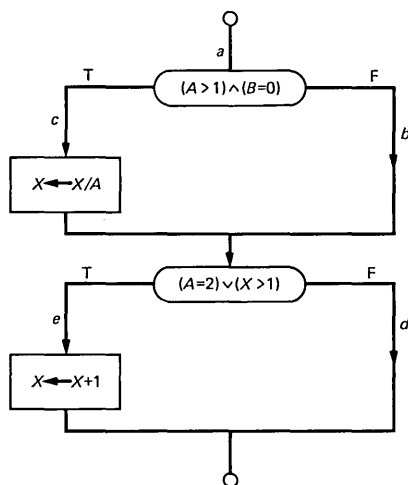


Fig. 2—Flowchart segment illustrating coverage criteria.

as well). For branch coverage, however, at least two tests would be required, e.g.,

$$\begin{aligned}acd: A = 3, B = 0, X = 3 \\abe: A = 2, B = 1, X = 1.\end{aligned}$$

In the multiple condition coverage criterion, there are $2 \cdot 2 + 2 \cdot 2$ or 8 outcomes to achieve in combination for the two simple conditions at each decision node. These may be satisfied, for example, by the selection of four separate tests, e.g.,

$$\begin{aligned}ace: A = 2, B = 0, X = 4 \\abe: A = 2, B = 1, X = 1 \\abe: A = 1, B = 0, X = 2 \\abd: A = 1, B = 1, X = 1.\end{aligned}$$

The first test satisfies the conditions $A > 1, B = 0$ in the first decision and $A = 2, X > 1$ in the second decision. The second test ensures that $A > 1, B \neq 0$ for the first decision and $A = 2, X \leq 1$ for the second decision. Further analysis shows that all eight combinations are achieved. For the path coverage criterion to be met, we again require four tests, e.g.,

$$\begin{aligned}ace: A = 2, B = 0, X = 4 \\acd: A = 3, B = 0, X = 3 \\abe: A = 1, B = 0, X = 2 \\abd: A = 1, B = 1, X = 1.\end{aligned}$$

Note, however, that this test set would not satisfy the multiple condition coverage criterion.

It is clear that “statement coverage” and “node coverage” are in themselves rather weak strategies for testing, representing necessary but by no means sufficient criteria for a reasonable structural test. The “branch coverage” criterion, however, implies the other two (as seen in the diagram above) and has come to be regarded as a minimal standard of achievement in structure-based testing. The stronger criteria of “multiple condition coverage” and “path coverage” are difficult to achieve in a program of any complexity. In fact, the path testing criterion is usually relaxed to the extent that only “equivalence classes” of paths are represented. In a program of any size, particularly in the presence of program loops, there is a virtual infinity of paths through the program graph. Two paths are then considered “equivalent” if they differ only in their number of loop traversals. One then chooses only one representative from each such equivalence class in devising a test set. But still, this *modified path coverage* criterion is difficult to achieve in practice.

A survey of the literature shows that there is little common agreement as to what would be considered as an ‘adequate’ structural test criterion. As we have noted, the “branch coverage” criterion has been widely recognized as a basic measure of testing thoroughness. This is evidenced by the fact that most of the major software testing tools in existence or in development do indeed include some provision for achieving this particular test goal. The disagreement seems to be in deciding how much more (or less) is needed beyond this basic requirement to entitle a structural testing strategy to be considered adequate.

If total branch coverage is indeed used as a measure of testing thoroughness, a simple calibration scheme can be invoked, using a set of *software counters*. One “prepares” the program for testing by inserting counters at appropriate points in a modified copy of the program, and after running through the test set, one can determine the degree of thoroughness from a listing of the resulting counter values. This is the method of *test instrumentation*. We first define a *decision to decision (DD) path* of a program to be a sequence of a statements leading from a decision box (or the “start” node) to a decision box (or the “stop” node), having no intervening decisions. To determine whether every branch of our program has been encountered at least once (branch coverage) in our testing, it is sufficient to insert a counter at the ‘head’ of each DD path.

Consider the classical flowchart solution (Fig. 3) to the problem of computing $z = x$ to the power y . Here, there are five DD paths:

abc, d, efhi, gfhi, jk

and we therefore insert our software counters at the points *a, d, e, g,*

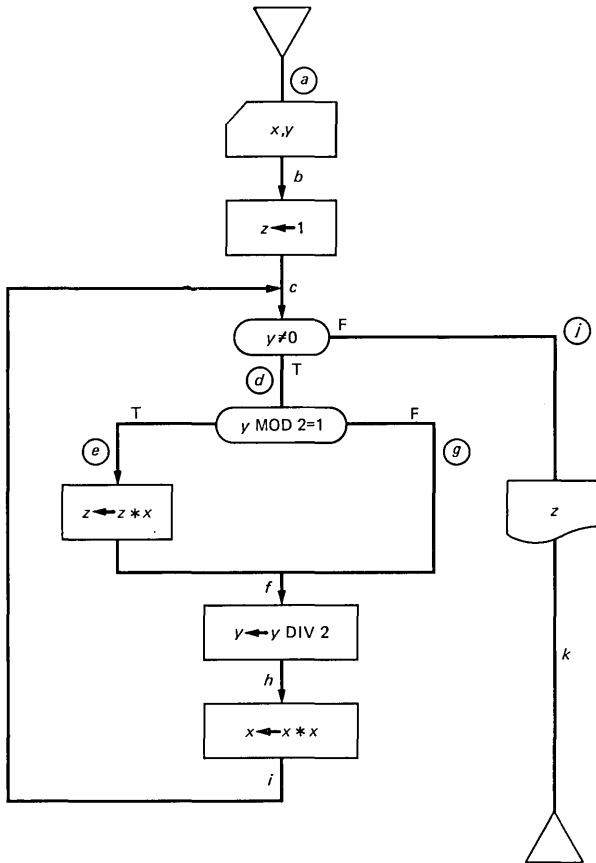


Fig. 3—Flowchart for computing x^y .

j , as shown. If we have run two test cases as shown in the table below,

x	y	a	d	e	g	j
10	0	1	0	0	0	1
20	1	1	1	1	0	1

we would find that 'counter g ' has not yet been activated. Inspection of the flowchart then shows that we need a test to traverse the path a, b, c, d, g , etc., requiring $y \neq 0, y \bmod 2 \neq 1$. So we may use $x = 24$ and $y = 2$, say, as an additional test case, thus ensuring complete branch coverage. Ideally, this latter phase, directing the tester to the area of untested code, would also be automated.

Of course, we would like to automate as much of the testing methodology as possible, recalling the three-stage process mentioned earlier. On the other hand, in lieu of a complete mechanization, the testing

instrumentation scheme presented here can be of great help in isolating areas in need of further testing. Furthermore, it can be argued that for the little extra cost entailed, it is a worthwhile investment in any testing process, fully automated or not. [Parenthetically, we might note as an indication of the expense associated with the development of testing tools generally, that a package that does little more than "test instrumentation," as described here, has been announced recently (Computer, May 1982) by Management and Computer Services, selling for \$12,000.00!.] It may be that some other criterion than "branch coverage" is being used as a measure of test thoroughness. It is still good practice to be concerned as to what extent this standard measure is being met. Moreover, it is reasonable to suppose that the "instrumentation concept," as exemplified here, might generalize to settings where other thoroughness criteria are being used.

IV. TEST PATH SELECTION

As we have indicated, there are a number of criteria that can be used in selecting program paths to achieve an adequate testing coverage. But the question then becomes: How do we automatically generate a collection of paths meeting a given criterion? The literature is somewhat "hazy" on this point. Perhaps the most explicit treatment of the problem is that of Paige,²⁰⁻²² in reference to programs built up from a strict adherence to the structured programming methodology. In fact, we know of no more general approach to the problem, one that would handle structured or unstructured code in relation to the whole spectrum of path selection criteria.

Recall that a *structured program* F is one that has been built up inductively from certain "simple statements" as a base (typically, the assignment, input and output statements, and procedure calls), using only the three familiar constructs:

1. *Sequence*: begin P_1 ; P_2 ; \dots ; P_n end
2. *Selection*: if C then P else Q
3. *Repetition*: while C do P

for structured (but possibly themselves compound) statements P_i , P , Q , respectively. The resulting program graph $F = (V, E)$ is then of a correspondingly restricted form, greatly facilitating the path analysis problem. Perhaps collapsing sequences of simple statements to a single block (graph node), one may then use a "regular expression" $r(F)$ to characterize the program flow, associating

1. The '.' operator to sequences
2. The '+' operator to selections
3. The (Kleene) '*' operator to repetitions,

respectively.

For example, if F is the (structured) program graph shown in Fig.

4, we have

$$r(F) = a(b(d + e)(k + 1) + c(f + g(h(i + j))^*m))$$

as the corresponding regular expression. Note the loop $(h(i + j))^*$ resulting from a “while” statement.

We have mentioned earlier, in reference to the modified path coverage criterion, how an equivalence relation is often used to obtain a finite representation of the path alternatives in the presence of loops. Accordingly, if we make the substitution $x^* = x + 1$ ($1 = \text{null}$) in the regular expression $r(F)$, we acknowledge that a loop is either executed or not. Multiplying out so as to obtain a “sum of products” expression, one then obtains the desired collection of paths satisfying the modified path coverage criterion, e.g.,

$abdk \quad acf$
 $abdl \quad acghim$
 $abek \quad acghjm$
 $abel \quad acgm$

in reference to the program graph of Fig. 4. On the other hand, it does not appear that this technique can be extended to handle unstructured programs.

But if we continue to deal with a structured program graph, we can describe a method for deriving a minimum number of paths sufficient to meet the “branch coverage” criterion. We assign a set of paths $S(r)$ to each regular expression $r = r(F)$ inductively, as follows. We let $S(a) = \{a\}$ for each simple statement a , and then, assuming that $S(r)$

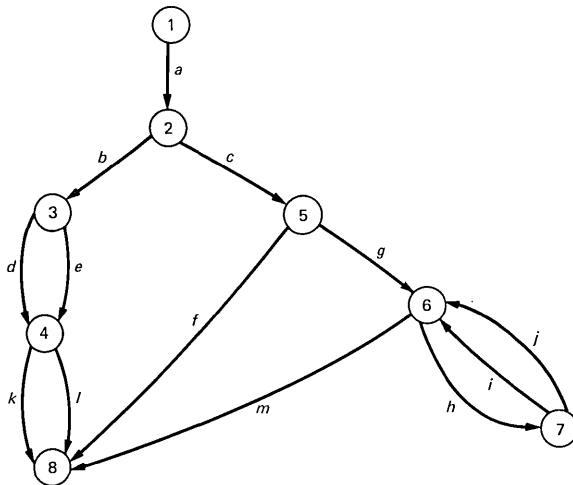


Fig. 4—Structured program graph illustrating modified path coverage criterion.

and $S(t)$ have been defined, for regular expressions r and t , we set:

1. $S(r \cdot t) = S(r) \cdot S(t)$
2. $S(r + t) = S(r) \cup S(t)$
3. $S(r^*) = S(r)^*$.

Here, $S(r)^*$ is the singleton set obtained by concatenating (in any order) all of the paths in $S(r)$, and similarly, the set product $S(r) \cdot S(t)$ is obtained by concatenating paths in $S(r)$ with those in $S(t)$ —but retaining only enough products so that each of the factors from $S(r)$ and $S(t)$ are represented. It follows that

1. $|S(r \cdot t)| = \max\{|S(r)|, |S(t)|\}$
2. $|S(r + t)| = |S(r)| + |S(t)|$
3. $|S(r^*)| = 1$.

By way of illustration, in considering once again the example from Fig. 4, we may compute:

$$\begin{aligned} S(i + j) &= \{i, j\} \\ S(h(i + j)) &= \{hi, hj\} \\ S((h(i + j))^*) &= \{hihj\} \\ S(g(h(i + j))^*m) &= \{ghihjm\}, \end{aligned}$$

etc., and finally,

$$S(r) = \{abdk, abel, acf, acghihjm\},$$

yielding four paths that together cover all of the branches of the program.

Once again, as in the case of the previous algorithm, there seems to be no easy extension of this technique that would handle unstructured programs as well. However, a general *upper bound* is readily available regarding the number of paths necessary for total branch coverage. Whether our program is structured or not, we make the observation that if a test path reaches a particular node of the program graph, then it must exit this node through one of the (two) branches leaving the node. If the graph F has e edges and n nodes, it follows that

$$v(F) = e - (n - 2) = e - n + 2$$

is an upper bound on the number of paths necessary to achieve total branch coverage. Coincidentally, this is the formula for McCabe's *cyclomatic complexity measure*,²³ a figure that has proved to be useful in estimating overall "program complexity". At the same time, the graph theoretic derivation of a program's "independent" circuits (paths) yields a branch covering of paths, $v(F)$ in number—though generally somewhat in excess of the minimum number of paths that would be required. In the context of our running example, we have

$\nu(F) = 13 - 8 + 2 = 7$ and a corresponding set of *basis paths*:

acf *acgm*
abdk *acghim*
abek *acghjm* .
abel

Note that the single path *abdl* from our “path coverage” list that is not present here is itself a linear combination of paths already listed. Note as well that we obtain seven paths here, whereas we know from the preceding analysis that four paths will suffice (for branch coverage).

The whole notion that McCabe’s *basis of program paths* should constitute a goal of program testing has attracted considerable attention, and we feel obliged to comment on this point. Perhaps this is best done by listing what we think are the pros and cons to the approach. On the positive side, we cite the following:

1. The method is sufficiently general as to be applicable to both structured and unstructured programs.
2. The resulting “basis” does indeed ensure total branch coverage.
3. The paths of a basis are feasibly computable, using standard graph theoretic techniques.

On the other hand, these aspects must be counterbalanced with the following:

1. A single basis is not uniquely determined—there are many, and we must make a choice.
2. The number $\nu(F)$ of paths in a basis can greatly exceed the minimum number of paths required to achieve branch coverage.
3. The notion that, in some sense, *every* path in the program graph is accounted for by our having selected a basis is somewhat specious. Note that we do not comment here on the inadequacies of McCabe’s $\nu(F)$ as a measure of overall program complexity—we leave this discussion to a separate paper. On the other hand, the arguments for and against the use of the associated “basis of program paths” as a testing strategy are inconclusive at best, particularly in comparison with the “level paths” of Paige^{21,22} that we now describe.

In a program graph $F = (V, E)$, a *level-0 path* is a simple (acyclic) path from “start” to “stop”. In effect, these paths trace the “fall through” conditions in the program. Then, inductively, a *level- i path* ($i > 0$) is a simple path (perhaps a circuit) that begins and ends on nodes of a path of lower level, but has none of its other nodes previously appearing on paths of a lower level. Intuitively, the level- i paths for $i > 0$ account for program loops of increasing nesting level and for feedback paths, etc., in the case of an unstructured program.

Considering our earlier structured program graph (Fig. 4) and the

unstructured program graph of Fig. 5, we tabulate the respective level- i paths as shown in Tables II and III.

In any case, we are able to say that a given level- $(i + 1)$ path is "associated with" a certain level- i path according as the given path begins and ends on nodes of the parent path. This relationship orders the level paths in a tree-like structure, in such a way that one can readily construct test paths that again effect a total branch coverage. In so doing, only level paths that associate can be combined to form a program test path. Thus, for example, in the case of the program graph of Fig. 4 above, we may construct the path $acghihjm$ as the linear combination:

$$acghihjm = (6) + (7) + (8)$$

using the notation in Table II.

It is clear that the level paths of a program graph span the space of

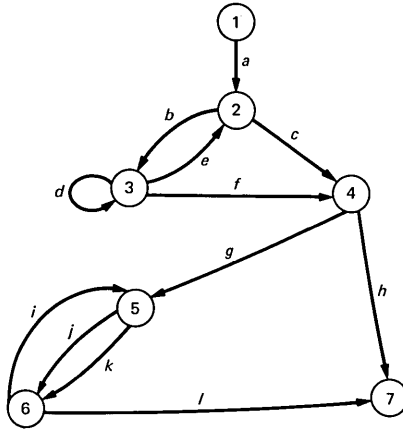


Fig. 5—Unstructured program graph.

Table II—Level- i paths for structured program graph (see Fig. 4)

Level	Level Paths	$V[i]$	$E[i]$
-1		{1, 8}	0
0	(1) $abdk$ (2) $abdl$ (3) $abek$ (4) $abel$ (5) acf (6) $acgm$	{2, 3, 4, 5, 6}	{a, b, c, d, e, f, g, k, l, m}
1	(7) hi (8) hj	{7}	{h, i, j}

Note: The sets $V[i]$ and $E[i]$ of vertices and edges at level- i are useful in the computation of the level- $(i + 1)$ paths.

Table III—Level- i paths for unstructured program graph
(see Fig. 5)

Level	Level Paths	$V[i]$	$E[i]$
-1		{1, 7}	0
0	(1) <i>acgjl</i> (2) <i>acgkl</i> (3) <i>abfgjl</i> (4) <i>abfgkl</i> (5) <i>ach</i> (6) <i>abfh</i>	{2, 3, 4, 5, 6}	{ <i>a, b, c, f, g, h, j, k, l</i> }
1	(7) <i>d</i> (8) <i>e</i> (9) <i>i</i>	0	{ <i>d, e, i</i> }

Note: The sets $V[i]$ and $E[i]$ of vertices and edges at level- i are useful in the computation of the level- $(i + 1)$ paths.

program paths. But taken together, they do not usually constitute a basis. Thus, again in Fig. 4 above, we have eight-level paths, whereas we know from our previous analysis that this graph has rank $v = 7$. On the other hand, Paige's level paths have a definite uniqueness, an advantage over the notion of a basis as developed by McCabe, and leading to a graduated *level path testing strategy* as follows:

1. First test all level-0 paths—in effect, keeping all loops in the “nonexecuting” mode.
2. Next test all level-1 paths, reaching them through their associated level-0 paths, etc.

The result is a highly structured testing strategy where segments of the program are treated in successive layers of nesting depth.

The level path testing strategy provides for a rather exhaustive treatment of a program's path structure at successive depths of nesting. In this sense, the approach has a potential thoroughness rivaling that of the “modified path coverage” criterion. On the other hand, Paige's strategy is readily applicable to both structured and unstructured programs. At the same time, his method lends itself to a convenient algorithmic solution,²² though one must be prepared to compute all simple paths (or circuits) between various identified pairs of nodes, along edges not previously used—most likely requiring the use of a “depth first search” strategy. Except for this computational difficulty, the approach is quite orderly; it provides for a more thorough testing than simple branch coverage, and it compares favorably against McCabe's “basis of program paths” in that:

1. The level paths are uniquely determined.
2. The number of level paths will exceed $v(F)$.
3. The notion that somehow every path in the program graph is accounted for by our successive treatment of its levels has a good deal more credibility.

In conclusion, it must be noted that all of the methods we have discussed for selecting program test paths are subject to one overriding criticism. There is absolutely no assurance that the paths selected will be *feasible*, i.e., executable with an appropriate choice of input data. We suggest that this problem becomes more serious (and is surely more difficult to analyze) in the case of paths selected in an attempt to minimize the number required for branch coverage. Paige's "level path" strategy would seem to be easier to handle in this respect, since we build up paths from the simple to the more complex, starting with those that are more likely to be feasible.

V. TEST CASE GENERATION

The whole question of path feasibility is related to the "test case generation" problem. This is the one remaining phase to be discussed of the three that were outlined in the rectangular problem-decomposition paradigm of Section III. Considering the question of feasibility, however, we can see that it is difficult to so trichotomize the automation of the overall testing program. Though useful as a paradigm, we must admit that this partition of the problem is overly idealistic in relation to the real world of program testing that we are likely to encounter.

Suppose we have selected a set of program paths because they meet one or another of the test coverage criteria, or for whatever reason. There still remains the problem of generating corresponding test cases that will drive the program through the indicated paths. This again turns out to be a nontrivial (and in some cases, unsolvable) problem. All we can do at this point is to summarize the approaches that have been taken by researchers in the field and to give a few suggestions that might aid in developing a workable methodology.

Perhaps the most comprehensive treatment of the problem is that of Clarke.^{19,24,25} Consider a single path p from "start" to "stop" through a program $F = (V, E)$, again viewed as a directed graph. We intend to show how p may be characterized as a *path predicate*, i.e., a logical condition

$$P = P_1 \wedge P_2 \wedge \dots \wedge P_n$$

expressed as a conjunction of "interpreted" branch conditions derived from the labels on the edges of F . Inductively, we may think of p as having developed as a sequence of "partial paths:"

$$p(k) = (v[0] = \text{start}, v[1], \dots, v[k])$$

leading from "start" to some intermediate vertex $v(k)$ on the way to

“stop”. Correspondingly, we may give an inductive derivation of the path predicate, writing $P(0) = \text{true}$ and

$$P(k) = P(k - 1) \wedge \text{ibp}(v[k - 1], v[k]),$$

where the latter conjunct is the *interpreted branch predicate* associated with the transition from vertex $v[k - 1]$ to $v[k]$. More precisely, $\text{ibp}(e)$ for an edge e labeled with the Boolean condition C will be computed by substituting (in C) the current “symbolic values” of all variables according to their updating along the partial path $p(k)$.

For example, consider the flowchart solution (Fig. 6) for estimating the point where a function f takes on its maximum value. For the path

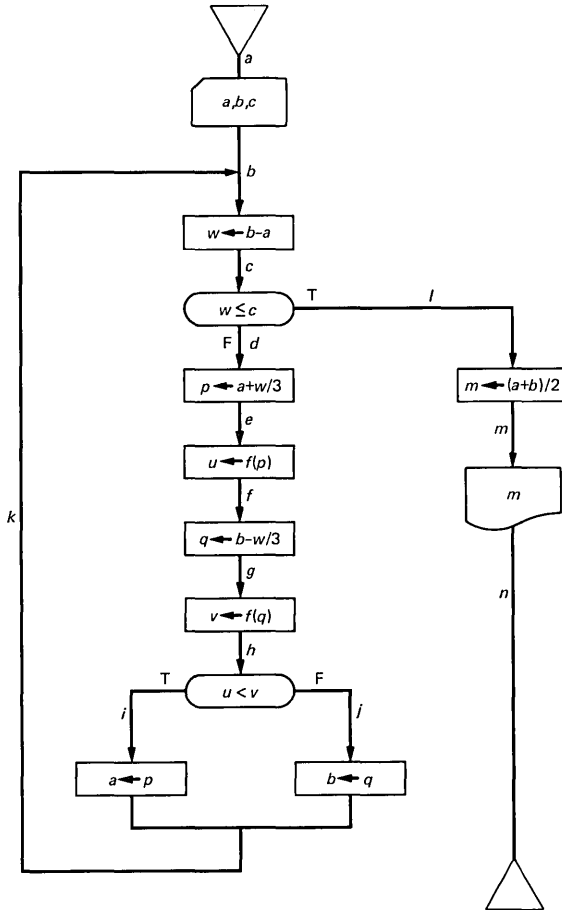


Fig. 6—Flowchart for estimating the point where a function is maximized.

$p = abclmn$, we compute

$$\begin{aligned}P(0) &= \text{true} \\P(1) &= P(0) \wedge \text{true} = \text{true} \\P(2) &= P(1) \wedge \text{true} = \text{true} \\P(3) &= P(2) \wedge (b - a \leq c) = (b - a \leq c) \\P(4) &= P(3) \wedge \text{true} = (b - a \leq c)\end{aligned}$$

and finally,

$$P = P(5) = P(4) \wedge \text{true} = (b - a \leq c),$$

noting that it was necessary to substitute $a - b$ for w in the condition for traversing edge 1 because of the earlier assignment statement.

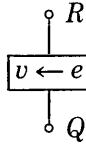
In general, this process of continually updating the symbolic values of program variables as we proceed along a path is called *symbolic execution* (or *symbolic evaluation*). The data descriptions generated in symbolic execution provide a precise representation of the changing *program state*. Initially, the program state is the three-place vector:

$$\begin{aligned}\text{state} &= [\text{start}, \text{values}(\text{start}), \text{pathpred}(\text{start})] \\&= (\text{start}, (\perp, \perp, \dots, \perp), \text{true}),\end{aligned}$$

where “values” tabulates the symbolic values of all program variables (\perp = undefined), and “pathpred” stores the inductively generated path predicate P as described earlier. Symbolic names are assigned (in “values”) to input variables whenever a read statement is encountered on the program path. Throughout the symbolic evaluation, all symbolic representations of variable and branch predicate values are then expressed in terms of these symbolic names, as representatives of input values. In particular, as one encounters an assignment statement ($v \leftarrow e$), the symbolic value of the program variable v is updated (as in the example above) through substitution of the symbolic value of the expression e . In this way, “state” and especially the “values” vector will provide a continually updated snapshot of the program’s development along the path. Moreover, the final value of the “pathpred” component of “state” provides the logical conjunction described above.

This path predicate P defines a corresponding (path) subdomain of the input space D , and by the nature of the symbolic evaluation technique, P is expressed as a set of conditions on the input variables alone. To generate a test case (of input data) that will cause the program to traverse the path p , it is then only necessary to find input values that satisfy all of these conditions. As we might expect, however, this is often easier said than done.

Before discussing this problem in any detail, it is better that we first describe an alternative to the above approach, one that proceeds in reverse—from the end of the path to its beginning. This technique, known appropriately as *backward substitution*, is best described in the survey paper by Huang.¹² To traverse a path p , certain conditions must be met, i.e., the set of branch conditions (C or $\sim C$) along the path must be satisfied as they are encountered. On the other hand, suppose that an assignment statement ($v \leftarrow e$) intervenes, between “start” and the predicate Q , the latter representing a given branch condition (though modified by “partial backward substitution” as we are now describing). In the following flowchart segment:



if we want Q to be true after the assignment ($v \leftarrow e$) has been executed, then the predicate $Q(v \leftarrow e)$ must be satisfied prior to its execution. Here, $Q(v \leftarrow e)$ is the predicate obtained by substituting the expression e for each occurrence of v in Q [and we speak of $Q(v \leftarrow e)$ as the predicate obtained by *dragging Q backward* through the indicated assignment statement]. It follows that the conjunction

$$R \wedge Q(v \leftarrow e)$$

is necessary for our passage along the edge with condition R (through the assignment) and then to satisfy Q .

Altogether, if we want the specific path p to be traversed in a program’s execution, then we must drag each of its edge conditions backward to “start”, and the conjunction of all resulting predicates must be satisfied by the corresponding test case of input data. Note once again that we obtain in this way a corresponding path predicate:

$$P = P1 \wedge P2 \wedge \dots \wedge Pn,$$

i.e., a conjunction of modified branch conditions, each expressed in terms of the input variables to the program.

Consider once again the example of Fig. 6, and suppose we wish to traverse the path $abcdefghiklmn$. The listing shown below traces the dragging of the three necessary branch conditions backward along this

path:

<i>l</i>	$w \cong c$		
<i>c</i>	$w \cong c$		
<i>k</i>	$b - a \cong c$		
<i>i</i>	$b - p \cong c$	$u < v$	
<i>h</i>	$b - p \cong c$	$u < v$	
<i>g</i>	$b - p \cong c$	$u < f(q)$	
<i>f</i>	$b - p \cong c$	$u < f(b - w/3)$	
<i>e</i>	$b - p \cong c$	$f(p) < f(b - w/3)$	
<i>d</i>	$b - (a + w/3) \cong c$	$f(a + w/3) < f(b - w/3)$	$\sim(w \cong c)$
<i>c</i>	$b - (a + w/3) \cong c$	$f(a + w/3) < f(b - w/3)$	$\sim(w \cong c)$
<i>b</i>	$b - a - (b - a)/3 \cong c$	$f(a + (b - a)/3) < f(b - (b - a)/3)$	$\sim(w \cong c)$
<i>a</i>	$b - a - (b - a)/3 \cong c$	$f(a + (b - a)/3) < f(b - (b - a)/3)$	$\sim(w \cong c)$

One finally obtains the conjunction of three predicates:

$$\begin{aligned}
 P1 &: b - a - (b - a)/3 \leq c \\
 P2 &: f(a + (b - a)/3) < f(b - (b - a)/3) \\
 P3 &: \sim(b - a \leq c)
 \end{aligned}$$

all expressed in terms of the inputs a, b, c (and the “called” function f). For purposes of comparison, the reader may try to compute an equivalent predicate using the symbolic execution method.

In an overall comparison of these two methods, one can identify an obvious “trade-off.” With backward substitution, we avoid the costly storage facility needed for the continuous updating of all the symbolic program variable values. On the other hand, an important advantage accrues to the symbolic execution method, one that is not available for the backward substitution technique. Namely, we are more easily able to determine whether a given path is (or will be) feasible. And we can make the determination early in the symbolic evaluation. We need only check that the inductively generated predicates $P(k)$ are noncontradictory, as far as they go. We begin with $P(0) = \text{true}$ —certainly there is no contradiction here. Then, in the expression for $P(k)$ in terms of $P(k - 1)$, we have only to see whether $ibp(v[k - 1], v[k])$ contradicts $P(k - 1)$. If so, $P(k)$ and hence P itself is contradictory, and the path p is infeasible. Otherwise, we keep going. Note, in comparison, that with the backward substitution method, we wouldn’t know whether a path was feasible until all of the calculation (of P) was completed—a definite disadvantage.

One must note, however, that all such “logical satisfiability” problems as we are now beginning to consider are exceedingly difficult to handle in practice. We include here the satisfiability question that results from the use of the “backward substitution” technique or the forward “symbolic evaluation” method, whichever is used. At the

conclusion of the backward substitution, we have a system of constraints on the inputs to the problem, and unless these constraints can be “solved” for the input data, we don’t have a test case at all. The same may be said for the forward symbolic execution, except for the slight advantage that we can be determining the satisfiability (or lack thereof) as we go.

Huang, in his survey paper,¹² presents a systematic approach for handling the satisfiability problem, and we now outline the major features of his plan. The simplifying assumption is made that the path predicate takes the form:

$$P = P1 \wedge P2 \wedge \dots \wedge Pn,$$

where the Pi are nonnegated atomic expressions:

$$d R e$$

with d, e arithmetic expressions in the input variables and R one of the six relational operators: $<, \leq, =, \langle \rangle, \geq, >$. Such a system of atomic logical expressions can readily be rewritten in the *prenex normal form*:

$$(Ex1)(Ex2) \dots (Exn)(x1 = e1) \wedge (x2 = e2) \wedge \dots \wedge (xn = en),$$

where the E ’s are “existential quantifiers” on auxiliary variable x ’s, and the new expressions (the e ’s) are differences of d, e above, sufficient to transform any inequalities to equalities. The inequalities are, in effect, shifted to the auxiliary variables, thereby serving to normalize the solution space. Thus, in place of the inequality $2(b - a)/3 \leq c$ at the end of the table above, we would have $(Ex1 \geq 0)[x1 = c - 2(b - a)/3]$. Altogether, the three inequalities of that problem are similarly transformed, and we have instead the prenex normal form:

$$(Ex1 \geq 0)(Ex2 \geq 0)(Ex3 > 0)$$

$$x1 = c - 2(b - a)/3$$

$$x2 = b + 2a - 6$$

$$x3 = b - a - c,$$

one that is somewhat easier to handle.

From this point, standard techniques of linear algebra can be used to further transform the system into one where a minimum number of variables are involved. Thus, in the case of our running example, we can simplify the system so as to finally obtain:

$$(Ex1 \geq 0)(Ex2 \geq 0)(Ex3 > 0)$$

$$3x1 - x2 + 3x3 = 6 - 3a.$$

From here, one may “guess” a solution, e.g., $x1 = x2 = 0$ and $x3 = 0.1$

say. One thereby obtains:

$$\begin{aligned}a &= 1.9 \\b &= 2.2 \\c &= 0.2,\end{aligned}$$

an input set that will cause the program to traverse the path *abcdefghijklmn* in Fig. 6, as originally required.

If we are going to have to “guess” a solution to the feasibility question in the end, however, the outright “trial and error” approach of Ramamoorthy et al.²⁶ offers an attractive alternative. One makes the assumption, as before, that the path predicate P is expressed as a logical conjunction:

$$P = P_1 \wedge P_2 \wedge \dots \wedge P_n,$$

where each of the P_i is a constraint on the program’s input variables. Moreover, it is assumed that the input variables have been ordered as $v[1], v[2], \dots, v[m]$. With each variable $v[i]$, we associate a set $S[i]$ of conjuncts from P , namely:

$$S[i] = \{P_j : \text{only } v[1], \dots, v[i] \text{ occur in } P_j\}$$

and these sets are then used as the basis for the “trial and error” algorithm shown in Fig. 7.

Assuming that values have been found for $v[1], \dots, v[i - 1]$ satisfying all the conjuncts in $S[1], \dots, S[i - 1]$, we either solve for $v[i]$ or randomly choose $v[i]$, depending on whether the set $S[i]$ contains an equality relation in $v[i]$. We then substitute this value in the conjuncts of $S[i]$. Should we thereby arrive at a contradiction, we “backtrack” to the iteration $i - 1$, generating a different value for $v[i - 1]$. Otherwise, we go ahead to the iteration $i + 1$. If the complete iteration on i concludes successfully, we arrive thereby at a “satisfiable” test case for the input variables of the program; otherwise we do not. Note that the “key” to the method is the fact that at each stage i , only the variable $v[i]$ has not yet been resolved. Note, however, that the loop at the right of Fig. 7 must include some criterion for deciding that “enough” random numbers have been tried in the current iteration. But however this is decided, it must be conceded that such an approach as presented here has much to offer in its favor, particularly considering the difficulty of the “satisfiability” question in general.

The authors²⁶ provide an example of the use of their algorithm on the “triangle classification problem” considered earlier. More generally, they suggest that the method has proven to be successful in treating a much wider class of problems. We would note further that the method could conceivably be applied to the “running satisfiability” questions that arise in the use of the “symbolic execution” technique.

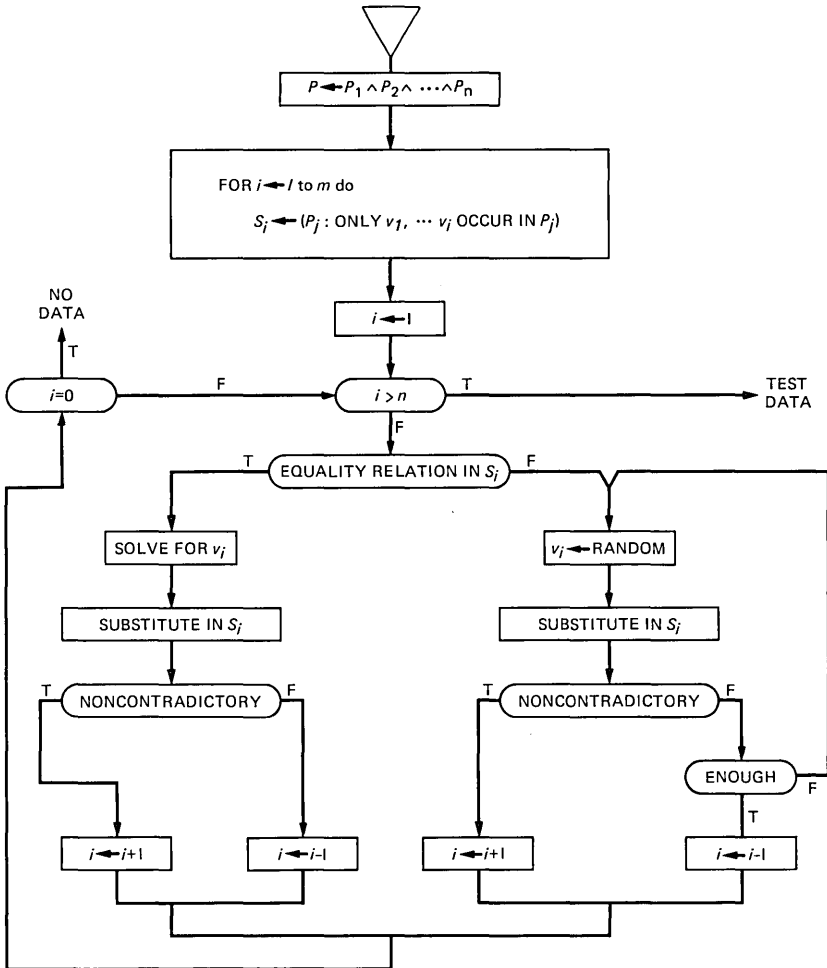


Fig. 7.—Trial and error algorithm for solving the satisfiability problem.

In fact, it seems that this “trial and error” approach has a definite place—at least as a method of last resort, to be used as a component of any overall testing methodology.

Without some technique such as this, we are forced to rely on the extremely costly and not wholly reliable methods of “mathematical programming,” particularly those routines that are designed to generate solutions to systems of inequalities. We cannot always assume that our systems are linear, in spite of the assumptions made by some authors. And in the absence of such an assumption, the problem is quite a difficult one, generally beyond the capability of the packages that are currently available.

Recognizing this, a most unusual and quite promising approach has been suggested by Kundu.²⁷ The idea is to combine the “test path selection” and “test case generation” phases of the solution, using the previous test case(s) $t[k]$ to help in determining the next test case $t[k + 1]$. The result is a sequence of determinations:

$$(t[0] \rightarrow) p[0] \rightarrow t[1] \rightarrow p[1] \rightarrow \dots$$

starting from an initial test case $t[0]$, chosen at random. The method is as follows:

1. Analyze $t[k]$: Execute the program with input $t[k]$, and determine its execution path $p[k]$. Then perform a (partial) symbolic execution of $p[k]$, so as to determine (an approximation to) its path predicate $P[k]$.

2. Select next test case: Determine the next test case $t[k + 1]$ so that it violates at least one constraint in each of the path predicates $P[j]$, for $j < k$.

We are thus assured that each new test case $t[k + 1]$ causes the program to traverse a genuinely new path, different from all those previously chosen.

In comparison with the previous methods we have discussed, Kundu reverses the roles of the test paths and the test data. The path $p[k]$ is determined from $t[k]$ in order to guide the next test case $t[k + 1]$ away from previous paths. That is, $p[k]$ is not used for finding an input that corresponds to that path itself. Therein lies the novelty of the approach.

Moreover, Kundu’s method is definitely not designed with any specific measure of test thoroughness in mind. (He asserts that no good measures of testedness are available, anyway.) It is clear, however, that one could easily augment his procedure with test instrumentation devices, as discussed earlier, for the purpose of assuring that some standard test coverage criterion (e.g., branch coverage) has been met.

The primary advantage of Kundu’s method is easily understood. Consider the constraint on $t[k + 1]$ as described in (2) above, i.e.,

$$t[k + 1] \text{ not in } P[1] \cup P[2] \cup \dots \cup P[k].$$

It is clear that the “forbidden region” for $t[k + 1]$ thus represents only a small portion of the total input domain D (see Fig. 8). This is so because the number of test cases generated in the testing activity is very small compared with the total number of executable paths in the program. Intuitively, the determination of the required $t[k + 1]$ should thus be relatively easy. And for the same reason, the determination of a test datum in a given path domain (as is required in the usual strategy) should be more difficult. Kundu (see Ref. 27, pp. 176–177)

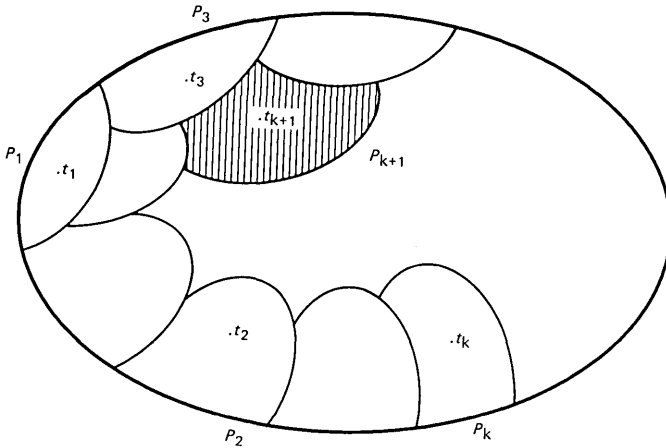


Fig. 8—Illustration of the forbidden region for selecting test cases.

gives a more detailed account of this reasoning, and the thrust of his argument is quite compelling. The reader may wish to consult Kundu's article for these additional details.

VI. CONCLUDING REMARKS

We have attempted to describe the many interesting and varied approaches to the program testing problem. Whereas no single approach to the problem may hold all the answers, it seems that there are enough good ideas around as to suggest the feasibility of a workable methodology, based on one or another combination of the strategies that have been advanced to date.

It must be remembered, however, that the thrust of our presentation, and, indeed, the main thrust in the literature has been toward the "unit test" level, where smaller programs are encountered. Thus, the ideas we have presented are, at the present time, feasible only in the case of programs of limited size. To think that we are nearing the point where we are ready to apply all of these techniques to the testing of an entire operating system or a compiler would be to miss the point completely. Nevertheless, our study has shown that indeed a start has been made.

We have tried to present a reasonably balanced survey of the recent contributions to the research literature on software testing methodology. It is perhaps likely that one or more worthwhile studies have escaped the author's attention, and therefore, their omission from this survey should not reflect on their importance to the development of the field. Moreover, the author can only hope that the studies that have been cited here have been presented in their best light. Limita-

tions of time and space have prevented a more complete treatment of these works, and for this, apologies to the authors are in order. At the same time, this author would like to acknowledge the use of the many cogent examples from the literature cited, hoping as well that these and other contributions have been faithfully reported.

In conclusion, the author would like to thank W. H. Leung, K. A. Gluck, and N. H. Petschenik for their most helpful comments in reviewing an earlier draft of the manuscript.

REFERENCES

1. W. C. Hetzel (ed.), *Program Test Methods*, Englewood Cliffs, NJ: Prentice Hall, 1973.
2. G. J. Myers, *The Art of Software Testing*, New York: John Wiley and Sons, 1979.
3. E. F. Miller, "Program Testing: Art Meets Theory," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 390-8.
4. E. F. Miller, "Program Testing Technology in the 1980's," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 399-406.
5. E. F. Miller, "Introduction to Software Testing Technology," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 3-14.
6. E. F. Miller, M. R. Paige, J. P. Benson, and W. R. Wisheart, "Structural Techniques of Program Validation," *Tutorial: Software Testing and Validation Techniques*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 262-5.
7. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. on Software Eng., SE-1*, No. 2 (June 1975), pp. 156-73.
8. J. B. Goodenough, "A Survey of Program Testing Issues," in *Research Directions in Software Technology*, P. Wegner (ed.), Cambridge, MA: MIT Press, 1979, pp. 316-40.
9. R. G. Hamlet, "Test Reliability and Software Maintenance," *Proc. Computer Software and Applications Conf. COMPSAC 78*, November 13-16, 1978, Chicago, IL, New York: IEEE, 1978, pp. 315-20.
10. E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng., SE-6*, No. 3 (May 1980), pp. 236-46.
11. E. J. Weyuker and T. J. Ostrand, "Current Directions in the Theory of Testing," *Proc. Computer Software and Applications Conf., COMPSAC 80*, October 27-31, 1980, Chicago, IL, New York: IEEE, 1980, pp. 386-9.
12. J. C. Huang, "An Approach to Program Testing," *Computing Surveys*, 7, No. 3 (September 1975), pp. 114-28.
13. J. C. Huang, "Program Instrumentation and Software Testing," *Computer*, 11, No. 4 (April 1978), pp. 25-32.
14. J. C. Huang, "Program Instrumentation: A Tool for Software Testing," *INFOTECH State of the Art Report, Software Testing*, Infotech Intl. Ltd. (1979), pp. 149-59.
15. W. E. Howden, "Methodology for the Generation of Program Test Data," *IEEE Trans. Computers, C-24*, No. 5 (May 1975), pp. 554-9.
16. W. E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Trans. Software Eng., SE-2*, No. 3 (September 1976), pp. 208-15.
17. W. E. Howden, "Introduction to the Theory of Testing," in *Tutorial: Software Testing and Validation*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 16-19.
18. W. E. Howden, "A Survey of Dynamic Analysis Methods," in *Tutorial: Software Testing and Validation*, Miller and Howden (eds.), New York: IEEE, 1978, pp. 184-206.
19. L. A. Clarke, "Automatic Test Data Selection Techniques," *INFOTECH State of the Art Report, Software Testing*, Infotech Intl. Ltd., 1979, pp. 43-63.
20. M. R. Paige, "On Sizing Software Testing for Structured Programs," *Intl. Symp. on Fault Tolerant Computing*, New York: IEEE, June 1977, p. 212.
21. M. R. Paige, "An Analytical Approach to Software Testing," *Proc. Computer*

- Software and Applications Conf., *COMPSAC 78*, November 13–16, 1978, Chicago, IL, New York: IEEE, 1978, pp. 527–31.
22. M. R. Paige, "Program Graphs, an Algebra, and Their Implication for Programming," *IEEE Trans. Software Eng.*, *SE-1*, No. 3 (September 1975), pp. 286–91.
 23. T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, *SE-2*, No. 4 (December 1976), pp. 308–19.
 24. L. A. Clarke and D. J. Richardson, "Symbolic Evaluation Methods for Program Analysis," in *Program Flow Analysis*, Muchnick and Jones (eds.), Englewood Cliffs, NJ: Prentice Hall, 1981, pp. 264–300.
 25. L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. Software Eng.*, *SE-2*, No. 3 (September 1976), pp. 215–22.
 26. C. V. Ramamoorthy, S. B. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. Software Eng.*, *SE-2*, No. 4 (December 1976), pp. 293–300.
 27. S. Kundu, "SETAR—A New Approach to Test Case Generation," INFOTECH State of the Art Report, Software Testing, Infotech Intl. Ltd., 1979, pp. 163–86.

AUTHOR

Ronald E. Prather, B.S. and M.S. (Electrical Engineering), 1955 and 1958, respectively; M.A. (Mathematics), 1966, University of California, Berkeley; Ph.D. (Mathematics), 1969, Syracuse University. Dr. Prather is a Professor of Mathematics and Computer Science at the University of Denver. He spent the 1982–1983 academic year on sabbatical leave with the Software Quality Analysis group at Bell Laboratories in Denver. He is the author of *Discrete Mathematical Structures for Computer Science* (Houghton Mifflin, 1976) and *Problem Solving Principles: Programming With Pascal* (Prentice Hall, 1982).

Parallel Fault Simulation Using Distributed Processing*

By Y. H. LEVENDEL,[†] P. R. MENON,[†] and S. H. PATEL[†]

(Manuscript received June 3, 1983)

This paper presents a method of performing fault simulation of digital logic circuits using a special-purpose computer with distributed processing. The architecture for true value simulation presented in an earlier paper can also be used for parallel fault simulation. The special-purpose computer consists of inexpensive microprocessors interconnected by either a time-shared parallel bus or a cross-point matrix. The cross-point matrix provides higher performance than the time-shared parallel bus. The performance of the proposed simulator is better by over two orders of magnitude than traditional logic fault simulation performed on a general-purpose computer. The power of the simulator is proportional to the number of microprocessors over a certain range.

I. INTRODUCTION

Fault simulation is an important part of the logic circuit design process. It is a means of determining the behavior of a logic circuit in the presence of each one of a predefined set of faults.

One of the most common uses of fault simulation is in determining the set of faults detected by a proposed test sequence, i.e., its fault coverage. Adequate fault coverage (usually greater than 90 percent of single stuck faults) is necessary to guarantee that the test sequence will detect most of the manufacturing defects.

* This paper is based upon material to be submitted by S. H. Patel in partial fulfillment of the requirements for the Ph.D. in Electrical Engineering at the Illinois Institute of Technology. [†] Bell Laboratories.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

In test pattern generation, fault simulation is used to determine the faults that are detected by the tests already generated so they can be removed from consideration. A yet undetected fault is then selected as a target for the next test. Additional faults detected by a newly generated test may be determined by simulation. Thus, fault simulation is used frequently as a part of the test generation process.

Fault simulation is also used to construct fault dictionaries for fault isolation. Other uses of fault simulation include the evaluation of test point effectiveness and the evaluation of self-checking circuitry. Effective test points are essential for factory testing. It is much cheaper and easier to locate and repair failures during manufacture than it is in the field. Also, good self-checking circuitry makes it easier to isolate faults in the field.

Currently, fault simulation is carried out on large general-purpose computers. This method has seen some use in large-scale integrated (LSI)* designs, but suffers from excessive run time at current levels of integration. Its applicability to very large-scale integration (VLSI) is doubtful, at least in the manner that it is currently used.¹ There is a need for more sophisticated and cost-effective fault simulators as very large simulation time and costs will result when dealing with circuits of VLSI complexity (more than 100,000 gates on a single chip).

II. PARALLEL FAULT SIMULATION

A number of different algorithms have been developed for performing fault simulation efficiently on general-purpose computers. Among these the best known and widely used are the parallel,² deductive,³ and concurrent⁴ methods. All these methods attempt to simulate the effects of a number of individual faults simultaneously. This paper will consider the use of the parallel fault simulation algorithm in the special-purpose simulation hardware architecture developed in Ref. 5.

In parallel fault simulation the fault-free circuit and several different faulty circuits are processed simultaneously. The number of faulty circuits simulated in parallel is normally constrained by the number of bits in the host computer word. One bit of the computer word represents the signal value on a line in the fault-free circuit, while the remaining bits represent values on the same line in the presence of different single faults. Word-oriented operations performed on the host computer imply that the fault-free and faulty circuits are handled simultaneously and in exactly the same manner.

The output fault word of a gate is computed by simple word-oriented logic operations on the input fault words. The logic operations performed on the fault words correspond to the logic operation performed

* Acronyms and abbreviations are defined in the Glossary at the back of this article.

by the gate. Faults are injected using predefined masks. A stuck at 1 fault on a lead is injected by ORing the fault word with a mask containing a 1 in the bit position for the faulty value and 0's elsewhere. Similarly, a stuck at 0 fault can be injected by ANDing the fault word with a mask containing a 0 in the bit position for the faulty value and 1's elsewhere.

Two logic values are not sufficient for accurate logic simulation. Since each bit position in the computer word can represent only a logical 0 or 1, more than one bit per signal is required for multiple-value simulation.⁶ In this case more than one word is required. For three-value representation two bits are required to represent each signal and, therefore, two computer words are required for representing a set of fault-free and faulty values. Since a pair of words are used to represent a signal, some sort of coding method is required to implement parallel simulation. A commonly used method of coding denotes one of the words as the 0-word and the other word as a 1-word. Let i_0 represent the i th bit in the 0-word and i_1 represent the i th bit in the 1-word. Then the $i_0i_1 = 01$ combination represents a logical 1, the $i_0i_1 = 10$ combination represents a logical 0, the $i_0i_1 = 00$ combination represents an unknown, and the $i_0i_1 = 11$ combination is unused. Simple word-oriented operations are still sufficient for performing the parallel simulation. This coding is the same as that in Ref. 7. Faults are injected in the 0-word and the 1-word using a 0-mask and a 1-mask, respectively. This injection is also done using simple logic operations. The method for simulating three logic values can be extended to any number of logic values by coding them using a sufficient number of bits.⁸ Only three-valued simulation is considered in this paper.

The most widely used method of parallel simulation is the *event-driven* method. Event-driven simulation means that an element is not simulated unless there is a change in one of its input fault words. The main operations performed in an event-driven simulation are processing of active elements and scheduling changes to occur in future. The scheduling is done on a timing wheel. A *timing wheel* is a list structure in which events are chained together in the order they are to occur. In parallel fault simulation an element is considered to be *active* if the fault word associated with its output changes. The fault word is considered changed even if only one of its bits changes. All the values (i.e., the whole fault word) are propagated even if only one of the values changes.

Since the number of faults simulated at one time is restricted by the length of the host computer word, multiple passes through the simulator are necessary to simulate a large number of faults. It is possible to reduce the number of passes by using extra computer words

and simulating more faults during one pass. For example, 64 computer words can be used to simulate (two-value simulation) 1024 faults on a computer with a word size of 16 bits. The string of values in the set of computer words representing the fault-free and faulty signal values on a line will be called the *value vector*. The configuration of the value vector is shown in Fig. 1. The value vector consists of L_p word pairs for three-value simulation. Two bits at the same position in the two words of a pair represent the signal value of one faulty circuit. Simulating L_p word pairs at a time is better than making L_p passes through the simulator since the overhead involved in the fanout search associated with each pass is saved. (However, as we will see in Section 6.1, some of these savings are lost due to increased activity as the number of faults per pass increases.) The number of words used in the value vector is usually constrained by the space requirements of the computer.

During simulation, operations are performed on value vectors by considering word pairs. Thus, the time required to perform an operation on the value vector is proportional to the number of word pairs used in the value vector. For example, if there are several word pairs in a value vector, then faults are injected one word pair at a time.

The whole value vector is considered active if any of the values in the vector changes. Furthermore, all the word pairs are propagated even if only one word pair changes. An element is considered *active* if the value vector associated with its output is active.

III. CONCURRENCY IN FAULT SIMULATION

At least three types of concurrencies exist in fault simulation of logic circuits. The first type of concurrency occurs in the actual simulated hardware, the second type occurs in the simulation algo-

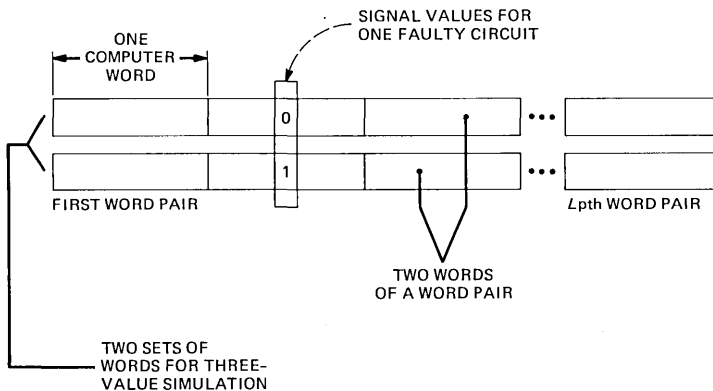


Fig. 1—Value vector configuration.

rithm, and the third type occurs in the form of fault activity. The first two types of concurrencies also occur in true value simulation of logic circuits and have been discussed in an earlier paper.⁵

The concurrency occurring in the actual simulated hardware can be called *logic circuit concurrency*. Utilizing this type of concurrency leads to distributed processing with identical processors performing identical tasks. The architecture for true-value simulation developed by Levendel et al.⁵ and Denneau et al.⁹⁻¹¹ takes advantage of this type of concurrency.

The concurrency occurring in the simulation algorithm can be called *algorithm concurrency*. This concurrency is indirectly due to the concurrency occurring in the actual simulated hardware. Since several elements can be simultaneously active and a sequence of steps is to be performed for each active element, they can be processed in a pipeline fashion. Utilizing this type of concurrency leads to functional partitioning of tasks among several processors and a pipelined architecture. The architecture for true-value simulation developed by Barto and Szygenda¹² and Abramovici et al.¹³ takes advantage of this type of concurrency.

For efficient fault simulation, a number of faults are simulated simultaneously in software-based simulators. This leads to *fault activity concurrency*, which can be utilized in special-purpose hardware for fault simulation.

This paper extends the architecture for true-value simulation described in Ref. 5 to fault simulation using the parallel method. The architecture takes advantage of the parallelism due to logic circuit concurrency and fault activity concurrency. The main difference between true-value simulation and fault simulation is in the algorithm executed by the individual processing units.

IV. SPECIAL-PURPOSE ARCHITECTURE

The simulator consists of one master and a number of slaves (processors) interconnected by a communication structure (Fig. 2). The communication structure is used as a medium for transferring data between the slaves and between the slaves and the master. The communication structure can be either a time-shared parallel bus or a cross-point matrix. The circuit to be simulated is partitioned into subcircuits and each subcircuit is simulated in a separate processor. Subcircuits in different processors become active as signal values proceed from the primary inputs to primary outputs. As simulation progresses, data are transferred between subcircuits as the logic values on the signal connections between two subcircuits change. These data are transported across the communication structure. Typical data sent across the data path consist of element information and changed value

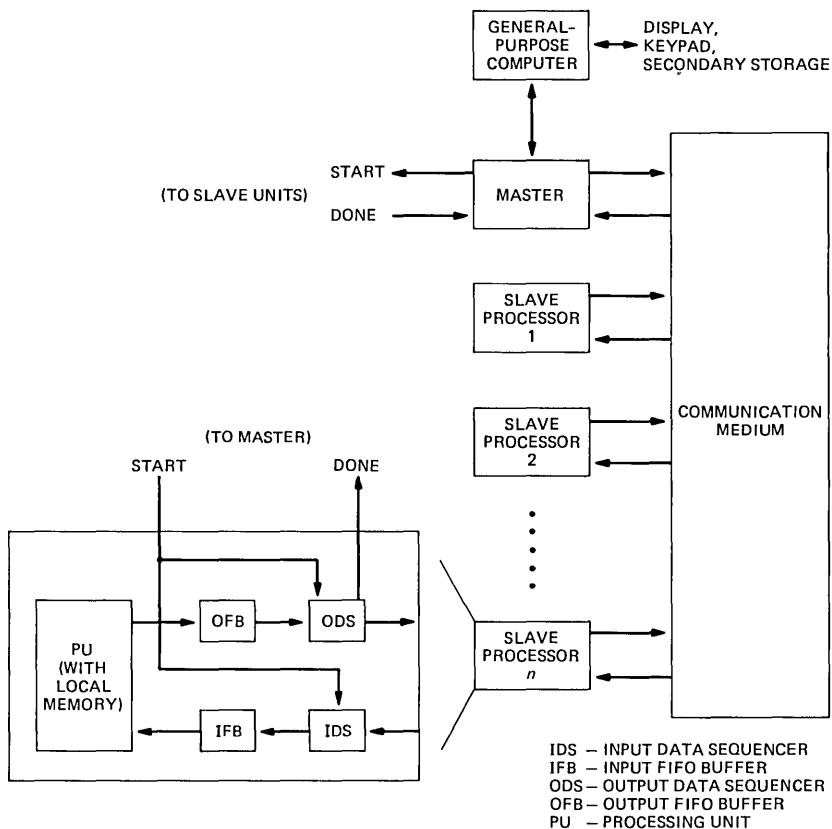


Fig. 2—Multiprocessor-based digital logic simulator.

vectors. The architecture of the simulator has been described in detail in the previous paper.⁵ A summary of the architecture description is presented in Appendix A for completeness.

The overall architecture for true-value simulation is applicable to fault simulation since the algorithms for the two types of simulations are the same except that fault simulation requires:

1. Carrying of faulty signal values in addition to true-value signals (more than one word may be used to allow representation of a larger number of faulty signals)
2. Fault injection using masks
3. Multiple passes if the fault set is large.

The processing time per pass will be higher for fault simulation due to the extra processing required for injecting faults and manipulating multiple-word pairs in a value vector. The latter requirement applies only when the value vector contains more than one word pair. As far as communication between the processors is concerned, the data

transferred between the processors are greater than those transferred for true-value simulation, since the signal values require one or more complete words in addition to a word for the element number.

V. ARCHITECTURE EVALUATION

The processing time per simulation cycle and the communication times using a time-shared parallel bus and a cross-point matrix are estimated first. These results are then used for selecting the communication structure. A multiple-bus communication structure is also discussed in this section.

5.1 Processing time t_p

The average number of active elements per processor during a simulation cycle for true-value simulation is given by kN/n , where N is the average number of active elements per simulation cycle during true-value simulation, n is the number of processors in the multi-processor simulator, and k is the average unbalance factor representing the extra active elements per processor during a simulation cycle due to nonideal partitioning.⁵ Ideal partitioning will cause an equal number of elements to be active in all the processors during all simulation cycles. However, because of some imbalance such as the fanout of all active elements during one simulation cycle not feeding equally into all the processors, some processors will have more active elements than the others during some simulation cycles. The average number of active elements per processor during a fault simulation cycle can be written as kN_f/n , where N_f is the average number of active elements per simulation cycle in one pass during fault simulation. The value of N_f is expected to be larger than N . Indeed, experimental runs on the Logic Analyzer for Maintenance Planning (LAMP) simulator¹⁴ show that the overall activity (total number of active elements) during parallel fault simulation increases by a factor of about 2 for 16 faults per pass and by about 3.5 for 1024 faults per pass compared to true-value simulation. These results are averaged over several runs made using both combinational and sequential circuits with sizes ranging from 420 gates to 1912 gates. The number of faults simulated ranged from 1020 faults for the 420-gate circuit to 5046 faults for the 1912-gate circuit. The LAMP simulator is based on the deductive method. A mapping mechanism from deductive simulation activity to parallel simulation activity was implemented to predict the results for parallel simulation.

During one simulation cycle the following major operations occur in the given order:

1. Using the current list of events, list L_t of the timing wheel (Fig.

11 in Appendix B), update, and find fanout of the elements whose outputs changed during the current simulation cycle.

2. Using the next list L_{t+1} of the timing wheel, prepare external events to be transmitted to other processors for the next time interval.

3. From data in the Input FIFO Buffer (IFB) (sent by other processors), update and find fanout of the elements active during the current simulation cycle.

4. Evaluate the fanout of active elements (this includes fault injection).

5. Schedule on timing wheel elements whose output changes.

The detailed algorithm is given in Appendix C.

Let t_a be the time required to process one active element. The average processing time per processor during one simulation cycle is then given by:

$$t_p = \frac{kN_f}{n} t_a.$$

Assume a microprogrammable microprocessor (e.g., Am2900 series) for each slave unit processing unit (PU) and the following operation times (150 ns cycle time): memory-to-memory move = 1.2 μ s, memory-to-register move = 0.6 μ s, and memory-to-memory logical AND/OR operation = 1.5 μ s. Using these major microprocessor operations, the execution time for each operation in the parallel simulation algorithm described in Appendix C can be estimated. For example, obtaining each fanout of an updated element (after the fanout list has been accessed) takes one memory-to-register instruction and moving the element number to the Output FIFO Buffer (OFB) takes one memory-to-memory instruction. The processing times per active element for the various steps of the algorithm are shown in Table I, where f_i is the average fan-in, f_o is the average fan-out, and L_p is the number of word pairs in the value vector.

The total processing time per element during one simulation cycle is the sum of all the expressions in Table I:

$$t_a = 9.6 + 8.7L_p + 3f_o + 3f_iL_p - \frac{(2.4 + 7.2L_p)}{f_o} \mu\text{s}.$$

Taking typical values of f_i and f_o to be 2 and an unbalance factor of $k = 1.1$, the processing time for a simulation cycle becomes:

$$t_p = (15.8 + 12.2L_p) \frac{N_f}{n} \mu\text{s}. \quad (1)$$

5.2 Communication time t_c

The value of t_c will depend on the type of communication structure.

Table I—Simulation cycle timings for parallel simulation

Step	Expression (μs)
Update data from timing wheel	$\frac{(1.8 + 2.4f_o)}{f_o}$
Prepare external events for next time interval	$\frac{(f_o - 1)(6.6 + 3.6L_p)}{f_o}$
Update data from IFB	$\frac{(f_o - 1)(1.2 + 3f_o + 3.6L_p)}{f_o}$
Evaluate schedule	$2.4 + 1.5L_p + 3f_iL_p + \left(\frac{3.6}{f_o}\right)$

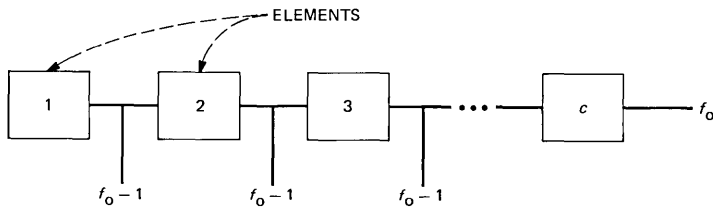


Fig. 3—An element string.

The two types of communication structures discussed in Ref. 5, namely the time-shared parallel bus and the cross-point matrix, will be considered here also. The partitioning algorithm discussed in the previous paper⁵ partitions a circuit along its depth rather than its breadth. Since the signals in a circuit propagate in parallel, this places concurrent activities in different blocks. The same partitioning algorithm will be assumed here, since during fault simulation the signals still propagate in the same manner. The logic circuit to be simulated is partitioned into elements strings (see Fig. 3). The average number of communication events generated by one active element during a simulation cycle that have to be sent over the communication structure is:

$$e = \frac{[f_o + (c - 1)(f_o - 1)]}{c},$$

where f_o is the average fanout and c is the average number of elements in one element string. The typical value of f_o can be taken as 2, and for large circuits c is expected to be greater than 10. For $f_o = 2$ and $c = 10$, e will be equal to 1.1.

5.2.1 Time-shared parallel bus

The total communication time during a simulation cycle for true-value simulation is given by⁵:

$$t_{c(\text{bus})} = (n + 200)Ne \text{ ns}$$

This expression assumes one word of data to be transferred per active element. For fault simulation with w words of data to be transferred per active element the expression becomes:

$$t_{c(\text{bus})} = (n + 200)N_f e w \text{ ns.}$$

The number of words to be transferred is given by:

$$w = 1 + 2L_p,$$

where L_p is the number of word pairs per value vector used in simulation. One word is required to carry the element number and the $2L_p$ words carry the value vector. Taking the value of $e = 1.1$:

$$t_{c(\text{bus})} = (1.1n + 220)(1 + 2L_p)N_f \text{ ns.} \quad (2)$$

5.2.2 Cross-point matrix

In a cross-point matrix-based communication structure several processors can be simultaneously sending data to other processors. The total communication time during a simulation cycle for true value simulation is given by⁵:

$$t_{c(\text{matrix})} = \frac{200Nke}{n} + 50j \text{ ns,}$$

where j is the number of events for which the destination processor is found busy, i.e., the destination processor is communicating with some other processor. Once again this expression assumes one word of data transferred per active element. For w words of data to be transferred, the expression for fault simulation becomes:

$$t_{c(\text{matrix})} = \frac{200(N_f)kew}{n} + 50jw \text{ ns.}$$

For $k = 1.1$, $e = 1.1$, $w = 1 + 2L_p$, and $j = 0.1N_f/n$ (the channel is found busy for 10 percent of the transfer requests), the above expression can be rewritten as:

$$t_{c(\text{matrix})} = (247 + 494L_p) \frac{N_f}{n} \text{ ns.} \quad (3)$$

5.3 Choice of communication structure

The expressions for the processing time per simulation cycle per active element and the communication times per simulation cycle per active element for the bus and matrix structures are plotted in Fig. 4. The expressions are plotted for value vector length of one word (16 bits/word), i.e., $L_p = 1$.

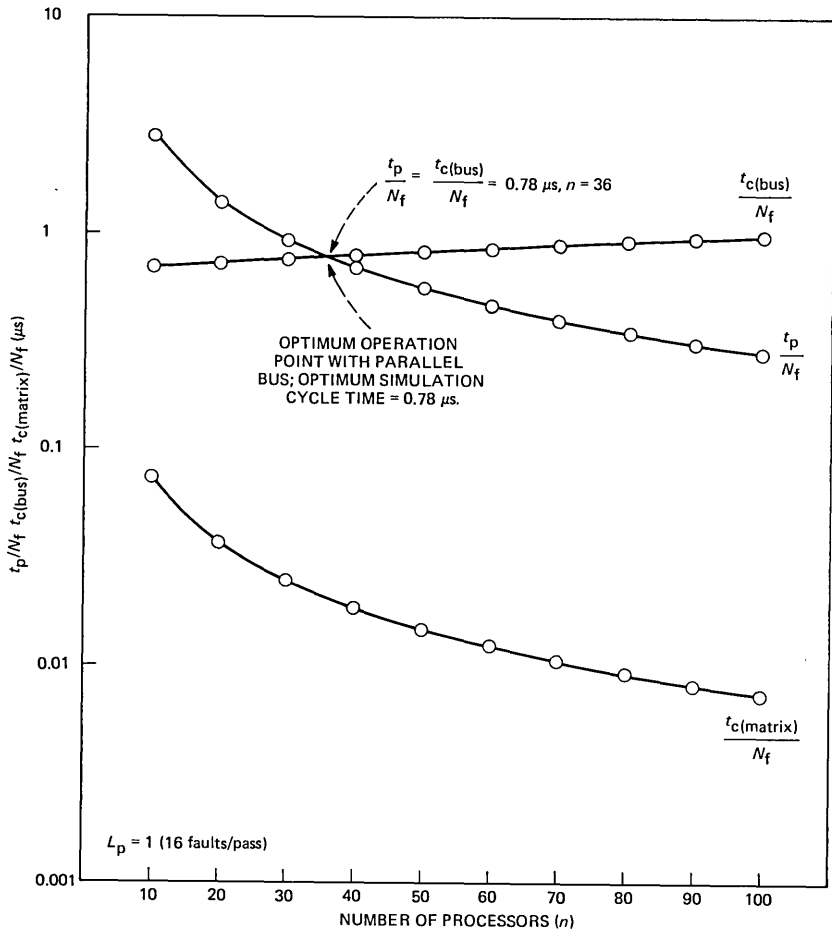


Fig. 4—Variation in processing and communication time.

The curves for the processing time and bus communication time for the parallel bus intersect at $n = 36$. The processing time is greater than the bus communication time for $n < 36$. Thus the processing time is the bottleneck. The processing time decreases as the value of n increases. For $n > 36$ the bus communication time becomes larger than the processing time and the bus communication time becomes the bottleneck. Therefore, using more processors than $n = 36$ will not speed up the simulation. For optimum performance $n = 36$, and the length of the simulation cycle per active element becomes $t_m = 0.78 \mu\text{s}$. Based on the parallel simulation algorithm given in Appendix C, the actual simulation cycle length per active element for a single processor can be estimated as $t_1 = 24 \mu\text{s}$. The multiprocessor fault

simulator with a bus-based communication structure provides a speedup of 31 over the traditional single-processor logic fault simulator.

For further speedup a faster communication structure must be used. Figure 4 also shows the curve for the matrix communication time. This curve does not intersect with the curve for the processing time, and the communication time is always less than the processing time. The communication time will therefore never be a bottleneck. More processors can be added to speed up the simulator. For example, for $n = 100$ the speedup compared to the traditional single-processor logic fault simulator is 86 and for $n = 256$ the speedup is 220. The speedup of simulation is expected to be greater than two orders of magnitude for $n > 120$.

5.4 Multiple-bus communication structure

The results of the previous section show that for a given number of faults per pass, the time-shared parallel bus is useful only for up to a fixed value of n . For further speedup the cross-point matrix has to be used. However, the cross-point is not used up to its maximum capability. For example, at 16 faults per pass and $n = 100$, the simulation cycle length is $280N_f$ ns while the communication cycle length is $7.4N_f$ ns, i.e., the communication structure is used less than 3 percent of the time. A communication structure that is slower and cheaper than the cross-point matrix might prove more cost-effective. This will be true especially since the control for the cross-point is very complex and thus expensive.

A communication structure that provides a capacity in between that of the time-shared parallel bus and the cross-point matrix is the multiple-bus structure. It consists of a bus arbitrator and several parallel buses. The configuration of the multiple-bus structure and its interface to the Output Data Sequencers (ODSs) and Input Data Sequencers (IDSs) of the slave units are given in Fig. 5. When the ODS needs to send data, it sets the Request To Send (RTS) line high and puts the destination address on the address lines. The requesting ODS keeps the RTS line high until granted a bus. The bus arbitrator grants it the use of the communication medium when it finds an unused bus and determines that the requested destination is not busy. The bus arbitrator grants the bus by setting the Bus Grant line high. The data are switched through the bus selector switch to the available bus. At the other end of the bus there is a line selector from which the data are sent to the destination processor. The ODS sends out all the data present in its Output FIFO Buffer (OFB). The data received by the IDS of the destination unit are put in its Input FIFO Buffer (IFB). The ODS then sets the RTS line low. This releases the bus, which is

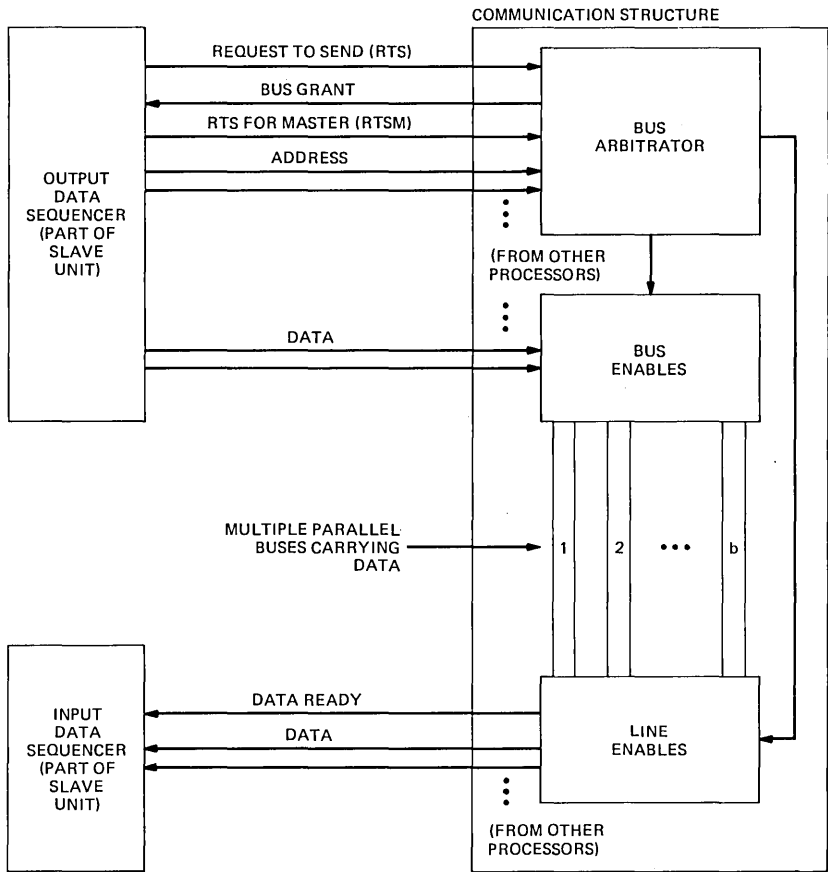


Fig. 5—Configuration of multiple-bus structure.

then granted by the bus control to another requesting slave or the master. All units have equal priority. The ODS will set the Request To Send line high again if it gets more data to transfer in the OFB.

The data sent out to a slave unit from another slave unit or the master consist of element information and changed value vectors. The data sent to a master consist of the address of the sending slave, element number (primary output or monitored point), and value vector. A separate line Request to Send to Master (RTSM) is used to address the master. When the destination is the master, the address lines from the ODS contain the sending slave unit address. This address together with the element number and value vector is stored in the master IFB by the master IDS.

To obtain an expression for the communication time for the multiple-bus structure, consider first of all the bus structure with only a

single bus. For this case, the communication time will consist of the same components as that for time-shared parallel bus⁵:

$$t_{c(\text{mbus})} = (t_{\text{brg}} + t_{\text{ds}} + t_{\text{da}} + t_{\text{br}})(1 + 2L_p)(N_f)e,$$

where t_{brg} is the bus request and grant time, t_{ds} is the address and data setup time, t_{da} is the data acknowledge, and t_{br} is the bus release time. For the multiple-bus structure the bus request and grant time, t_{brg} , will be greater than in the parallel bus since extra checking has to be done before a bus is granted. The bus arbitrator will have to determine if a bus is available and if available then it has to further determine if the requested destination is busy. Assuming these extra actions double the time required for the bus request and grant time and the other times remain the same: $t_{\text{brg}} = 200$ ns, $t_{\text{ds}} = n$ ns, $t_{\text{da}} = 50$ ns, $t_{\text{br}} = 50$ ns, and $e = 1.1$. Each transaction across the multiple-bus has to wait for a bus to be granted. As more buses are added, the transactions can occur in parallel. Assuming the number of parallel buses is much smaller than the number of processors, the probability of the destination processor being busy will be small. The decrease in the total communication time will then be proportional to the number of buses. The expression for the total communication time for the multiple-bus communication structure becomes:

$$t_{c(\text{mbus})} = \frac{(1.1n + 330)(1 + 2L_p)N_f}{b}, \quad (4)$$

where b is the number of parallel buses.

Let n_o be the number of processors at the optimum operation point, where $t_p = t_{c(\text{mbus})}$. If the simulator operates with the number of processors not equal to n_o , then the speed of simulation will be lower. An expression for n_o can be derived by equating eqs. (1) and (4):

$$n_o = -150 + 150 \left(1 + b \left(\frac{0.49L_p + 0.64}{2L_p + 1} \right) \right)^{0.5}.$$

This expression is plotted for various values of b with $L_p = 1$ in Fig. 6. It can be seen that higher performance is available with the multiple-bus structure compared to the single time-shared parallel bus for $b \geq 2$.

The implementation of the multiple-bus structure is expected to be less expensive than that of the cross-point matrix. The complexity of the communication structures and thus their cost is proportional to the number of switch points. For the cross-point matrix, the number of cross-points is proportional to the square of the inputs, i.e., n^2 . For the multiple-bus structure two points have to be connected to establish a connection and thus the number of switch points is $2bn$. The cost of the multiple-bus structure will be less than that of the cross-point

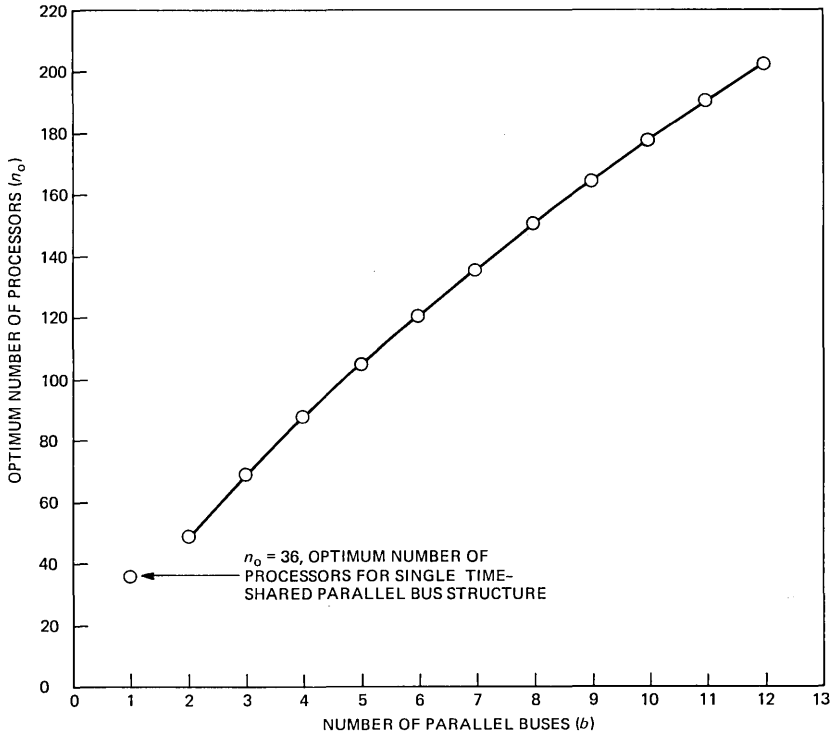


Fig. 6—Variation in the optimum number of processors with the number of parallel buses.

matrix as long as $2b < n$. As we saw in the analysis done earlier this will always be the case. For $b = 5$ and $n_o = 105$, the cost of the multiple-bus structure will be an order of magnitude lower than that of the cross-point matrix. Furthermore, it must also be noted that physical switching is not necessary in the case of the multiple-bus structure; it is cheaper to use logic enables for connecting an ODS and an IDS to a bus. This will result in even lower cost when compared with the cross-point matrix.

VI. EFFECT OF NUMBER OF FAULTS PER PASS

When simulating a large number of faults per pass in parallel simulation, several pairs of words carrying the true and faulty values (L_p) must be manipulated per active element. All the faulty values in the words can be considered together as a vector. All the faulty values are evaluated even if only one faulty value is active.

The behavior of the multiprocessor fault simulator, using the parallel fault simulation algorithm, will be investigated for variations in the number of faults simulated per pass. Comparisons will be made

between 16, 32, 64, 256, and 1024 faults per pass, assuming a processor word length of 16 in all the cases.

Only the time-shared parallel bus and the cross-point matrix are discussed. The communication structure with multiple buses is not considered since the results for the time-shared parallel bus will apply, except for a scaling factor.

6.1 Simulator with time-shared parallel bus

The expressions for processing time per simulation cycle, t_p , and the parallel bus communication time per simulation cycle, $t_{c(\text{bus})}$, derived earlier in eqs. (1) and (2) are used to analyze the effects of variations in the number of faults per pass. It was seen earlier that the optimum operation point for the simulator occurs when the processing time and communication time are equal, i.e., $t_p = t_{c(\text{bus})}$. An expression for the number of processors required to meet this condition for a given length of value vector (L_p) can be derived by equating the expressions for t_p and $t_{c(\text{bus})}$. Let n_o be the number of processors at the optimum operation point and t_o be the length of the processing and communication cycles at the optimum operation point. Values of n smaller than the optimum n_o cause the processing to be a bottleneck, while larger values of n cause communication to be a bottleneck. Equating eqs. (1) and (2) yields the following expression for n_o , the number of processors required for optimum operation:

$$n_o = -100 + 100 \left(\frac{3.11L_p + 2.44}{2L_p + 1} \right)^{0.5}.$$

Let α be the number of faults per pass. Then $\alpha = 16L_p$ assuming 16 bits per word. The above expression for n_o can be rewritten as a function of the number of the faults α , as:

$$n_o(\alpha) = -100 + 100 \left(\frac{0.19\alpha + 2.44}{0.125\alpha + 1} \right)^{0.5}. \quad (5)$$

For $n = n_o(\alpha)$, the processing time per simulation cycle and the bus communication time per simulation cycle both reduce to:

$$t_o(\alpha) = (15.8 + 0.76\alpha) \frac{N_f(\alpha)}{n_o(\alpha)}. \quad (6)$$

For a simulator with optimum number of processors, $n_o(\alpha)$, the total time required to simulate a fixed number of faults is obtained by multiplying the time required for processing one active element ($t_o(\alpha)/N_f(\alpha)$) by the total number of active elements during the simulation ($N_T(\alpha)$):

$$T(\alpha) = \frac{t_o(\alpha)}{N_f(\alpha)} N_T(\alpha).$$

The total number of active elements during simulation is given by:

$$N_T(\alpha) = N_f(\alpha) \times (\text{number of simulation cycles per pass}) \\ \times (\text{number of passes}).$$

Substituting the expression for $t_o(\alpha)$ given in eq. (6):

$$T(\alpha) = (15.8 + 0.76\alpha) \frac{N_T(\alpha)}{n_o(\alpha)}. \quad (7)$$

Define the *simulation time ratio* as the ratio of the total time required to simulate a set of faults with α faults per pass to the total time required to simulate the given set of faults with 16 faults per pass, i.e., $T(\alpha)/T(16)$. Using a value of $n_o(16) = 36$ (eq. 5), the expression for the simulation time ratio is given by:

$$\frac{T(\alpha)}{T(16)} = \frac{(20.3 + 0.98\alpha)}{n_o(\alpha)} \frac{N_T(\alpha)}{N_T(16)}. \quad (8)$$

The values of the simulation ratio are first calculated theoretically and then compared with the experimental results.

6.1.1 Theoretical simulation time ratio

Let *simulation activity*, $N_T(\alpha)$, refer to the number of active elements during all passes of a simulation. The simulation activity can be expected to be inversely proportional to the number of faults per pass:

$$\frac{N_T(\alpha)}{N_T(16)} = \frac{16}{\alpha}.$$

For example, the expected simulation activity at 32 faults per pass will be half the simulation activity of 16 faults per pass since the number of passes needed will be halved. The theoretical expression for the simulation time ratio becomes:

$$\frac{T(\alpha)}{T(16)} = \frac{(20.3 + 0.98\alpha)}{n_o(\alpha)} \frac{16}{\alpha}. \quad (9)$$

Note that the theoretical simulation time ratio is independent of the simulation activity. Table II gives the variation of the optimum number of processors, n_o , and the variation of the simulation time ratio as a function of the number of faults per pass, α . The theoretical results show that the simulation time ratio, and thus the total simulation time, decreases as the number of faults per pass increases.

Also, it is interesting to note that for 16 faults per pass the operation

Table II—Theoretical simulation time ratio

Faults per Pass, α	Optimum Number of Processors, n_o	Simulation Time Ratio, $\frac{T(\alpha)}{T(16)}$
16	36	1.0
32	32	0.81
64	29	0.72
256	26	0.65
1024	25	0.64

Table III—Experimental simulation time ratio

Faults per Pass, α	Simulation Activity, $N_T(\alpha)$	Optimum Number of Processors, n_o	Simulation Time Ratio, $\frac{T(\alpha)}{T(16)}$
16	17586	36	1.0
32	9374	32	0.86
64	5204	29	0.84
256	1787	26	1.06
1024	689	25	1.6

peaks at 36 processors, while for 1024 faults per pass the operation peaks at 25 processors. As the number faults per pass increases, the point of optimum operation occurs for a slightly smaller number of processors. This is because the time required to transfer the increased data is more than the time required to process the increased data.

6.1.2 Experimental simulation time ratio

The values of $N_T(\alpha)$ averaged over several experimental runs made on the LAMP simulator¹³ (with a mapping algorithm from deductive simulation to parallel simulation as discussed in Section 5.1) are given in Table III. Also shown in Table III are values for the simulation time ratio derived from eq. (8).

The experimental simulation time ratio, $T(\alpha)/T(16)$, is plotted together with the theoretical simulation time ratio in Fig. 7.

It is interesting to note that as the number of faults per pass increases, the experimental simulation time ratio falls below 1.0 initially, i.e., the simulation speeds up. After a certain point, the simulation time ratio then starts increasing and goes above 1.0. The fastest simulator is a 64 faults per pass simulator with 29 processors. On the other hand, the theoretical simulation time ratio always stays below 1.0 and keeps decreasing as the number of faults per pass increases. The difference in the experimental and theoretical results can be

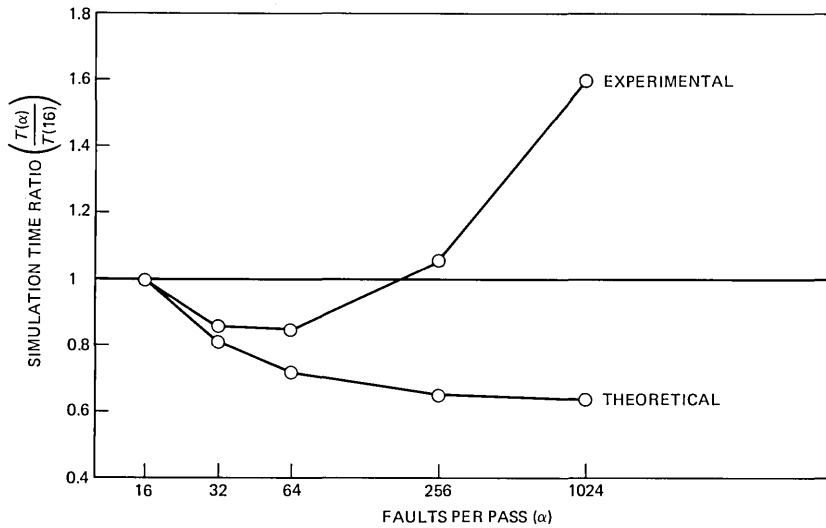


Fig. 7—Experimental and theoretical simulation time ratio for variation in number of faults per pass—a time-shared parallel bus.

explained by examining the variation in simulation activity as the number of faults per pass changes.

The curve for the theoretical simulation time ratio in Fig. 7 shows that the simulation speed increases as the number of faults per pass increases. This is as expected, since increasing the number of faults per pass decreases the number of passes and thus the fanout search and related processing time. In practice, however, there is extra simulation activity due to longer value vectors and this tends to increase the processing time. As more faults are simulated per vector, i.e., as the value of α gets larger, the simulation activity during the simulation will be higher than the theoretical. For example, as shown earlier, the theoretical activity at 32 faults per pass will be half the simulation activity at 16 faults per pass. However, the experimental simulation activity at 32 faults per pass will be more than the expected half. This is because the active faults in two value vectors at 16 faults per pass will not always directly map into one value vector at 32 faults per pass. Any active fault in the value vector will cause simulation activity even if the good value does not change. The effect of this is to cause extra schedulings. This increase in schedulings can be represented by an effective increase in the length of the simulation cycle and the number of simulation cycles. The runs made on the LAMP simulator show that most of the increase is in the length of the simulation cycle (i.e., more computation during the simulation cycle). The expected simulation activity and the actual simulation activity

obtained from runs made using the LAMP simulator are given in Table IV.

For 32 faults per pass, the simulation activity is only 1.07 times that theoretically expected. Thus the increase in processing time due to this extra activity is not substantial and the overall speed of simulation is higher due to the greater savings in the fanout search processing. For 1024 faults per pass, the simulation activity is 2.51 times that theoretically expected. In this case, the increase in processing time due to the extra activity is substantial compared to the savings obtained in the fanout search processing. This results in lowering the overall speed of simulation when compared with 16 faults per pass.

In summary, for the multiprocessor fault simulator with a parallel-bus-based communication structure, the fastest simulation speed occurs for 64 faults per pass and 29 processors. Note, however, that the number of processors required for 64 faults per pass is greater than the number of processors for 1024 faults per pass. Decreasing the number of processors for 64 faults per pass to 25 yields the total simulation time of 13,413 μ s. This still favors the 64 fault per pass simulator over the 1024 fault per pass simulator.

6.2 Simulator with cross-point matrix

The expressions for the processing time per simulation cycle, t_p , and the cross-point matrix communication time per simulation cycle, $t_{c(\text{matrix})}$, derived earlier in eqs. (1) and (3) are applicable to variations in the number of faults per pass. The increase in simulation activity caused by simulating more faults per pass will increase both the processing time and the communication time for the cross-point matrix. However, adding one word pair to the value vector will cause an increase in the communication time that is only 4 percent of the increase in the processing time. Thus, the communication time will always be less than the processing time. The cross-point matrix provides sufficient communication capacity for the parallel fault sim-

Table IV—Effect of multiple passes on simulation activity

Faults per Pass, α	Experimental Simulation Activity, $N_T(\alpha)$	Theoretical Simulation Activity, $N_T(\alpha)$
16	17,586	17,586
32	9374	8793
64	5204	4396
256	1787	1099
1024	689	275

ulator and does not cause a communication bottleneck. For faster fault simulation, more processors can be added.

The variation of the simulation time ratio as a function of the number of faults per pass will be investigated to obtain the optimum number of faults per pass. Since the processing time dominates, the total time required to simulate a set of faults with α faults will be equal to the total processing time. Using eq. (1):

$$T(\alpha) = (15.8 + 0.76\alpha) \frac{N_T(\alpha)}{n}$$

The simulation time ratio is given by:

$$\frac{T(\alpha)}{T(16)} = (0.56 + 0.027\alpha) \frac{N_T(\alpha)}{N_T(16)} \quad (10)$$

The experimental and theoretical simulation time ratios are plotted in Fig. 8. As was the case for the time-shared parallel bus, the experimental simulation time ratio increases after 64 faults per pass. This is because the time required to process the increased simulation activity is greater than the time saved in the fanout search overhead associated with each pass. For the multiprocessor fault simulator with a cross-point-matrix-based communication structure, the optimum number of faults per pass is 64. Using a different value for the number of faults per pass will decrease the speed of simulation. Note that the

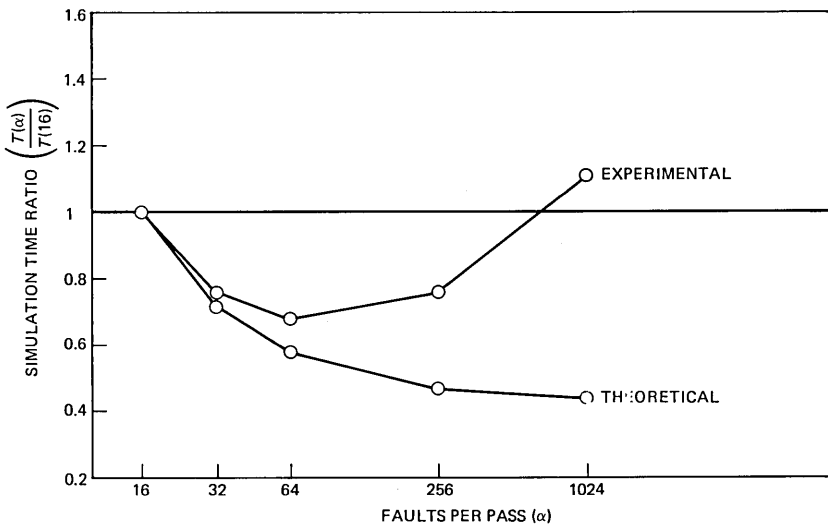


Fig. 8—Experimental and theoretical simulation time ratio for variation in number of faults per pass—cross-point matrix.

number of processors, n , does not affect the simulation time ratio. More processors can be added to obtain greater speed.

When compared with the parallel bus the cross-point matrix provides greater speed for any value of n greater than the optimum n for the parallel bus. For example, for 64 faults per pass, the cross-point matrix can be made faster than the parallel bus by selecting $n > 29$.

VII. SUMMARY

In this paper we presented a special-purpose logic fault simulator based on the parallel simulation method. The simulator is expected to be two orders of magnitude faster than traditional logic fault simulators implemented on general-purpose computers. For both the time-shared parallel bus and the cross-point matrix, the simulator performs the best at 64 faults per pass. Decreasing the number of faults per pass slows down the simulation due to fanout search and other overhead required for every pass. Increasing the number of faults per pass also slows down the simulation due to the increase of simulation activity.

When the parallel bus is used, the power of the simulator can be increased over a certain range by increasing the number of slaves. The power of the simulator can be further increased by using the cross-point matrix.

The application of the special-purpose simulator to other fault simulation methods is being investigated currently.

REFERENCES

1. D. F. Barbe (ed.), "Very Large Scale Integration (VLSI), Fundamentals and Applications," Berlin, Germany: Springer-Verlag, 1980.
2. E. W. Thompson and S. A. Szygenda, "Digital Logic Simulation in a Time-Based, Table-Driven Environment: Part 2. Parallel Fault Simulation," *Computer*, 8-3 (March 1975), pp. 38-49.
3. D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," *IEEE Trans. Computers*, C-21, No. 5 (May 1972), pp. 464-71.
4. E. G. Ulrich and T. G. Baker, "Concurrent Simulation of Nearly Identical Networks," *Computer*, 7-4 (April 1974), pp. 39-44.
5. Y. H. Levendel, P. R. Menon, and S. H. Patel, "Special-Purpose Logic Simulator Using Distributed Processing," *B.S.T.J.*, 61, No. 10, Part 1 (December 1982), pp. 2873-2909.
6. M. A. Breuer, "A Note on Three Valued Logic Simulation," *IEEE Trans. Computers*, C-21, No. 4 (April 1972), pp. 399-402.
7. S. G. Chappell, "Automatic Test Generation for Asynchronous Digital Circuits," *B.S.T.J.*, 53, No. 8 (October 1974), pp. 1477-1503.
8. Y. H. Levendel and P. R. Menon, "Fault Simulation Methods—Extensions and Comparison," *B.S.T.J.*, 60, No. 9 (November 1981), pp. 2235-58.
9. M. M. Denneau, "The Yorktown Simulation Engine," 19th Design Automation Conference, Las Vegas, Nevada, June 14-16, 1982, pp. 55-9.
10. E. Kronstadt and G. Pfister, "Software Support for the Yorktown Simulation Engine," 19th Design Automation Conference, Las Vegas, Nevada, June 14-16, 1982, pp. 60-4.
11. G. Pfister, "The Yorktown Simulation Engine: Introduction," 19th Design Automation Conference, Las Vegas, Nevada, June 14-16, 1982, pp. 51-4.

12. R. Barto and S. A. Szygenda, "A Computer Architecture for Digital Logic Simulation," *Electronic Engineering*, September 1980, pp. 35-66.
13. M. Abramovici, Y. H. Levendel, and P. R. Menon, "A Logic Simulation Machine," 19th Design Automation Conference, Las Vegas, Nevada, June 14-16, 1982, pp. 65-73.
14. S. G. Chappell, C. H. Elemendorf, and L. D. Schmidt, "LAMP: Logic Circuit Simulators" *B.S.T.J.*, 53, No. 8 (October 1974), pp. 1451-76.
15. N. D. Phillips and J. G. Tellier, "Efficient Event Manipulation, A Key to Large Scale Simulation," *IEEE 1978 Semiconductor Test Conference*, Cherry Hill, New Jersey, October 31-November 2, 1978, pp. 266-73.
16. J. G. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms," *Comm. ACM*, 18-4 (April 1975), pp. 233-30.

APPENDIX A

Architecture Description

A.1 Introduction

The simulator consists of one master and a multiplicity of slaves (processors) interconnected by a communication structure (see Fig. 2). The communication structure can be either shared or dedicated. The circuit to be simulated is partitioned into subcircuits and each subcircuit is simulated in a separate processor. The partitioning is such that the number of simultaneously active subcircuits (processors) is maximum and the number of simultaneously active elements in each subcircuit (processor) is minimum while keeping the communication from being a bottleneck.

The circuit to be simulated is initially modeled in the general-purpose computer, partitioned and loaded into the slave memories. The general-purpose computer performs all functions of input, output, and user interactions. The simulation is carried out in the multiprocessor simulator.

The simulator can be programmed to output intermediate results to the general-purpose computer. It also can be interrupted by the general-purpose computer for intermediate results. The user can ask for information about a simulation run while it is in progress (e.g., the status of a variable) and make certain run-time decisions like continuing simulation, applying extra input vectors, or stopping. After simulating each vector input, the simulation results and any other user requested information are sent to the general-purpose computer. User-requested information typically includes output values of elements (monitored points) at specific simulated times or under some other specified conditions. The general-purpose computer formats this information for suitable presentation to the user.

A.2 Multiprocessor operation

At the beginning of each simulation cycle the master sends primary input values (if any) to the appropriate slaves using the communication structure. The master then issues a start signal to the slaves. This

signal informs the slaves to start processing for the next simulation cycle. During the processing of a simulation cycle a slave unit may generate data for the other slaves or the master. The data are sent to the destination slave or the master using the communication structure. Only data for the subsequent time interval are transferred between the slaves to reduce the amount of information sent over the communication structure, thus minimizing the communication overhead. Therefore, the scheduled time is not sent.

Each slave informs the master when it has finished processing and transferring data for the current simulation cycle. When all slaves have informed the master about their completion of processing for the current simulated time interval, and also the master has finished transferring any primary input values scheduled for the next simulated time interval to the slaves, the master issues a start signal to the slaves for the next simulation cycle.

There are two signal lines between each slave unit and the master. The master signals the slaves using a START signal and the slaves signal the master using the DONE line. The START line from the master initiates processing for the next simulation cycle. The DONE line will become one when all the slaves have finished processing for the current simulation cycle.

A.3 Slave unit

Each slave unit consists of a processing unit (PU), an input FIFO buffer (IFB), an output FIFO buffer (OFB), an input data sequencer (IDS), and an output data sequencer (ODS).

The slave unit PUs perform the actual fault simulation. The PU stores any data it has for other PUs or the master in the OFB. The ODS makes a request for the communication structure if there are any data to be transferred from the OFB. The ODS of the slave, if granted the use of the communication structure, takes data from the OFB and sends it over the communication structure to other slaves or the master. The data are received by the IDS of the destination slave or the master. Any data received by an IDS are put in its IFB. End of Data (EOD) flags are used to separate data streams since a PU can be writing new data to the OFB before its ODS has finished transferring current data, and similarly, an IDS can be receiving new data in the IFB before its PU has finished reading current data.

The slave unit PU operation can be described in terms of two essentially concurrent processes, namely the simulation cycle (execution of simulation algorithm) and the communication cycle (communication of events). Data generated during a simulation cycle are transferred as they are generated in a corresponding communication cycle (see Fig. 5). The algorithm executed by the Slave Unit PU is

similar to the traditional parallel simulation algorithm used on general-purpose computers except that it is modified to operate on a data structure distributed over several processors (see Appendix C).

A communication cycle is the period in between two START commands issued from the master. This cycle is phased with respect to the simulation cycle as shown in Fig. 9.

A.4 Master processor

The master processor is the interface between the general-purpose computer and the simulator. Its main functions are to keep track of simulated time, keep the slaves in synchronism, supply the slaves with primary input values, and gather the primary output values from the slaves. The configuration of the master is similar to that of a slave unit and is shown in Fig. 10. It consists of a central processing unit (CPU) with some local memory, an IFB, an OFB, an IDS, and an ODS.

A.5 Communication structure

The communication structure is used as a medium for transferring data between the slaves and between the slaves and the master. Either a shared or a dedicated structure can be used for the multiprocessor

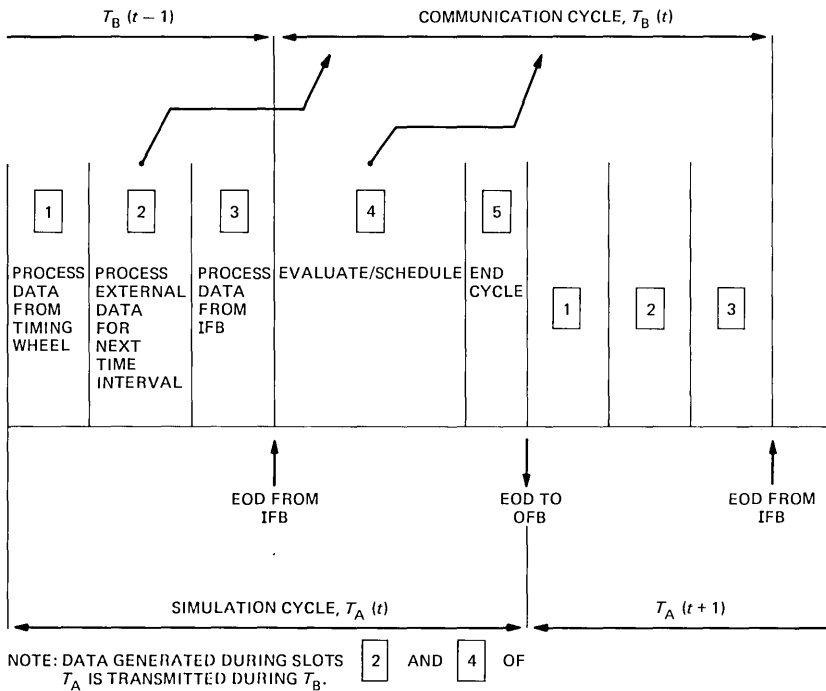


Fig. 9—Relationship between a simulation cycle and communication cycle.

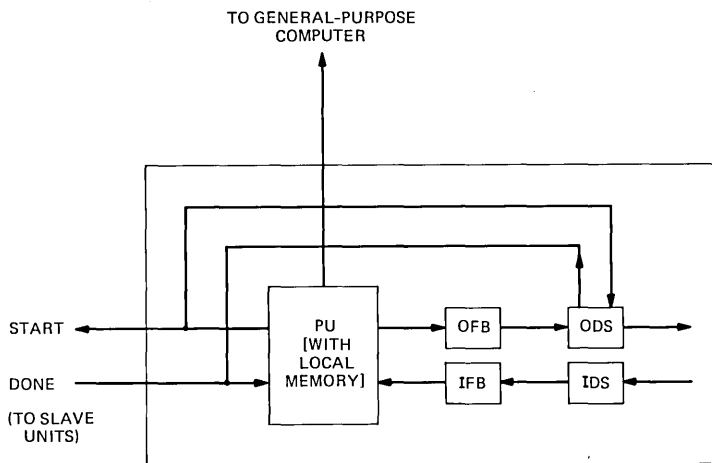


Fig. 10—Master configuration.

simulator. The details of a communication structure based on time-shared parallel bus and the cross-point matrix are discussed in detail in Ref. 5.

APPENDIX B

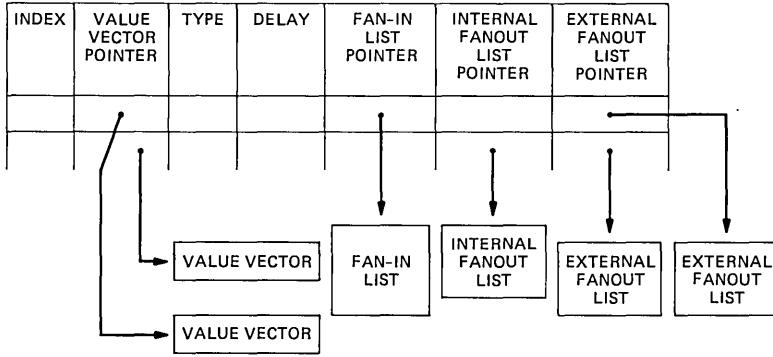
Data Structure for Parallel Method

The following data tables are used by each PU for its operation (see also Fig. 11):

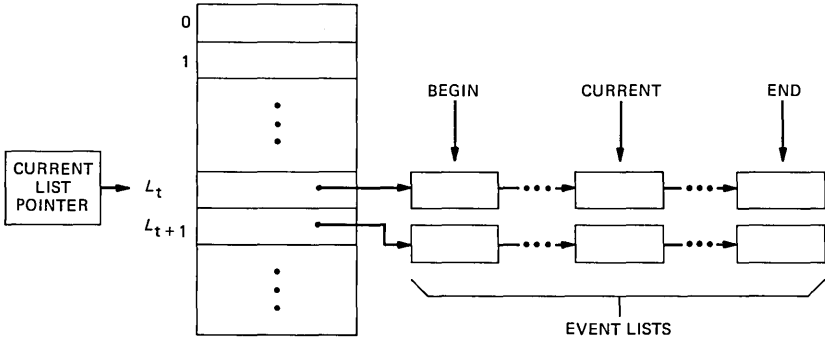
1. Element table—This table contains the interconnection data and signal values for the circuit. For each element it contains the value vector, type, delay, fan-in list pointer and corresponding fan-in list, internal fanout list pointer and fanout list, external fanout list pointer and external fanout list. The value vector consists of a word pair. The first bit of each word of the word pair represents the good machine, while the remaining bits represent faulty machines. Three values can be represented: 0, 1, and X. If multiword parallel simulation is required, then additional word pairs are used to represent more faulty machines. The internal fanout list pointer and corresponding fanout lists give the fanout that remains in the subcircuit. The external fanout list pointer and corresponding fanout lists give fanout that goes to subcircuits located in other slaves. An element may have only internal fanout, only external fanout, or both internal and external fanout. Note that storing the external fanout takes up more space than storing an internal fanout, since both the destination processor address and element index have to be stored for the external fanout.

2. Activity list—This list is used to keep track of active elements during a simulation time interval. These elements are to be evaluated.

ELEMENT TABLE



TIMING WHEEL



L_t = CURRENT LIST OF TIMING WHEEL
 L_{t+1} = NEXT LIST OF TIMING WHEEL

Fig. 11—Data tables for PU operation (parallel simulation).

3. Timing wheel—This data area contains the events that are scheduled in the future. A large amount of work has been done in this area.^{15,16}

APPENDIX C

Parallel Simulation Algorithm

1. Update data from timing wheel
 - for each event in list L_t of timing wheel
 - begin
 - update value vector pointer in Element Table;
 - for each fanout f of updated element
 - if activity flag of f not set {
 - set activity flag;

```

        put  $f$  on activity list;
    }
end
deallocate space on Timing Wheel;
2. Prepare external events for next time interval
for each event in list  $L_{t+1}$  of Timing Wheel
begin
    if element has external fanout {
        for each external fanout {
            move destination processor address to OFB;
            move element number to OFB;
            for each word of value vector associated with event
                move word to OFB;
        }
        if element does not have internal fanout also
            deallocate space on Timing Wheel;
    }
end
3. Update data from IFB
enable interrupts from IFB;
for each interrupt received from IFB
begin
    if event is not EOD flag {
        for each word of value vector associated with event
            move word to Element Table;
        for each fanout  $f$  of updated element
            if activity flag of  $f$  not set {
                set activity flag;
                put  $f$  on activity list;
            }
        }
    }
    else {
        disable interrupts from IFB;
        reset activity flags in Element Table;
        go to step 4 (Evaluate/Schedule);
    }
end
(The EOD flag signifies end of data present in the IFB for the
current simulation cycle. The EOD flag is loaded by the IDS
when it receives a START signal from the master.)
4. Evaluate/schedule
for each entry in activity list
begin

```



```

inject faults, if any, on inputs;
for each set of words associated with input value vectors
of element
  begin
    calculate corresponding output word;
    if output word changed from previous value
      set change flag;
    end
inject faults, if any, on outputs;
if change flag set or faults injected {
  schedule event on Timing wheel;
  if (element delay is 1 && element has external fanout)
    for each external fanout {
      move destination processor address to OFB;
      move element number to OFB;
      for each word of value vector associated with event
        move word to OFB;
    }
  }
}
end
5. End cycle
  store EOD flag in OFB;
  increment  $t$ ;
  go to STEP 1.

```

APPENDIX D

Glossary

- BIU—bus interface unit
- CPU—central processing unit (part of a master unit)
- EOD—end of data (flag)
- IDS—input data sequencer
- IFB—input first in first out (FIFO) buffer
- L_p —number of word pairs used in parallel simulation
- LAMP—logic analyzer for maintenance planning
- LSI—large-scale integration
- L_t —current list of timing wheel
- N —average number of active elements per simulation cycle in true value simulation
- N_T —total number of active elements during all passes of a simulation
- N_f —average number of active elements per simulation cycle in fault simulation with 16 faults per pass
- ODS—output data sequencer

- OFB—output FIFO buffer
- PU—processing unit (part of a slave unit)
- T—time required to simulate a given set of faults
- c —average number of elements in an element string
- f_i —average fan-in of an element
- f_o —average fanout of an element
- j —number of events for which the channel is found busy in cross-point matrix
- k —imbalance factor due to nonideal partitioning
- n —number of processors in the multiprocessor simulator
- n_o —number of processors required for optimum operation of multiprocessor simulator
- t_l —length of simulation cycle for a single processor simulator
- t_a —time required to process one active element
- t_{br} —bus release time for a parallel bus structure
- t_{brg} —bus request and grant time for a parallel bus structure
- t_c —total communication time during one simulation cycle
- $t_{c(\text{bus})}$ —total communication time during one simulation cycle for a single parallel bus structure
- $t_{c(\text{matrix})}$ —total communication time during one simulation cycle for a matrix structure
- $t_{c(\text{mbus})}$ —total communication time during one simulation cycle for a multiple parallel bus structure
- t_{da} —data acknowledge time for a parallel bus structure
- t_{ds} —address and data setup time for a parallel bus structure
- t_m —length of simulation cycle for a multiprocessor simulator
- t_o —average processing time, t_p , for number of processors $n = n_o$
- t_p —average processing time per processor during one simulation cycle, where average processing time consists of the time required to process all active elements and schedule resulting events
- VLSI—very large-scale integration
- w —number of words to be transferred across the communication structure for one active event

AUTHORS

Ytzhak H. Levendel, B.S.E.E., 1971, Technion-Israel; M.S.C.S., 1974, The Weitzman Institute of Science; Ph.D., 1976, University of Southern California; Bell Laboratories, 1976—. Mr. Levendel has done research in fault diagnosis and until lately was involved in the development of a logic and test design aid system. He is now a Supervisor in 5ESS.™ Member, IEEE, Eta Kappa Nu.

Premachandran R. Menon, B.S. (Electrical Engineering), 1954, Banaras Hindu University; Ph.D. (Electrical Engineering), 1962, University of Washington; Bell Laboratories, 1963—. Mr. Menon has done research in switching

theory and fault diagnosis and is currently involved in the development of a logic simulation system. Recipient of the Distinguished Technical Staff Award; member, IEEE.

Suresh H. Patel, B.E. (Electrical), 1975, University of Zambia; M.S.E.E., 1976, Illinois Institute of Technology; Association of American Railroads, 1977–1981; Bell Laboratories, 1981—. At the Association of American Railroads Mr. Patel was involved in hardware/software design, development and testing of a multi-microcomputer-based instrumentation system for freight trains. He was also involved in development of analytical and computer models on track circuit. At Bell Laboratories he is a member of the Processor Design Department. Mr. Patel was a part-time instructor at Illinois Institute of Technology during 1977–1978.

Two New Kinds of Biased Search Trees

By J. FEIGENBAUM* and R. E. TARJAN†

(Manuscript received May 20, 1983)

In this paper, we introduce two new kinds of biased search trees: biased, a , b trees and pseudo-weight-balanced trees. A biased search tree is a data structure for storing a sorted set in which the access time for an item depends on its estimated access frequency in such a way that the average access time is small. Bent, Sleator, and Tarjan were the first to describe classes of biased search trees that are easy to update; such trees have applications not only in efficient table storage but also in various network optimization algorithms. Our biased a , b trees generalize the biased 2, b trees of Bent, Sleator, and Tarjan. They provide a biased generalization of B -trees and are suitable for use in paged external memory, whereas previous kinds of biased trees are suitable for internal memory. Our pseudo-weight-balanced trees are a biased version of weight-balanced trees much simpler than Bent's version. Weight balance is the natural kind of balance to use in designing biased trees; pseudo-weight-balanced trees are especially easy to implement and analyze.

I. INTRODUCTION

The following problem, which we shall call the *dictionary problem*, occurs frequently in computer science. Given a totally ordered universe U , we wish to maintain one or more subsets of U under the following operations, where R and S denote any subsets of U and i denotes any item in U :

access (i, S)—If item i is in S , return a pointer to its location. Otherwise, return a special **null** pointer.

* Research done partly while a summer employee of Bell Laboratories and partly while a graduate student supported by Air Force grant AFOSR-80-042.

† Bell Laboratories.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

insert (i, S)—Insert i in S , assuming it is not previously there.

delete (i, S)—Delete i from S .

join (R, S) (two-way join)—Return the set consisting of the union of R and S , assuming that every item in R precedes every item in S . This operation destroys R and S , and can be regarded as a concatenation of R and S .

split (i, S)—Split S into three sets L, I , and R , where L and R are the sets of items strictly smaller and strictly larger than i , respectively, and $I = \{i\}$ if i is in S (three-way split), $I = \emptyset$ if i is not in S (two-way split).

One way to solve the dictionary problem is to store the items of each set in the external nodes of a search tree in left-to-right order, one item per external node. To guide the operations, the search tree also contains auxiliary items, called *keys*, in the internal nodes. The worst-case access time in a search tree is proportional to the depth of the tree. By imposing any one of a number of well-known *balance conditions* on the tree, we can guarantee that its depth is $O(\log n)$, where n is the number of items it contains. Such a balance condition can be maintained during update operations by performing appropriate local rearrangements of the tree. With balanced search trees, each of the dictionary operations has an $O(\log n)$ time bound. Examples of balanced trees include height-balanced (AVL) trees,¹ 2, 3 trees,² *B*-trees,³ and trees of bounded balance.⁴

In many applications of search trees the access frequencies of different items are different, and we would like our data structure to take this into account. To deal with this problem formally we assume that each item i has a known weight w_i providing an estimate of the access frequency. The *biased dictionary problem* is that of implementing the dictionary operations so that operations on heavier items are faster than those on lighter items. In particular, when representing a set S as a search tree, we wish to bias the tree so as to approximately minimize its total weighted depth $\sum_{i \in S} w_i d_i$, where d_i is the depth of the external node containing item i , while preserving the ability to update the tree rapidly. In addition to the five dictionary operations, we allow the following operation for changing the weight of an item:

reweight (i, w, S)—Redefine the weight of item i in set S to be w .

The following theorem, due to Shannon, gives a lower bound on the total weighted depth of a search tree:

*Theorem 1:*⁵ If T is a search tree for a set S and every node of T has no more than b children, then

$$\sum_{i \in S} w_i d_i \geq \sum_{i \in S} \frac{w_i}{W} \log_b \frac{W}{w_i},$$

where $W = \sum_{i \in S} w_i$ is the total weight of the items in S .

In light of Theorem 1, our goal is to devise classes of search trees that are easy to update and have $d_i = O(\log_b W/w_i)$ for every item i . We call $O(\log_b W/w_i)$ the ideal access time of item i .

Bent, Sleator, and Tarjan⁶⁻⁸ have devised several kinds of such biased search trees. Our work is an extension of theirs. A thorough discussion of previous work by others on the biased dictionary problem may be found in Ref. 8.

In their running time analyses, Bent, Sleator, and Tarjan used a technique called *amortization*, which we shall also use. The idea of amortization is to average the running time of individual operations over a (worst-case) sequence of operations. As a tool in deriving amortized time bounds we use *credits*. A credit will pay for one unit of computing time. To each operation we allocate a certain number of credits, called the *credit time* or *amortized time* of the operation. If a given operation does not need all its credits, we can save them for use in later operations; if an operation needs more than its share of credits we can use those previously saved. In any analysis using credits, the objective is to prove that an arbitrary sequence of operations can be performed without running out of credits.

Three points about amortization using credits are worth making. First, credits are a way of charging earlier operations for later ones. If a credit analysis is successful, we can assert that any sequence of operations requires an amount of actual time that is at most a constant multiplied by the sum of the credit times of the individual operations; slow operations are only possible if there are corresponding earlier fast ones. Second, although the word “average” appears in the description of the technique, it is not the usual kind of average-case analysis, and in particular we make no probabilistic assumptions; we obtain worst-case bounds holding for *any* sequence of operations. Third, credits serve as a kind of “potential energy”: we place them in regions of search trees that may cause abnormally long update operations. This idea may illuminate the credit invariants we define below.

The remainder of the paper consists of three sections. In Sections II and III, we define and analyze biased a, b trees, which generalize the biased 2, b trees of Ref. 8 and provide a biased version of B -trees. Biased a, b trees are a form of biased tree appropriate for paged external memory; earlier forms of biased trees are more appropriate for internal memory. In Section IV we introduce pseudo-weight-balanced trees, which give a biased version of weight-balanced trees much simpler than Bent’s earlier version.⁶ Weight balance is the natural kind of balance to use in designing a biased search tree; pseudo-weight-balanced trees are especially easy to implement and analyze.

We shall assume that the reader is familiar with search trees. In particular we shall not discuss the use and updating of keys, and we

shall draw freely on the results and techniques of Ref. 8. We shall use the terminology of Ref. 8 except that we use “external node” in place of “leaf”. When appropriate we shall regard a node x of a search tree as denoting the entire subtree rooted at x , with the context resolving whether a node or a tree is meant. The **null** node denotes the empty search tree. We denote the parent of a node x by $p(x)$; $p(x) = \mathbf{null}$ if x is a tree root.

II. LOCALLY BIASED a, b TREES

Our first class of biased trees uses height balance to guarantee fast access and variable node size to allow easy updating. The class is parameterized by two positive integer constants a and b such that $2 \leq a \leq \lceil b/2 \rceil$. The integer b specifies the maximum allowed number of children of an internal node. If an internal node has at least a children, we say it is *filled*; otherwise it is *underfilled*. (An external node is always filled.) Ideally, we would like every node to have at least a children; since in our scheme this is impossible to achieve, we allow underfilled nodes but treat them specially.

If x is a node in a search tree, we define its *weight* $w(x)$ to be the sum of the weights of all items in descendants of x . (We use the convention that every node is a descendant of itself.) We define the *rank* $s(x)$ of x recursively by $s(x) = \lfloor \log_a w(x) \rfloor$ if x is an external node, $s(x) = 1 + \max\{s(y) \mid p(y) = x\}$ if x is an internal node. A node x is *minor* if x is not a root and either $s(x) < s(p(x)) - 1$ or x is underfilled. All other nodes are *major*. A *locally biased a, b tree* is a search tree with the following properties (see Fig. 1):

1. Every internal node has at least two and at most b children;

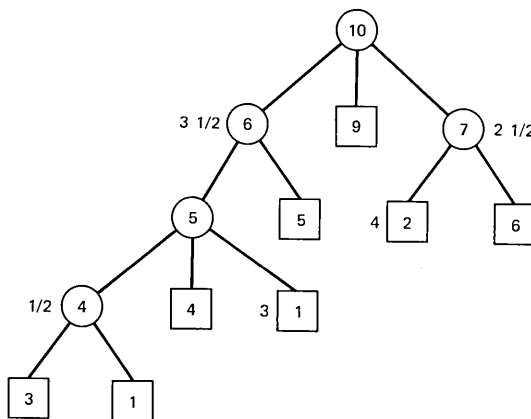


Fig. 1—A biased 3, b tree. The numbers in nodes are ranks, and the numbers beside nodes are credits for the credit invariant.

2. If x is a minor node, any adjacent sibling of x is both major and external. When a node x has this property, we say the tree is *locally biased* at x .

We call two nodes r -compatible if they can be adjacent children of a node of rank r in a biased a, b tree. That is, two nodes are r -compatible if both have rank at most $r - 1$ and either at least one has rank $r - 1$ and is external or both have rank $r - 1$ and at least a children each.

If $a = 2$ we obtain exactly the biased 2, b trees of Bent, Sleator, and Tarjan. Our main new idea is in the definition and handling of underfilled nodes. Our first theorem guarantees that a, b trees have ideal access time if b is chosen appropriately.

Lemma 1: Consider any biased a, b tree. If x is an external node, $a^{s(x)} \leq w(x) < a^{s(x)+1}$. If x is an internal node with at least one minor child, $a^{s(x)-1} \leq w(x)$. If x is an internal node with k children, $k'a^{s(x)-2} \leq w(x)$, where $k' = m\{k, a\}$.

Proof: The first part of the lemma is immediate from the definition of rank. The second part follows from the first part and property 2: if x has a minor child, it must have another child that is major and external. We prove the third part by induction on the height of x . If x has a minor child, then $k'a^{s(x)-2} \leq a^{s(x)-1} \leq w(x)$ by the second part of the lemma. Otherwise, all children of x are major. Let y be a child of x . If y is external, or internal with a minor child, then $a^{s(x)-2} = a^{s(y)-1} \leq w(y)$. Otherwise, y has at least a major children, and by the induction hypothesis $a^{s(x)-2} = a \cdot a^{s(y)-2} \leq w(y)$. Summing over the k children of x gives $ka^{s(x)-2} \leq w(x)$. \square

Lemma 2: If x is an external node in a biased a, b tree of total weight W , the depth of x is at most $\log_a W/(w(x)) + 3$.

Proof: Let r be the root of the tree and d the depth of x . Since the rank increases by at least one from child to parent, $d \leq s(r) - s(x)$. Lemma 1 implies $\log_a w(x) \leq s(x) + 1$ and $\log_a W = \log_a w(r) \geq s(r) - 2$. Combining inequalities gives the lemma. \square

Theorem 2: A biased a, b tree has ideal access time, with a constant factor proportional to $\log_a b$.

Proof: Immediate from Lemma 2. \square

According to Theorem 2, to minimize the access time in a, b trees, we should choose b as small as possible. The requirement $b \geq 2a - 1$ is necessary to allow efficient updating. The best choice of b seems to be $2a - 1$ or $2a$. The choice $b = 2a$ allows purely up-down updates (see the end of Section III). The choice $b = 2a - 1$ gives a biased version of ordinary B -trees.³ Any other choice gives a biased version of “hysterical” or “weak” B -trees.⁹⁻¹¹ Note also that Theorem 2 gives a worst-case example, not an amortized bound on the access time.

As Ref. 8 shows, all the update operations on search trees can be carried out using one or more joins. Our next task is thus to define a join algorithm for biased a, b trees.

Algorithm 1: local join (x, y). Join two locally biased a, b trees with roots x and y , assuming that all items in tree x precede all items in tree y .

Case 0— $x = \text{null}$ or $y = \text{null}$. If $x = \text{null}$, return y ; if $y = \text{null}$, return x .

Case 1— $s(x) \geq s(y)$ and x and y are $(s(x) + 1)$ -compatible, or $s(x) \leq s(y)$ and x and y are $(s(y) + 1)$ -compatible. Create a new node with x and y as its children and return the new node.

Case 2— $s(x) = s(y)$ and x and y are not $(s(x) + 1)$ compatible. Let u be the rightmost child of x and v the leftmost child of y (see Fig. 2a). Perform *join*(u, v), letting w be the root of the resulting tree. If $s(w) < s(x)$, construct a node z whose children are those of x not including u , the node w , and the children of y not including v . If $s(w) = s(x)$, construct a node z whose children are those of x not including u , those of w , and those of y not including v (see Fig. 2b). In either case, if z has more than b children, split it into two nodes z' and z'' with z as parent, dividing the old children of z as evenly as possible between z' and z'' (see Fig. 2c). Return z .

Case 3— $s(x) > s(y)$ and x and y are not $(s(x) + 1)$ -compatible. Let u be the rightmost child of x (see Fig. 2d). Perform *join* (u, y), letting v be the root of the resulting tree. If $s(v) < s(x)$, replace u as a child of x by v . If $s(v) = s(x)$, replace u as a child of x by the set of children of v . If x now has more than b children, split it into two nodes x' and x'' with x as parent, dividing the old children of x as evenly as possible between x' and x'' . Return x .

Case 4— $s(x) < s(y)$ and x and y are $(s(y) + 1)$ -compatible. Symmetric to Case 3.

Theorem 3: Given two biased a, b trees with roots x and y , the join algorithm produces a biased a, b tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$. In the latter case, the root of the new tree has exactly two children.

Proof: An easy induction on rank shows that the root of the tree produced by the join has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$ and that in the latter case the root has exactly two children. Furthermore, it is clear that every internal node in the new tree has at least two and at most b children. All that remains is to show that any two adjacent siblings in the new tree are allowed to be adjacent by property 2. We prove this by induction on rank, using the same cases as in the algorithm.

Case 1—Assume without loss of generality that $s(x) \geq s(y)$. Since

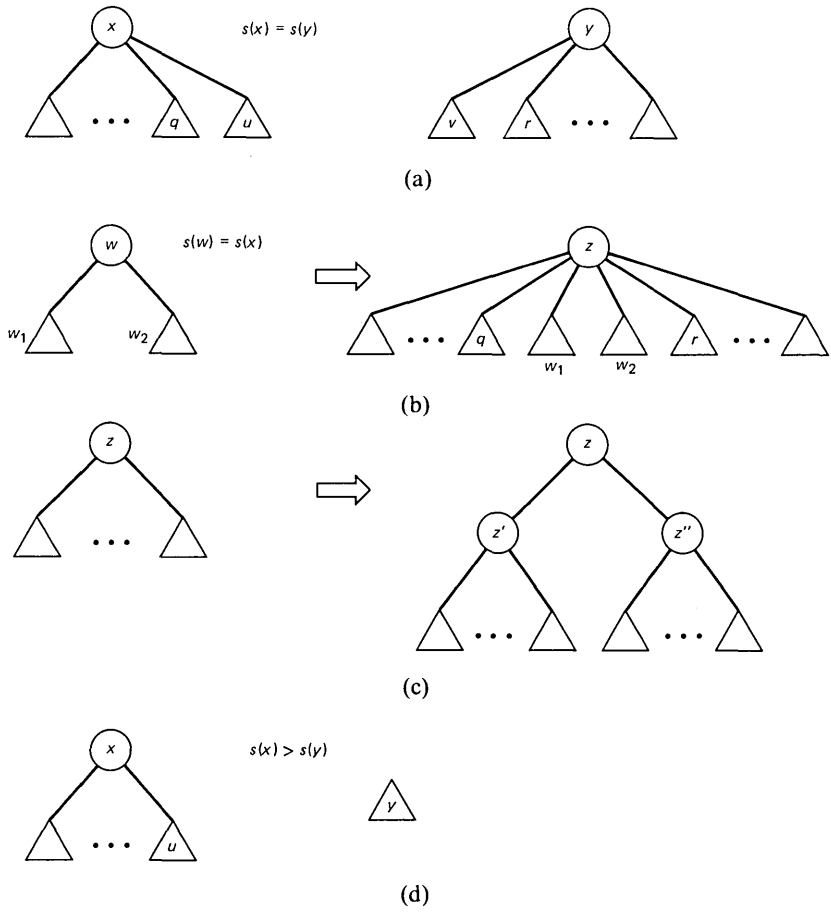


Fig. 2—Cases of the join algorithm. (a) Situation at the beginning of Case 2. (b) Recursive call $join(u, v)$ produces a tree of rank $s(x)$ with root w . (c) Division of an overfilled root. (d) Case 3.

x and y are $(s(x) + 1)$ -compatible, the new tree is locally biased at x and y .

Case 2—For the moment, ignore the split of z if it takes place. Since x and y are not $(s(x) + 1)$ -compatible, neither x nor y is external, and u and v exist. Suppose the left sibling of u , say q , is minor. Then u is major and external, which means that the join of u and v will immediately terminate in Case 1, and the new right sibling of q will be u . The symmetric statement holds for the right sibling of v . Finally, suppose w is minor, i.e., $s(w) < s(x) - 1$ or w has fewer than a children. Then both u and v must be minor as children of x and y , respectively, and both adjacent siblings of w in the new tree will be major and external. Thus the tree before the split has property 2.

Splitting z preserves property 2 since both new children of z will have at least a children of their own (this is where we use $b > 2a - 1$) and each will have rank $s(x)$. (Property 2 implies that before the split, of any two adjacent children of z , at least one has rank $s(z) - 1$).

Case 3—Similar to but simpler than Case 2. Case 4 is symmetric. \square

To make our timing analysis as similar as possible to the one in Ref. 8 for biased 2, b trees, we shall assume that credits can be divided in half, and that half a credit will pay for the work in one call of *join*, not counting the work in the recursive call. We use the following credit invariant: A nonroot node x holds $s(p(x)) - s(x) - 1$ credits, plus an additional half credit if x is underfilled.

Theorem 4: The join algorithm runs in $O(|s(x) - s(y)|)$ amortized time. Specifically, carrying out the join while preserving the invariant takes $|s(x) - s(y)| + 1$ credits in Case 1 or 2, $|s(x) - s(y)| + 1/2$ credits in Case 3 or 4.

Proof: We use induction on rank and a surprisingly complicated case analysis.

Case 1—Assume without loss of generality that $s(x) \geq s(y)$ and that if $s(x) = s(y)$, then x is external or filled. We need half a credit for the work in the join and at most $s(x) - s(y) + 1/2$ credits to place on y , for a total of $s(x) - s(y) + 1$.

Case 2—The split of z , if it occurs, does not affect the credit invariant; therefore we ignore it. Assume without loss of generality that $s(u) \geq s(v)$. There are three subcases:

Subcase 2a— $s(w) = s(x)$ and *join* (u, v) is a Case 1 join. The credits originally on u and v suffice to maintain the credit invariant on u and v after the join of x and y is completed. We have one credit on hand to join x and y ; we spend half for the work in the outer call and half for the work in the inner call.

Subcase 2b— $s(w) = s(x)$ and *join* (u, v) is not a Case 1 join. To perform the join of x and y we are given one credit and can obtain at least $2s(x) - s(u) - s(v) - 2$ from u and v , for a total of $2s(x) - s(u) - s(v) - 1$. We need half a credit for the work in the outermost call and at most $s(u) - s(v) + 1$ for the recursive call, for a total of at most $s(u) - s(v) + 3/2$. The difference between what we have and what we need is $2(s(x) - s(u)) - 5/2$. Since $s(x) - s(u) \geq 1$, we must find at most an additional half credit to spend.

If *join* (u, v) is a Case 2 join, and $s(x) - s(u) = 1$, then either u or v is underfilled and yields an extra half credit. If *join* (u, v) is a Case 3 join, we save half a credit on the join. Thus, in any case we can obtain the needed half credit.

Subcase 2c— $s(w) < s(x)$. As in Subcase 2b we have at least $2s(x) - s(u) - s(v) - 1$ credits to perform the join of x and y . We need

one half for the work in the outermost call, at most $s(u) - s(v) + 1$ for the recursive call, and at most $s(x) - s(w) - 1/2$ to place on w , for a total of at most $s(x) + s(u) - s(v) - s(w) + 1$. The difference between what we have and what we need is $s(x) - s(u) + s(w) - s(v) - 2$. Since $s(x) - s(u) \geq 1$ and $s(w) \geq s(v)$, we have enough credits unless $s(x) - s(u) = 1$ and $s(w) = s(v)$, in which case we must find an extra credit.

Suppose $s(x) - s(u) = 1$ and $s(w) = s(v)$. Since $s(w) = s(v)$, $\text{join}(u, v)$ is not a Case 1 join, and u is internal. If $\text{join}(u, v)$ is a Case 2 join, then both u and v are internal. Either both u and v are underfilled, giving us two additional half credits, or one of u and v is underfilled and w is filled, giving us an extra half credit from u or v and saving a half credit that does not need to go on w . The only other possibility is that $\text{join}(u, v)$ is a Case 3 join, which saves us half a credit.

Furthermore in this case either u is underfilled or w is filled, either giving us an extra half credit from u or saving us a half credit on w . Thus in all cases we obtain the necessary extra credit.

Case 3—We ignore the possible split of x , which does not affect the credit invariant. There are two subcases:

Subcase 3a— $s(v) = s(x)$. To perform the join we are given $s(x) - s(y) + 1/2$ credits and can obtain at least $s(x) - s(u) - 1$ more from u , for a total of $2s(x) - s(u) - s(y) - 1/2$. We need one half for the outermost call and at most $|s(u) - s(y)| + 1$ for the recursive call, for a total of $|s(u) - s(y)| + 3/2$. The difference between what we need and what we have is $2(s(x) - \max\{s(u), s(y)\}) - 2 \geq 0$.

Subcase 3b— $s(v) < s(x)$. To perform the join we have at least $2s(x) - s(u) - s(y) - 1/2$ credits. We need $|s(u) - s(y)| + 3/2$ for the outermost and recursive calls, plus at most $s(x) - s(v) - 1/2$ to place on v , for a total of $s(x) + |s(u) - s(y)| - s(v) + 1$. The difference between what we have and what we need is $s(x) - \max\{s(u), s(y)\} + s(v) - \max\{s(u), s(y)\} - 3/2$. We have enough credits unless $s(x) - \max\{s(u), s(y)\} = 1$ and $s(v) = \max\{s(u), s(y)\}$, in which case we need an extra half credit.

Suppose $s(x) - \max\{s(u), s(y)\} = 1$ and $s(v) = \max\{s(u), s(y)\}$. Then $\text{join}(u, y)$ is not a Case 1 join. If it is a Case 2 join, then either u is underfilled, giving us an extra half credit, or v is filled, saving us a half credit on v . If it is a Case 3 join, we save half a credit on the join. Thus in all cases we obtain the necessary half credit.

Case 4—Symmetric to Case 3. □

It is useful to restate Theorem 4 as follows. We say a biased a, b tree with root x is *cast to rank k* if it satisfies the credit invariant and has $k - s(x)$ credits on its root. Theorem 4 implies that if x and y are

the roots of two biased a, b trees cast to a rank $k > \max\{s(x), s(y)\}$, then they can be joined, without using additional credits, to produce a tree cast to rank k .

We can implement a split as a sequence of joins, exactly as described in Ref. 8. The following algorithm splits at an item i in the tree:

Algorithm 2: split (i, r). Split the biased a, b tree with root r at item i , assumed to be in the tree.

Locate the node x containing item i . Initialize the current node to be $p(x)$ and the previous node to be x . Initialize the left and right trees to empty; they will contain the items smaller than and larger than i , respectively. Repeat the following step until the current node is **null** (the previous node is the root of the tree):

Split Step—Delete every child of the current node to the left of the previous node. If there is one such child, join it to the left tree; if there are two or more such children, give them a common parent and join the resulting tree to the left tree. Repeat this process with the children to the right of the previous node, joining the resulting tree to the right tree. Remove the previous node as a child of the current node and destroy it if it is not x . Make the current node the new previous node and its parent the new current node.

Theorem 5: The split algorithm is correct and takes $O(s(r) - s(x))$ credit time, where x is the node containing item i .

Proof: Correctness follows immediately from the correctness of the join algorithm. The time bound follows as in Ref. 8; we shall sketch the idea. Let *cur*, *prev*, *left*, and *right* be the current node, the previous node, and the roots of the left and right trees, respectively. An induction shows that $s(\text{prev}) \geq \max\{s(\text{left}), s(\text{right})\}$ just before each split step. Another induction shows that by allocating $O(s(\text{cur}) - s(\text{prev}))$ credits to a split step, we can carry out the step while preserving the invariant that the left and right trees are cast to a rank of $s(\text{prev}) + 1$. That is, the amortized time associated with a single step of the split is proportional to the rank difference of two consecutive nodes along the split path. If we sum over all split steps, then the sum telescopes, and we obtain the time bound in the statement of the theorem. \square

Splitting at an item not in the tree is just like splitting at an item in the tree, except that the initialization is different. Let r be the root of the tree, i the item at which the split is to take place, i^- and i^+ the largest item in the tree less than i and the smallest item in the tree greater than i , respectively, and x the *handle* of i , defined to be the nearest common ancestor of the external nodes containing i^- and i^+ . We can locate x by searching down the path from r to x if appropriate keys are stored in the tree (see Ref. 8). To carry out the split, we combine all children of x whose descendants contain items less than i

to form the initial left tree and all other children of i to form the initial right tree. Then we initialize the previous and current nodes to be x and $p(x)$, respectively, and repeat the split step until the current node is **null**. Such a split also runs in $O(s(r) - s(x))$ credit time.

We can implement insert, delete, and reweight as combinations of splits and joins: an insertion is a two-way split followed by two joins, a deletion is a three-way split followed by one join, and a weight change is a three-way split followed by two joins. Using the same kind of analysis as in Ref. 8, we obtain the following credit times for these three operations (we have stated the bounds in terms of weights rather than ranks):

$$\text{insert } (i, x): O\left(\log_a \left(\frac{w(x) + w_i}{\min\{w_{i^-} + w_{i^+}, w_i\}} \right)\right),$$

where i^- and i^+ are as defined above.

$$\text{delete } (i, x): O\left(\log_a \left(\frac{w(x)}{w_i} \right)\right).$$

$$\text{reweight } (i, w, x): O\left(\log_a \left(\frac{\max\{w(x), w\}}{\min\{w_i, w\}} \right)\right),$$

where $w(x)$ and w_i are as defined before the weight change.

Remark: In all the time bounds derived in this section and the next the constant factor is proportional to b . \square

III. GLOBALLY BIASED a, b TREES

Local bias is sufficient to guarantee good amortized but not good worst-case running times for the dictionary operations. If it is important that every single operation be fast, we need a stronger balance condition. Figure 3 shows how a split can take more actual time than

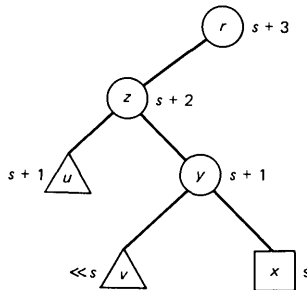


Fig. 3—Locally biased α, β tree that cannot be split at x in actual time $O(s(r) - s(x))$. Not all children of y, z , and v are shown. Join of u and v can take an unbounded amount of time.

its credit time, and illustrates why local bias is not enough: a minor node that is the leftmost child of its parent has a constraint on its right but not on its left side, and symmetrically for a minor node that is the rightmost child of its parent. To overcome this problem we introduce globally biased a, b trees.

If x is a node in a search tree, we define x^+ to be the external node containing the smallest item greater than the largest item in a descendant of x . Symmetrically, x^- is the external node containing the largest item less than the smallest item in a descendant of x . If x is on the rightmost path of the tree, x^+ is undefined; if x is on the leftmost path, x^- is undefined. A *globally biased a, b tree* is a search tree with two properties (see Fig. 4):

1. Every internal node has at least two and at most b children.
2. If x is a minor nonroot node and x^+ is defined, $s(x^+) \geq s(p(x)) - 1$; if x^- is defined, $s(x^-) \geq s(p(x)) - 1$. When a node x has this property, we say the tree is *globally biased at x* .

Global bias implies local bias. Thus globally biased a, b trees have ideal access time. We can join two globally biased a, b trees using almost the same join algorithm as in Section II; the only difference is that the conditions determining the cases are different. We shall call the algorithm in Section II *local join* to distinguish it from the following *global join* algorithm:

Algorithm 3: global join (x, y). Join two globally biased a, b trees with roots x and y , assuming that all items in tree x precede all items in tree y .

In each case, proceed as in the corresponding case of the local join algorithm:

Case 0— $x = \text{null}$ or $y = \text{null}$.

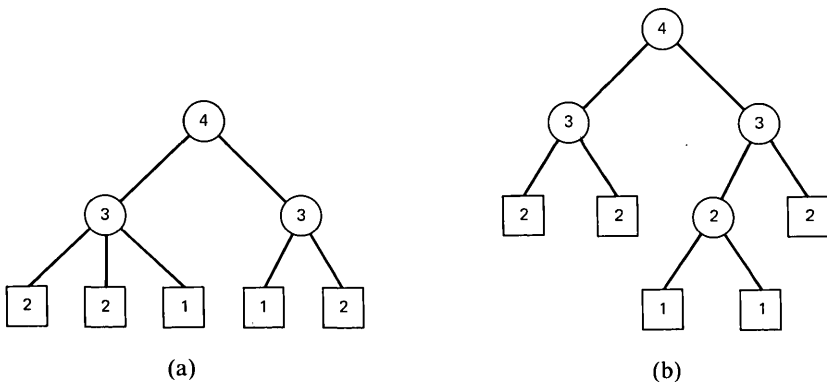


Fig. 4—Two biased 2, 3 trees with the same external nodes. Numbers in nodes are ranks. (a) Locally biased tree. (b) Globally biased tree.

Case 1— $s(x) \geq s(y)$ and x is external, or $s(x) \leq s(y)$ and y is external.

Case 2— $s(x) = s(y)$ and both x and y are internal.

Case 3— $s(x) > s(y)$ and x is internal.

Case 4— $s(x) < s(y)$ and y is internal.

The only difference between this and the local join algorithm is that if x and y have the same rank and both are internal with at least a children, we apply Case 2 instead of Case 1. The algorithm is identical to the join algorithm for globally biased 2, b trees given in Ref. 8.

Theorem 6: The global join algorithm is correct.

Proof: As in the proof of Theorem 3, we use induction on rank and a case analysis.

Case 1—Immediate.

Case 2—Ignore the split of z if it takes place. Let q be the left sibling of u (recall that u is the rightmost child of x). Suppose q or one of the nodes on the rightmost path descending from q is minor. Let this minor node be r . The join changes neither $p(r)$ nor r^+ and thus preserves global bias at r . The symmetric statement holds for the right sibling of v (recall that v is the leftmost child of y). Suppose w (the join of u and v) or one of the nodes on the leftmost path descending from w is minor. Let this minor node be r . There must be a corresponding minor node r' on the leftmost path descending from u , such that $s(p(r)) \leq s(p(r'))$. Since the original tree is globally regular at r' , the new tree must be globally regular at r . The symmetric statement holds for the rightmost path descending from w . Thus the new tree is globally biased before the split. The split preserves global bias.

Case 3—Similar to but simpler than Case 2. Case 4 is symmetric. \square

Theorem 7: The worst-case running time of the global join algorithm is $O(\max\{s(x), s(y)\} - \max\{s(u), s(v)\})$, where u is the rightmost external descendant of x and v is the leftmost external descendant of y .

Proof: The global join algorithm descends rank by rank concurrently along the rightmost path descending from x and the leftmost path descending from y , until reaching a leaf; then it ascends. The theorem follows. \square

We split a globally biased a, b tree in exactly the same way as a locally biased a, b tree, *using local rather than global joins*. This method not only produces globally biased trees, it has a worst-case time bound equal to the amortized bound given in Theorem 5.

Theorem 8: Algorithm 2 (or its variant for an item not in the tree) correctly splits a globally biased a, b tree with root r at a node x in $O(s(r) - s(x))$ worst-case time.

Proof: The proof is the same as the corresponding proof for globally

biased 2, b trees given in Ref. 8. For completeness, we sketch it here. Let x_1, x_2, \dots, x_k be the roots of the trees joined to form the final left tree. Let $y_1 = x_1$ and for $i = 2, \dots, k$ let y_i be the root of the tree formed by executing join (x_i, y_{i-1}) . With this definition y_k is the final left tree. Each node x_i is either a child of an ancestor of x , say a_i , in the original tree, or the newly constructed parent of a set of children of such a node a_i ; furthermore a_{i+1} is a proper ancestor of a_i for $i = 1, \dots, k - 1$. Consider a node x_i for $i \geq 2$. If x_i is a child of a_i , global bias implies x_i is a major child, for otherwise its right sibling is external, which is impossible since this right sibling has x_1 or its children as ancestors. Thus $s(x_i) = s(a_i) - 1$. If x_i is the new parent of children of a_i , then $s(x_i) = s(a_i)$. It follows that $s(x_i) \leq s(x_{i+1})$ for $i = 1, \dots, k - 1$. As noted in the proof of Theorem 5, an induction shows that $s(y_i) \leq s(a_i)$ for $i = 1, \dots, k$; if $i \geq 2$ and $s(x_i) = s(a_i)$, x_i has at most $b - 1$ children, and the join of x_i and y_{i-1} cannot split x_i . Thus if $i \geq 2$, $s(a_i) - 1 \leq s(x_i) \leq s(y_i) \leq s(a_i)$.

Consider the join of x_{i+1} and y_i . The join will descend rank by rank along the rightmost path from x_{i+1} and the leftmost path from y_i . Global bias in the original tree implies that if the descent from x_{i+1} encounters a minor node z (other than the root of x_{i+1}), the leftmost external node of y_i will have rank at least $s(p(z)) - 1$ and the join will immediately terminate. Thus the join either terminates by reaching an external descendant of x_{i+1} of rank at least $s(y_i)$, in which case the global bias of the joined tree is immediate, or it reaches an internal descendant, say z , of x_{i+1} of rank exactly $s(y_i)$. Now we need a similar but more complicated statement about the leftmost path from y_i . There are several cases.

Case 1— $s(x_i) < s(a_i)$ and x_i is minor in the original tree. This can only happen if $i = 1$, i.e., $x_i = y_i$. Global bias implies that the rightmost external descendant of x_{i+1} has rank at least $s(a_i) - 1$. The join descends along the path from x_{i+1} to this external node and then ascends.

Case 2— $s(x_i) < s(a_i)$, x_i is major, and $s(x_i) < s(y_i)$. In this case y_i is also major and the leftmost paths descending from x_i and y_i , not including x_i and y_i themselves, are identical. If z is underfilled, y_i is external, and the join terminates in Case 1. If z is filled, y_i is either external or filled, and the join also terminates in Case 1.

Case 3— $s(x_i) < s(a_i)$, x_i is major, and $s(x_i) < s(y_i)$. In this case the left child of y_i is major and the join stops descending either at rank $s(y_i)$ (if y_i is external or filled) or at rank $s(y_i) - 1$ (if y_i is internal).

Case 4— $s(x_i) = s(a_i)$. In this case the leftmost paths descending from x_i and y_i , not including x_i and y_i themselves, are identical. If y_i is external or filled, the join will stop descending at rank $s(y_i)$. If y_i is underfilled, the join will stop at rank $s(y_{i-1})$; either the rightmost child

of z or the leftmost child of y_i is external of rank $s(y_{i-1})$, or both are filled and of rank $s(y_{i-1})$.

In all cases the global bias of the original tree implies that the joined tree is globally biased. This means that the final left tree is globally biased. Furthermore, the time required for joining x_{i+1} and y_i is $O(s(a_{i+1}) - s(a_i))$ in all cases. Summing over all joins gives a time bound of $O(s(r) - s(x))$ to form the final left tree. A symmetric argument applies to the right tree. \square

If we implement insertion, deletion, and weight change as described in Section II, we obtain the following worst-case time bounds for the various operations on globally biased a, b trees (see Ref. 8):

$$\text{split } (i, r): O\left(\log_a \left(\frac{w(r)}{w_i}\right)\right)$$

if i is in the tree, or

$$O\left(\log_a \left(\frac{w(r)}{w_{i^-} + w_{i^+}}\right)\right)$$

if i is not in the tree, where i^- and i^+ are the items before and after i in the tree, respectively.

$$\text{insert } (i, x): O\left(\log_a \frac{w(x)}{w_{i^-} + w_{i^+}} + \log_a \frac{w(x) + w_i}{w_i}\right).$$

$$\text{delete } (i, x): O\left(\log_a \frac{w(x)}{w_i} + \log_a \frac{W'}{w_{i^-} + w_{i^+}}\right),$$

where W' is the weight of the tree root after the deletion.

$$\text{reweight } (i, w, x): O\left(\log_a \frac{w(x)}{w_i} + \log_a \frac{W'}{w}\right),$$

where $w(x)$ and w_i are as defined before the weight change, and W' is the weight of the tree root after the change.

As compared with the amortized time bounds for locally biased a, b trees, the worst-case bounds for globally biased a, b trees are larger for *join* and *delete* and the same for the other operations.

We conclude our discussion of biased a, b trees with two remarks. First, if $b \geq 2a$ we can implement either local or global join in an iterative, purely top-down fashion by preemptively splitting nodes with b children as they are encountered. By extending this idea we can implement all the operations top-down. This is a reason to choose $b = 2a$ over $b = 2a - 1$.

Second, for appropriate large values of a and b , biased a, b trees offer a biased alternative to B -trees. One of the advantages of B -trees is that there are no underfilled nodes except tree roots; thus if nodes

are stored one node per page in fixed-size pages, the storage efficiency is at least 50 percent, not counting root pages. Biased a, b trees do not share this property. We leave open the problem of devising a space-efficient version of biased a, b trees.

IV. PSEUDO-WEIGHT-BALANCED TREES

In biased a, b trees, we maintain balance through a height constraint. However, there are other possible balance constraints, such as weight balance. Nievergelt and Reingold⁴ defined trees of bounded balance by imposing upper and lower bounds on the ratio *leftweight/rightweight* at each internal node, where the left and right weights count the number of items in the left and right subtrees, respectively. Bent developed a biased version of weight-balanced trees.⁶ However, his data structure suffers from a complicated seven-case join algorithm that needs up to three recursive calls and also uses rebalancing rotations more complicated than standard single and double rotations. In this section we introduce a form of biased weight-balanced trees much simpler than Bent's. Our simplification comes from two new ideas: we discretize the weights and allow arbitrarily bad imbalance in some situations where balancing is possible. We call our trees pseudo-weight-balanced.

We consider binary trees, in which each internal node x has exactly two children, a left child $l(x)$ and a right child $r(x)$. As in Section II we define the *weight* $w(x)$ of a node x by $w(x) = w_i$ if x is an external node containing item i , $w(x) = w(l(x)) + w(r(x))$ if x is an internal node. We define the *rank* $s(x)$ of a node x differently: $s(x) = \lfloor \lg w(x) \rfloor$. We call a binary search tree *pseudo-weight-balanced* (pwb) if it has two properties:

1. If three nodes in a row, say $x, p(x)$, and $p(p(x))$, have the same rank, then x is external and either x is a left child and $p(x)$ a right child or vice-versa (see Fig. 5a).

2. If x and $l(x)$ (symmetrically x and $r(x)$) are internal nodes of the same rank, then $w(r(l(x))) + w(r(x))$ (symmetrically $w(l(r(x))) + w(l(x))$) is at least $2^{s(x)}$ (see Figure 5b). (This property allows us to do single rotations when necessary to maintain property 1.)

The following result bounds the access time in pwb trees.

Theorem 9: A pwb tree has ideal access time for all items. Specifically, if x is an external node containing item i in a tree of total weight W , then the depth of x is at most $2 \lg(W/w_i) + 3$.

Proof: Let r be the root of the tree and d the depth of x . According to property 1, after the first step up from x , every two steps taken along the path from x to r must cause a rank increase. Thus $\lfloor (d-1)/2 \rfloor \leq s(r) - s(x) = \lfloor \lg W \rfloor - \lfloor \lg w(x) \rfloor$, which implies $d \leq 2\lfloor (d-1)/2 \rfloor + 1 \leq 2(\lg W - \lg w(x) + 1) + 1 \leq 2 \lg(W/w(x)) + 3$. \square

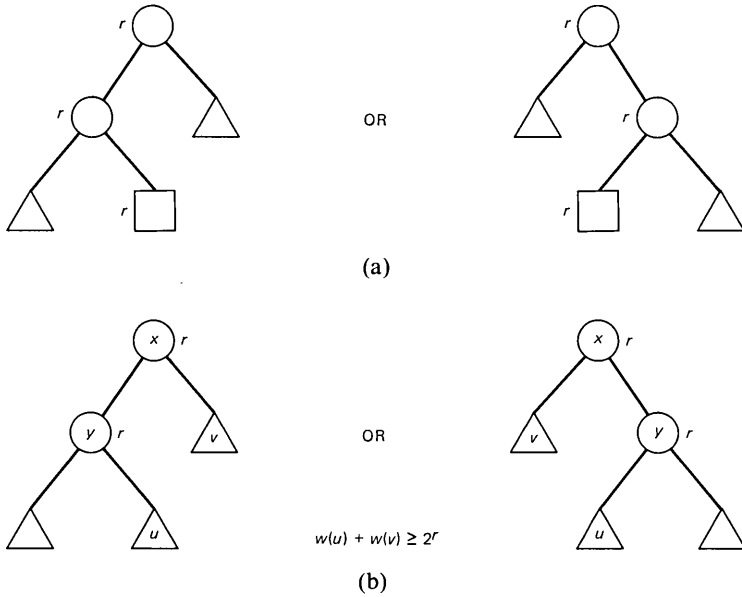


Fig. 5—Legal configurations in a pseudo-weight-balanced tree. (a) Three nodes of the same rank in a row. (b) Two internal nodes of the same rank in a row.

We join two pwb trees using the following algorithm:

Algorithm 4: join (x, y). Join two pwb trees with roots x and y , assuming that all items in tree x precede all items in tree y .

Case 0— $x = \mathbf{null}$ or $y = \mathbf{null}$. Return y if $x = \mathbf{null}$ or x if $y = \mathbf{null}$.

Case 1— $\lg(w(x) + w(y)) \geq 1 + \max\{s(x), s(y)\}$ or the heavier of x and y is an external node. Return a new root with left child x and right child y .

Case 2— $s(x) > s(y)$ and $\lg(w(x) + w(y)) < 1 + s(x)$ and x is not external. If $r(x)$ is external, or internal of rank at most $s(x) - 1$, replace $r(x)$ by *join* ($r(x), y$). Otherwise, perform a single left rotation at x and replace the right child $r(u)$ of the new root u by *join* ($r(u), y$) (see Fig. 6).

Case 3— $s(x) < s(y)$ and $\lg(w(x) + w(y)) < 1 + s(y)$. Symmetric to Case 2.

Remark: Although the join algorithm is stated recursively, it is easy to implement it in an iterative, purely top-down fashion, since the rank of a node depends only on the total weight of its descendants and not on their arrangement.

Theorem 10: Algorithm 4 produces a pwb tree of rank $\max\{s(x), s(y)\}$ or $1 + \max\{s(x), s(y)\}$.

Proof: By induction on the depth of the recursion. The definition of rank implies that the rank of the new tree is $\max\{s(x), s(y)\}$ or $1 +$

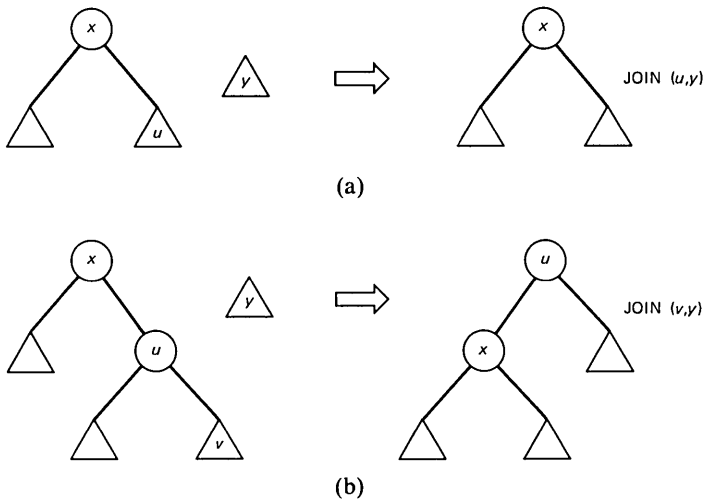


Fig. 6—Case 2 of the join algorithm for pwb trees. (a) Node $u = r(x)$ external or $s(u) < s(x)$: no rotation. (b) Node u internal and $s(u) = s(x)$: rotation.

$\max\{s(x), s(y)\}$. In the latter case, the join must have executed Case 1 and both children of the new root must have rank smaller than the root's rank. Case 1 obviously constructs a tree with properties 1 and 2. In Case 2, property 2 guarantees that the single rotation, if it occurs, creates a pwb tree. If the tree produced by the recursive join has rank less than $s(x)$, the overall joined tree clearly has properties 1 and 2. This is also true if the tree produced by the recursive join has rank $s(x)$, by the observation above. \square

In analyzing the running time of Algorithm 4, we use the following credit invariant: Any node x contains $\max\{0, s(p(x)) - s(x) - 1\}$ credits.

Theorem 11: Algorithm 4 runs in $O(|s(x) - s(y)|) = O(\lg(w(x) + w(y))/\min\{w(x), w(y)\})$ amortized time. Specifically, if we assume without loss of generality that $s(x) \geq s(y)$, performing the join while maintaining the credit invariant takes at most $s(x) - s(y) + 1$ credits.

Proof: We consider the same cases as in the algorithm.

Case 1—We need one credit to build the new tree and either $s(x) - s(y)$ or $s(x) - s(y) - 1$ to establish the invariant on y , for a total of at most $s(x) - s(y) + 1$.

Case 2—Suppose the single rotation does not take place. We have on hand $s(x) - s(y) + 1$ tokens for performing the join plus $s(x) - s(r(x)) - 1$ from $r(x)$, for a total of $2s(x) - s(y) - s(r(x))$. We need one token for performing the outermost call of *join* plus $\max\{s(r(x)), s(y)\} - \min\{s(r(x)), s(y)\} + 1$ for the recursive call plus $s(x) - \max\{s(r(x)), s(y)\} - 1$ to establish the invariant on the root of the

tree returned by the recursive call, for a total of $s(x) - \min\{s(r(x)), s(y)\} + 1 \leq 2s(x) - s(y) - s(r(x))$, since $s(x) > \max\{s(r(x)), s(y)\}$. Exactly the same argument applies if the rotation does take place, since the rotation preserves the credit invariant. \square

The algorithm for splitting a pwb tree is almost identical to but simpler than the algorithm for splitting a biased a, b tree.

Algorithm 5: split (x, r). Split a pwb tree with root r at a node x .

Initialize the current node cur , the previous node $prev$, and the left and right nodes $left$ and $right$ to be $p(x)$, x , $l(x)$, and $r(x)$, respectively. Repeat the following step until $cur = \text{null}$:

Split step—If $prev = l(cur)$, replace $right$ by $join(right, r(cur))$; otherwise, replace $left$ by $join(l(cur), left)$. Remove $prev$ as a child of cur and destroy it if it is not x . Replace $prev$ and cur by cur and $p(cur)$, respectively.

This algorithm is the same as that described by Bent, Sleator, and Tarjan⁸ for splitting biased binary trees, and indeed will work for any class of binary search trees for which a join algorithm is known.

Theorem 12: The amortized time of *split* (x, r) is

$$O\left(\lg\left(\frac{w(r)}{w(x)}\right)\right).$$

More precisely, performing the split while maintaining the credit invariant takes $O(s(r) - s(x))$ credits, where x is the node containing item i .

Proof: The definition of ranks implies that $s(prev) \geq \max\{s(left), s(right)\}$ before each split step. An easy induction shows that if we allocate $O(s(cur) - s(prev) + 1)$ credits to each split step, we can carry out the step while maintaining the credit invariant on the trees $left$ and $right$ and in addition maintaining $2s(prev) - s(left) - s(right)$ credits on hand. Summing over all split steps and using property 2 gives the theorem. \square

Using the appropriate combinations of *join* and *split*, we obtain the same amortized time bounds as in Section II (with binary logarithms) for insertion, deletion, and weight change in pwb trees. Pseudo-weight-balanced trees are a very simple version of locally biased trees, competitive with the biased binary trees presented in Ref. 8. We have been unable to devise a globally biased version of pwb trees and leave this as an open problem.

REFERENCES

1. G. M. Adelson-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," Soviet Math. Dokl., 3, No. 5 (September 1962), pp. 1259-62.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

3. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Info.*, 1, No. 3 (1972), pp. 173-89.
4. J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *SIAM J. Comput.*, 2 (1973), pp. 33-43.
5. N. Abramson, *Information Theory and Coding*, New York: McGraw-Hill, 1963.
6. S. W. Bent, "Dynamic weighted data structures," Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA, 1982.
7. S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased 2-3 trees," *Proc. Twenty-First Annual IEEE Symp. on Foundations of Computer Science*, October 13-15, 1980, pp. 248-54.
8. S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased search trees," unpublished work.
9. S. Huddleston and K. Mehlhorn, "Robust balancing in B-trees," *Lecture Notes in Computer Science*, 104 (1981), Berlin: Springer-Verlag, pp. 234-44.
10. S. Huddleston and K. Mehlhorn, "A new data Structure for representing sorted lists," *Acta Info.*, 17, No. 2 (1982), pp. 157-84.
11. D. Maier and S. C. Salveter, "Hysterical B-trees," *Info. Proc. Letters*, 12, No. 4 (August 1981), pp. 199-205.

AUTHORS

Joan Feigenbaum, B.A. (Mathematics), 1981, Harvard University. Ms. Feigenbaum is a graduate student in the Computer Science Department of Stanford University and holds a grant from the Bell Laboratories Graduate Research Program for Women. She spent the summers of 1980, 1981, and 1982 at Bell Laboratories. During the summer of 1982, she was a member of the Mathematical Foundations of Computing Department. Her current research is in the areas of data structures and cryptography. Member, Phi Beta Kappa.

Robert E. Tarjan, B.S. (Mathematics), 1969, California Institute of Technology; M.S., 1971, and Ph.D., 1972 (Computer Science), Stanford University; Cornell University, 1972-1973; University of California, Berkeley, 1973-1975; Stanford University, 1975-1980; Bell Laboratories, 1980—. Mr. Tarjan is a member of the Mathematical Foundations of Computing Department, where he has been studying the design and analysis of efficient data structures and combinatorial algorithms. Member, ACM, SIAM, AAAS, Tau Beta Pi, and Sigma Xi.

An Algebraic Theory of Relational Databases

By T. T. LEE*

(Manuscript received December 10, 1982)

In this paper we present a theory of relational database systems based on the partition lattice, which represents a new mathematical approach to the structure of relational database systems. A partition lattice can be defined for any given relation. This partition lattice is shown to be a meet-morphic image of the Boolean algebra of subsets of the attribute set. The partial ordering in the lattice is proved to be equivalent to the concept of functional dependency, and thus Armstrong's axioms for functional dependencies are proved. We solve the problem of finding the list of all keys by seeking the prime implicants of the Boolean function associated with the principal ideals generated by the attributes. We demonstrate the properties of the Boyce-Codd Normal Form (BCNF), and give a modified algorithm for synthesizing an information-lossless BCNF based on the principal filter. The necessary and sufficient conditions for multivalued dependency (MVD) are given in terms of a lattice equation, and the inference rules of MVD are proved. The necessary and sufficient conditions for join dependency (JD) are given; consequently, we can prove the known result that acyclic join dependency (AJD) is equivalent to a set of MVDs. The concept of data independence is introduced, and is extended to conditional independence and mutual independence. We established this algebraic theory of relational databases in the same spirit that the theory of probability was constructed. We present a comparison that demonstrates the similarities.

I. INTRODUCTION

The existing theory of relational databases is based on Codd's relational model of data.^{1,2} This relational database theory can be considered to be the study of data dependencies (or independencies).

* Bell Laboratories.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

The theory was initiated by Codd with the introduction of the concept of functional dependency; Codd observed that this concept can be used to design better, normalized, database schemes. The advantage of normalized database schemes is that they remove the possibility of updating anomalies caused by undesirable data dependencies.²⁻⁵

In the existing theory of logical database design, functional dependencies are input constraints that must always hold in the relation.⁶ In the present paper, however, we take a different approach. We assume that for a particular database designer, there exists a (finite) universal relation $R[\Omega]$ for a given set of attributes Ω , such that any relation T on Ω is a subset of $R[\Omega]$. Furthermore, each subset X of Ω corresponds to an equivalence relation (partition) on the set of tuples of $R[\Omega]$. That is, if two tuples in $R[\Omega]$ have the same X value, then they are in the same equivalence class. With this approach, the concept of functional dependency becomes equivalent to the refinement partial ordering of the partition lattice. The partitions on the (finite) set of tuples of the universal relation $R[\Omega]$ can then be considered as the fundamental constraints, from which the functional dependencies (partial ordering) can be derived. Consequently, with our approach, the functional dependencies are inherent properties of the universal relation $R[\Omega]$. The input constraints of course must be consistent with the inherent properties within the database.

Another kind of data dependency, proposed by Fagin⁷ and Zaniolo,⁸ is the multivalued dependency, which includes functional dependency as a special case. Multivalued dependency is the necessary and sufficient condition for the lossless-join decomposition of a relation into two subrelations, such that the original relation can be regenerated by the (natural) join operation.⁷⁻¹¹ Using the partition lattice we propose, we can formulate multivalued dependency as a lattice equation (see Section VI). We show that the axioms for functional dependencies¹² and the inference rules of multivalued dependencies¹³ can all be proved as theorems within the framework of partition lattice theory. We show how the concept of join dependency^{10,11,14} is connected to multivalued dependency. We give the necessary and sufficient condition for join dependency and, consequently, we can prove the known result that the acyclic join dependency is equivalent to a set of MVDs.^{15,16} We also introduce the concept of data independence, and the extension to conditional independence and mutual independence of sets of attributes.

The problem of listing all the keys of a relation is solved by using the concept of principal ideals in lattice theory. One form of a relation having desirable properties is the Boyce-Codd Normal Form (BCNF); we show that the concept of the principal filter (dual ideal) can be used to produce information-lossless Boyce-Codd Normal Forms.

Both the theoretical foundation and the practical application of the existing theory of relational databases appear to be fragmented. This paper shows that all the diverse kinds of data dependencies can be formulated within the lattice theory, which has the important advantage of unifying the theory of relational databases into a coordinated whole. Because of this, it would appear that future work in relational databases should be conducted using lattice theory as the basic framework.

The establishment of this algebraic theory of relational databases is done in the same spirit as the construction of the theory of probability, although probability theory is of course unrelated to database theory. We are convinced that the lattice theory could play a role in the theory of relational databases similar to the role that measure theory plays in the theory of probability.¹⁷

The basic notion of relational databases is defined in Section II, and the partition lattice of the relation is introduced in Section III. The problem of listing all keys is solved in Section IV, where the Boolean functions associated with the principal ideals are defined. The properties of the Boyce-Codd Normal Form are studied in Section V, where we present a modified algorithm for synthesizing information-lossless BCNFs based on the principal filters. Section VI is devoted to the proof of equivalence between multivalued dependency and a lattice equation. Section VII discusses join dependency and acyclic join dependency. Finally, in Section VIII we outline a possible direction for future research, as well as a comparison that shows the similarities between probability theory and the algebraic theory of relational databases. In Appendix A we list the laws of lattice theory for reference. The proofs of the axioms for functional and multivalued dependencies are listed in Appendix B.

Unless otherwise stated, we refer to the universal relation as simply "the relation" in the remainder of this paper.

II. RELATIONS

An *attribute* is a symbol taken from a finite set $\Omega = \{A_1, A_2, \dots, A_n\}$. For each attribute A there is a set of possible values called its *domain*, denoted $\text{DOM}(A)$. We will use capital letters from the beginning of the alphabet (A, B, \dots) for single attributes, and capital letters from the end of the alphabet (X, Y, \dots) for sets of attributes. For a set of attributes $X \subseteq \Omega$, an X -value x is an assignment of values to the attributes of $X = \{A_{i_1}, A_{i_2}, \dots, A_{i_k}\}$ from their respective domains. The notation XY will be used to represent the union of two arbitrary sets of attributes $X, Y \subseteq \Omega$.

A *relation* R on the set of attributes $\Omega = \{A_1, \dots, A_n\}$ is a subset of the Cartesian product $\text{DOM}(A_1) \times \dots \times \text{DOM}(A_n)$. The elements

(rows) of R are called *tuples*. A relation R on $\{A_1, \dots, A_n\}$ will be denoted by $R[A_1 \dots A_n]$. Similarly, if R is defined on the union of sets (X_1, X_2, \dots, X_m) , then the notation $R[X_1 \dots X_m]$ will be used. A relation can be visualized as a table whose columns are labeled with attributes and whose rows depict tuples. The ordering of the rows and columns is immaterial. The *cardinal* of R is the total number of tuples in R and is denoted by $|R|$.

Let t be a tuple in $R[\Omega]$. For $X \subseteq \Omega$, $t[X]$ denotes the tuple that contains the components of t corresponding to the attributes of X . The *projection* of R on X , denoted by $R[X]$, is defined as follows:

$$R[X] = \{t[X] \mid t \in R\}.$$

Similarly, the *conditional projection* of R on X by a Y -value y , where $Y \subseteq \Omega$, is defined as follows:

$$R_y[X] = \{t[X] \mid t \in R \text{ and } t[Y] = y\}.$$

Let $R[XZ]$ and $S[XZ]$ be relations where X, Y , and Z , are disjoint sets of attributes. The *join* (natural join) of R and S , denoted by $R \mid \times \mid S$, is the relation $T[XYZ]$ whose attributes are XYZ , and is defined as follows:

$$\begin{aligned} T[XYZ] &= R[XZ] \mid \times \mid S[YZ] \\ &= \{(x, y, z) \mid (x, z) \in R \text{ and } (y, z) \in S\}. \end{aligned}$$

The join can also be defined as the union of a collection of Cartesian products:

$$\begin{aligned} T[XYZ] &= R[XZ] \mid \times \mid S[YZ] \\ &= \{R_z[X] \times S_z[Y] \times (z) \mid (z) \in R[Z] \cap S[Z]\}. \end{aligned}$$

Let R be a relation on the set of attributes Ω . We may have two sets of attributes $X, Y, \subseteq \Omega$, such that for any two tuples $t_1, t_2 \in R$, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$. We say then that X *functionally determines* Y in R , and denote this fact by $X \rightarrow Y$. A functional dependency (FD) $X \rightarrow Y$ is *trivial*, meaning it holds in all relations, if $Y \subseteq X$. Note that FDs enjoy the projectivity and inverse projectivity properties.^{3,4} For sets $X, Y \subseteq \Omega' \subseteq \Omega$, the FD: $X \rightarrow Y$ is valid in $R[\Omega]$ iff it is valid in $R[\Omega']$.

We say that a set of relations $\{R[\Omega_1], \dots, R[\Omega_n]\}$ has the *information-lossless join* property if $\Omega = \Omega_1 \dots \Omega_n$ and

$$R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n].$$

If the set $\{R[\Omega_1], \dots, R[\Omega_n]\}$ does not have this property, we say that it has a *lossy join*.¹⁴ An important property of functional dependency¹⁰ is that if FD: $X \rightarrow Y$ is valid in $R[\Omega]$ then

$$R[\Omega] = R[(\Omega - Y)X] \mid \times \mid R[XY].$$

This property will be discussed in more detail in Section VI.

III. THE RELATION LATTICE

If S is a nonempty set, then a subset ρ of $S \times S$ is called a *binary relation* on S . The *product* of two binary relations $\rho, \rho' \subseteq S \times S$ is defined as:

$$\rho \circ \rho' = \{(a, b) \in S \times S \mid \exists c \in S \text{ such that } (a, c) \in \rho, (c, b) \in \rho'\}.$$

We say that a relation ρ on S is *reflexive* if $(a, a) \in \rho$ for every a in S ; that ρ is *symmetric* if $\rho^{-1} = \rho$, i.e., if

$$(\forall a, b \in S), \quad (a, b) \in \rho \text{ implies } (b, a) \in \rho;$$

and that ρ is *transitive* if $\rho \circ \rho \subseteq \rho$, i.e., if

$$(\forall a, b, c \in S), \quad (a, b) \in \rho \text{ and } (b, c) \in \rho \text{ imply } (a, c) \in \rho.$$

A binary relation is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

A family $\pi = \{B_i \mid i \in I\}$ of subsets, called *blocks* of S , is said to form a *partition* of S if the following conditions hold:

1. Each B_i is nonempty
2. For all $i \neq j$ in I , $B_i \cap B_j = \emptyset$
3. $\cup\{B_i \mid i \in I\} = S$.

The two apparently different notions of “equivalence relation” and “partition” are interchangeable: Let ρ be an equivalence relation on a set S . Then the family $\alpha_\rho = \{b \mid (a, b) \in \rho\}$ of subsets of S is a partition of S . Conversely, if $\pi = \{B_i \mid i \in I\}$ is a partition of S , then the relation $\{(a, b) \mid (\exists i \in I), (a, b) \in B_i\}$ is an equivalence relation on S .

If ρ is an equivalence relation (partition) on S , we shall sometimes write apb as an alternative to $(a, b) \in \rho$. The sets α_ρ that form the associated partition of the equivalence relation are called ρ -*classes*. The set of ρ -classes is called the *quotient set* of S by ρ and is denoted by S/ρ .

A binary relation \leq on the set S is a *partial ordering* of S if and only if \leq is reflexive; antisymmetric, i.e., if

$$(\forall a, b \in S), \quad a \leq b \text{ and } b \leq a \text{ imply } a = b;$$

and transitive. A set S with a partial ordering \leq is called a *partially ordered set* (poset) and it is denoted by the pair (S, \leq) .

Let (S, \leq) be a poset and let T be a subset of S . Then, $a \in S$ is the *greatest lower bound* (g.l.b.) of T iff

1. $(\forall t \in T), a \leq t$.

2. $(\forall t \in T), a' \leq t$ implies $a' \leq a$.

Similarly, $a \in S$ is the *least upper bound* (l.u.b.) of T iff

1. $(\forall t \in T), t \leq a$.

2. $(\forall t \in T), t \leq a'$ implies $a \leq a'$.

A *lattice* is a poset in which any two elements a and b have a g.l.b., called a *meet* and denoted by $a \cdot b$, and a l.u.b., called a *join* and denoted by $a + b$. We sometimes write the meet $a \cdot b$ as ab if no confusion is created. The properties of the meet and join operations of a lattice¹⁸ are listed in Appendix A.

Let the set of all partitions π_i on S be denoted by $\Pi(S)$, and define the partial ordering on $\Pi(S)$ as follows:

If $(\forall a, b \in S), a\pi_1 b$ implies $a\pi_2 b$, then $\pi_1 \leq \pi_2$.

The poset $(\Pi(S), \leq)$ is seen to be a lattice $(\Pi(S), \cdot, +)$ with a universal lower bound $\mathbf{0} = \{B_i | i \in I\}$ such that every block B_i is a singleton, and an universal upper bound $\mathbf{1} = \{S\}$. To specify a particular partition, we list the elements, and distinguish blocks with bars and semicolons. For example, if $S = \{1, 2, 3, 4, 5\}$ and partition π on S has blocks $\{1, 3, 4\}, \{2, 5\}$, then we write $\pi = \{\bar{1}, \bar{3}, \bar{4}; \bar{2}, \bar{5}\}$. The meet and join of any two partitions $\pi_1, \pi_2 \in \Pi(S)$ can be determined as follows:

1. $(\forall a, b \in S), a\pi_1 \cdot \pi_2 b$ iff $a\pi_1 b$ and $a\pi_2 b$.

2. $(\forall a, b \in S), a\pi_1 + \pi_2 b$ iff $\exists n \in N$ and $c_0, \dots, c_n \in S$ such that $a = c_0, b = c_n$ and $c_i\pi_1 c_{i+1}$ or $c_i\pi_2 c_{i+1}$ for each $i, 0 \leq i \leq n - 1$.

A complemented distributive lattice is called a *Boolean algebra* (see Appendix A). The set of all subsets of S , called the *power set* of S , and denoted by 2^S , with the partial ordering $(\forall S_1, S_2 \in 2^S), S_1 \leq S_2$ iff $S_1 \supseteq S_2$, is a Boolean algebra $(2^S, \cdot, +, \bar{})$ with the universal bounds $\mathbf{0} = S$ and $\mathbf{1} = \emptyset$. The *dual* of a poset is the poset with the converse partial ordering on the same elements. The Boolean algebra defined above is the dual of the conventional Boolean algebra of the power set. The operations of meet and join are defined by

1. Meet (g.l.b.) $S_1 \cdot S_2 = S_1 \cup S_2$,

2. Join (l.u.b.) $S_1 + S_2 = S_1 \cap S_2$,

and the complement of $S_1 \in 2^S$ is $\bar{S}_1 = S - S_1$.

Let $\psi: L \rightarrow M$ be a function from a lattice L into a lattice M . We say ψ is a *meet-morphism* if

$$(\forall a, b \in L), \quad \psi(a \cdot b) = \psi(a) \cdot \psi(b),$$

and ψ is a *join-morphism* if

$$(\forall a, b \in L), \quad \psi(a + b) = \psi(a) + \psi(b).$$

Meet-morphisms and join-morphisms are both *isotone* (order-preserving); i.e.,

$$(\forall a, b \in L), \quad a \leq b \text{ implies } \psi(a) \leq \psi(b),$$

and any order-preserving one-to-one mapping with an inverse is an *isomorphism*.¹⁸

Let R be a relation on the set of attributes Ω . The set of all subsets of Ω , denoted 2^Ω , with the partial ordering defined by set-containment, is a Boolean algebra $(2^\Omega, \cdot, +, -)$,¹⁸ where the meet, join, and complement operations are defined as above. For every $X \in 2^\Omega$, there is an equivalence relation (partition) on the set of tuples in $R[\Omega]$ defined as follows:

Definition 1: Let R be a relation on the set of attributes Ω . Each subset of Ω is associated with a partition of the set of tuples of R . We define the function $\theta: 2^\Omega \rightarrow \prod(R[\Omega])$, which we call the *partition function* (associated with $R[\Omega]$), by

$$\theta: X \rightarrow \theta(X) = \{(t_1, t_2) \in R[\Omega] \times R[\Omega] \mid t_1[X] = t_2[X]\}. \quad \blacksquare$$

In general, the image set $Im(\theta)$ of θ is not a sublattice of $\prod(R[\Omega])$. Since $\pi_1, \pi_2 \in Im(\theta)$ implies $\pi_1 \cdot \pi_2 \in Im(\theta)$, $Im(\theta)$ is a complete lattice in its own right,²⁰ and it will be called the *relation lattice* of $R[\Omega]$, and denoted by $L(R[\Omega])$. Note that there are no duplicated tuples in $R[\Omega]$, so that $\theta(\Omega) = \mathbf{0}$. Since the tuples cannot be “differentiated” by the empty set of attributes, we define $\theta(\emptyset) = \mathbf{1}$. The universal bounds of $L(R[\Omega])$ are the same as those in $\prod(R[\Omega])$. We immediately recognize the concept of functional dependency to be equivalent to the refinement partial ordering of the partitions.

Lemma 1: Let $R[\Omega]$ be a relation on the set of attributes Ω , and let $\theta: 2^\Omega \rightarrow \prod(R[\Omega])$ be the partition function associated with $R[\Omega]$, defined above. Then

$$X \rightarrow Y \text{ iff } \theta(X) \leq \theta(Y). \quad \blacksquare$$

An immediate consequence of the above lemma is that the projection $R[X]$ of $R[\Omega]$ on X is simply the quotient of $R[\Omega]$ by $\theta(X)$, i.e., $R[X] = R[\Omega]/\theta(X)$. Thus each tuple in $R[X]$ corresponds to a $\theta(x)$ -class in $R[\Omega]/\theta(X)$ and it takes the X -value only. Note that $\theta(X) = \theta(Y)$ does not imply $R[X] = R[Y]$ because the attributes X and Y may have different sets of values.

Theorem 1: Let R be a relation on the set of attributes Ω , and let $L(R[\Omega])$ be the relation lattice of R . Then the partition function $\theta: 2^\Omega \rightarrow L(R[\Omega])$ is a meet-morphism.

Proof: We want to show that

$$\theta(XY) = \theta(X)\theta(Y), \quad \forall X, Y \in 2^\Omega.$$

Suppose $t_1\theta(XY)t_2$. Then, $t_1[XY] = t_2[XY]$, which implies

$$t_1[X] = t_2[X] \quad \text{and} \quad t_1[Y] = t_2[Y].$$

Hence,

$$t_1\theta(X)t_2 \text{ and } t_1\theta(Y)t_2.$$

By the definition of the meet operation, we have

$$t_1\theta(X)\theta(Y)t_2,$$

so that

$$\theta(XY) \leq \theta(X)\theta(Y).$$

Suppose $t_1\theta(X)\theta(Y)t_2$. Then,

$$t_1\theta(X)t_2 \text{ and } t_1\theta(Y)t_2,$$

so that

$$t_1[X] = t_2[X] \text{ and } t_1[Y] = t_2[Y],$$

and thus

$$t_1[XY] = t_2[XY].$$

Consequently,

$$t_1\theta(XY)t_2,$$

so that

$$\theta(X)\theta(Y) \leq \theta(XY).$$

Hence

$$\theta(XY) = \theta(X)\theta(Y). \quad \blacksquare$$

Note that the partition function θ is order-preserving, but it is in general not a join-morphism.* However, if $\theta(X + Y) = \theta(X) + \theta(Y)$ holds in $L[R]$, the pair (X, Y) has a special property in the relation. This is discussed further in Section VI.

It is clear now that Armstrong's axioms for functional dependencies become theorems within the framework of lattice theory. The proofs of the axioms for functional dependencies are given in Appendix B.

Let R be a relation on the set of attributes Ω , and let $\theta: \mathbf{2}^\Omega \rightarrow L[R(\Omega)]$ be the partition function associated with $R[\Omega]$. Then the relation $\theta \circ \theta^{-1}$ on $\mathbf{2}^\Omega$ defined by

$$\theta \circ \theta^{-1} = \{(X, Y) \in \mathbf{2}^\Omega \times \mathbf{2}^\Omega \mid \theta(X) = \theta(Y)\}$$

is obviously an equivalence relation. Sets in the quotient set $\mathbf{2}^\Omega / \theta \circ \theta^{-1}$ will be called θ classes.

* The join of π_1 and π_2 in $L(R[\Omega])$ may be different from their join in $\prod(R[\Omega])$. We will use the notation $\pi_1 \oplus \pi_2$ to denote the join of π_1 and π_2 in $\prod(R[\Omega])$, while $\pi_1 + \pi_2$ will denote the join of π_1 and π_2 in $L(R[\Omega])$; e.g., in Example 1 below, $\theta(E) \oplus \theta(S) = \{1, 2, 3, 4, 5, 6; 7, 8\}$, and $\theta(E) + \theta(S) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Table I—Relation $R[ECSY]$

	Employee	Child	Salary	Year
1	Hilbert	Hubert	\$35K	1975
2	Hilbert	Hubert	\$40K	1976
3	Gauss	Gwendolyn	\$40K	1975
4	Gauss	Gwendolyn	\$50K	1976
5	Gauss	Greta	\$40K	1975
6	Gauss	Greta	\$50K	1976
7	Pythagoras	Peter	\$15K	1975
8	Pythagoras	Peter	\$20K	1976

Example 1: Consider the relation R in Table I (see Ref. 7). Let $\Omega = \{E, C, S, Y\}$ be the set of attributes, where $E =$ employee, $C =$ child, $S =$ salary, $Y =$ year. Then

$$2^\Omega = \{\emptyset, E, C, S, Y, EC, ES, EY, CS, CY, SY, ECS, ECY, ESY, CYS, CESY\},$$

and

$$\begin{aligned} \theta(\emptyset = \{\overline{1, 2, 3, 4, 5, 6, 7, 8}\}) &= \mathbf{1}, \\ \theta(E) &= \{\overline{1, 2}; \overline{3, 4, 5, 6}; \overline{7, 8}\} = \pi_1, \\ \theta(C) = \theta(EC) &= \{\overline{1, 2}; \overline{3, 4}; \overline{5, 6}; \overline{7, 8}\} = \pi_2, \\ \theta(S) &= \{\overline{1}; \overline{2, 3, 5}; \overline{4, 6}; \overline{7}; \overline{8}\} = \pi_3, \\ \theta(Y) &= \{\overline{1, 3, 5, 7}; \overline{2, 4, 6, 8}\} = \pi_4, \\ \theta(ES) = \theta(EY) = \theta(SY) = \theta(ESY) &= \{\overline{1}; \overline{2}; \overline{3, 5}; \overline{4, 6}; \overline{7}; \overline{8}\} = \pi_5, \\ \theta(CS) = \theta(CY) = \theta(ECY) = \theta(ECS) = \theta(CSY) = \theta(ECSY) &= \{\overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6}; \overline{7}; \overline{8}\} = \mathbf{0}. \end{aligned}$$

The Hasse diagram^{18,21} of the relation lattice is illustrated in Fig. 1. \square

IV. LIST OF KEYS

Let R be a relation on the set of attributes Ω . We say that $X \subseteq \Omega$ is a *superkey* of R if $X \rightarrow A, \forall A \in \Omega$. If X is a superkey and no proper subset of X is a superkey, X is said to be a *key* of R .^{1,2,5}

Lemma 2: $X \subseteq \Omega$ is a superkey of R iff $\theta(X) = \mathbf{0}$.

Proof: (Necessity) Let $\Omega = \{A_1, \dots, A_n\}$, and $X \rightarrow A_i, \forall A_i \in \Omega$. Then,

$$\theta(X) \leq \theta(A_i), \quad \forall A_i \in \Omega.$$

By the definition of the meet operation, we have

$$\theta(X) \leq \theta(A_1)\theta(A_2) \dots \theta(A_n).$$

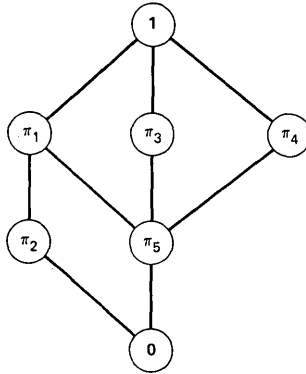


Fig. 1—Relation lattice $L(R[\Omega])$.

It follows from Theorem 1 that

$$\theta(X) \leq \theta(A_1 A_2 \cdots A_n) = \theta(\Omega) = \mathbf{0}.$$

Hence,

$$\theta(X) = \mathbf{0}.$$

(Sufficiency) Suppose $\theta(X) = \mathbf{0}$. Then

$$\theta(X) \leq \theta(A_i), \quad \forall A_i \in \Omega.$$

Hence,

$$X \rightarrow A_i, \quad \forall A_i \in \Omega. \quad \blacksquare$$

An *ideal* is a subset J of a lattice L with the properties¹⁸

1. $a \in J, x \in L$, and $x \leq a$, imply $x \in J$,
2. $a, b \in J$ implies $a + b \in J$.

For every $a \in L$, the subset of all elements “less than or equal to” a is evidently an ideal; it is called the *principal ideal* of L generated by a , and is denoted by $[a]$, i.e.,

$$[a] = \{x \in L \mid x \leq a\}.$$

Definition 2: Let R be a relation on the set of attributes $\Omega = \{A_1, \dots, A_n\}$. For each $A_i \in \Omega$, $J_i = (\theta(A_i))$ is the principal ideal of the relation lattice $L(R[\Omega])$ generated by $\theta(A_i)$. A Boolean function

$$f_i(A_1, \dots, A_n) = \sum_{\theta(X) \in J_i} X$$

defined on 2^Ω is the Boolean sum of all $X \in 2^\Omega$ such that $\theta(X) \leq \theta(A_i)$. We will call f_i the *principal ideal function* (generated by A_i). \blacksquare

This function plays a role similar to the Boolean function used in Ref. 16.

Theorem 2: Let R be a relation on $\Omega = \{A_1, \dots, A_n\}$. $X \subseteq \Omega$ is a superkey of R iff X is a product term in the expansion of the Boolean function

$$F(A_1, \dots, A_n) = \prod_{i=1}^n f_i(A_1, \dots, A_n),$$

where f_i is the principal ideal function generated by A_i .

Proof: The Boolean function $F(A_1, \dots, A_n)$ has the expansion

$$F(A_1, \dots, A_n) = \prod_{i=1}^n f_i = \sum_{\theta(X_i) \in J_i} X_1 \dots X_n.$$

We want to show that every term $K = X_1 \dots X_n$ is a superkey. Since $\theta(X_i) \in J_i = (\theta(A_i))$, it follows that

$$\theta(X_i) \leq \theta(A_i), \quad 1 \leq i \leq n.$$

From L6 in Appendix A, we have

$$\theta(X_1)\theta(X_2) \dots \theta(X_n) \leq \theta(A_1)\theta(A_2) \dots \theta(A_n).$$

It follows from Theorem 1 that

$$\theta(X_1X_2 \dots X_n) \leq \theta(A_1 \dots A_n) = \theta(\Omega) = \mathbf{0},$$

and thus

$$\theta(X_1X_2 \dots X_n) = \mathbf{0}.$$

Hence, $K = X_1X_2 \dots X_n$ is a superkey of R .

Conversely, suppose X is a superkey of R . Then

$$X \rightarrow A_i, \quad \forall A_i \in \Omega.$$

Thus,

$$\theta(X) \leq \theta(A_i), \quad 1 \leq i \leq n.$$

By the definition of the principal ideal J_i , we must have

$$\theta(X) \in J_i = (\theta(A_i)), \quad 1 \leq i \leq n.$$

It follows that $X = X \dots X$ (n times) is a product term in the expansion of $F(A_1, \dots, A_n)$. \blacksquare

It is natural to call $F(A_1, \dots, A_n)$ the *key Boolean function* of the relation $R[A_1 \dots A_n]$. Since any key X is a superkey of R , X must be a product term of the key Boolean function $F(A_1, \dots, A_n)$. Since no proper subset of X is a superkey, then by the definition of the *prime implicant* of a Boolean function,²² we have

Corollary 1: Let R be a relation on the set of attributes $\Omega = \{A_1, \dots, A_n\}$. $X \subseteq \Omega$ is a key of R iff X is a prime implicant of the key Boolean function $F(A_1, \dots, A_n)$. \blacksquare

An attribute $A \in \Omega$ is *prime* in $R[\Omega]$ if A is in any key of R ; otherwise A is *nonprime*. $A \subseteq \Omega$ is a nonprime attribute if and only if the key Boolean function is independent of A .

Theorem 3: Let R be a relation on Ω . $A \in \Omega$ is a nonprime attribute iff there exists $X \subseteq \Omega$ such that

1. $A \notin X, X \rightarrow A,$
2. $AZ \rightarrow X$ implies $Z \rightarrow X.$

Proof: (Necessity) Let $A \in \Omega$ be a nonprime attribute, and let X be any key of R . Then

$$A \notin X, \text{ and } X \rightarrow A.$$

Suppose $AZ \rightarrow X$. Then $\theta(AZ) \leq \theta(X) = \mathbf{0}$. It follows that $\theta(AZ) = \mathbf{0}$ and thus AZ is a superkey; it contains a key $K \subseteq AZ$ and $A \notin K$. We have $K \subseteq Z$, so that

$$\theta(Z) \leq \theta(K) = \mathbf{0} = \theta(X).$$

Hence,

$$Z \rightarrow X.$$

(Sufficiency) Let $\Omega = \{A_1, \dots, A_n\}, n \geq 2$. Assume there is an $X = A_2, \dots, A_m$, such that (1) $X \rightarrow A_1$, and (2) $A_1Z \rightarrow X$, implies $Z \rightarrow X$. We want to show that A_1 must be a nonprime attribute. The key Boolean function $F(A_1, \dots, A_n)$ of $R[\Omega]$ can be written in the form

$$\begin{aligned} F(A_1, \dots, A_n) &= \prod_{i=1}^n f_i = f_1 f_2 \cdots f_m f_{m+1} \cdots f_n \\ &= (f_1 f_X f_{m+1}) \cdots (f_1 f_X f_n), \end{aligned}$$

where $f_X = f_2 \cdots f_m$. For any product term Y in f_X we have

$$\theta(Y) \leq \theta(X) \leq \theta(A_1).$$

Therefore, Y must be a term in f_1 . It follows that f_1 has the form

$$f_1 = f_X + g$$

for some Boolean function g . Since $\theta(A_1Z) \leq \theta(X)$ implies $\theta(Z) \leq \theta(X)$, f_X can be written in the form

$$f_X = A_1 h + h + p = h + p$$

for some Boolean functions h and p which are independent of A_1 . Also, every $f_j, j = m + 1, \dots, n$, can be written in the form

$$f_j = f_1 e + f_X e + q$$

for some Boolean functions e and q , which are independent of A_1 . It follows that

$$\begin{aligned}
f_1 f_x f_j &= f_1 f_x (f_1 e + f_x e + q) \\
&= f_x (f_1 e + f_1 f_x e + f_1 q) \\
&= f_x (f_1 e + f_1 q) \\
&= f_1 f_x (e + q) = (f_x + g) f_x (e + q) \\
&= f_x (e + q) = (h + p)(e + q).
\end{aligned}$$

Since $h, p, e,$ and q are all independent of A_1 , we know that $f_1 f_x f_j$ is independent of A_1 for all $j = m + 1, \dots, n$. Clearly, no prime implicant of $F(A_1, \dots, A_n)$ contains A_1 , and therefore A_1 is a nonprime attribute. \square

Example 2: Consider the relation R in Example 1. To obtain the prime implicants of the key Boolean function F , we can first simplify each principal ideal function. The principal ideal functions of the relation $R[ECSY]$ are

$$\begin{aligned}
f_E &= E + C + SY, \\
f_C &= C, \\
f_S &= S + EY + CY, \\
f_Y &= Y + ES + CE,
\end{aligned}$$

and the key Boolean function is

$$\begin{aligned}
F(E, C, S, Y) &= (E + C + SY) \cdot C \cdot (S + EY + CY) \\
&\quad \cdot (Y + ES + CE) \\
&= CS + CY.
\end{aligned}$$

The sets CS and CY are the keys, and E is the only nonprime attribute. \square

V. BOYCE-CODD NORMAL FORM

Normalization is a logical database design process that can be viewed as the decomposition of a relation into a set of subrelations, such that the original relation can be regenerated by the joins of the subrelations. The purpose of decomposition is to separate the independent components into distinct relations, to avoid updating anomalies.² It is claimed in Ref. 4 that the Boyce-Codd Normal Form is one that is free of insertion and deletion anomalies. This section is devoted to the BCNF and its relation lattice. A modified algorithm for synthesizing an information-lossless BCNF⁶ is included, based on the concept of the principal filter of the relation lattice.

Recall that a functional dependency $X \rightarrow Y$ is trivial if $Y \subseteq X$. A

relation $R[\Omega]$ is said to be in Boyce-Codd Normal Form if, for all nontrivial FDs $X \rightarrow Y$, X is a superkey.^{2,4}

Definition 3: Relation $R[\Omega]$ is in BCNF if $X \rightarrow Y$ implies either

1. X is a superkey, i.e., $\theta(X) = \mathbf{0}$,
or

2. $Y \subseteq X$. ■

If a relation is in BCNF, we will show that its relation lattice has some special properties. To analyze these properties we need the concept of the principal filter.¹⁸

An ideal of the dual of the lattice L is called a *filter* of L . A subset M of L is a filter of L if

1. $a \in M$, $x \in L$, and $x \geq a$, imply $x \in M$,
2. $a, b \in M$ implies $a \cdot b \in M$.

For every $a \in L$, the subset of all elements "greater than or equal to" a is a filter; it is called the *principal filter* of L generated by a , and is denoted by $[a]$, i.e.,

$$[a] = \{x \in L \mid x \geq a\}.$$

If a and b are elements of a lattice L , where $a < b$, and there is no $c \in L$ such that $a < c < b$, then we say that a is *covered* by b (or b covers a).¹⁸ An element that covers the universal lower bound $\mathbf{0}$ of L is referred to as an *atom* of L .¹⁸

Definition 4: Let R be a relation on the set of attributes Ω , and let π be an atom of the relation lattice $L(R[\Omega])$. Let $\Omega_\pi = \{A \mid A \in \Omega, \theta(A) \geq \pi\} \subseteq \Omega$. Then the projection $R[\Omega_\pi]$ of R and Ω_π is called an *atomic projection*, and $[\pi]$ is called an *atomic filter*. ■

It is easy to verify that the relation lattice of the atomic projection $R[\Omega_\pi]$ is isomorphic to the principal filter $[\pi]$ of $L(R[\Omega])$ generated by π .

Definition 5: Let R be a relation on the set of attributes Ω , and let $\pi \in L(R[\Omega])$ be an atom. The principal filter $[\pi]$ of $L(R[\Omega])$ is called *normal* iff whenever $X \rightarrow Y$ is valid in the atomic projection $R[\Omega_\pi]$ then $Y \subseteq X$; otherwise, it is called *abnormal*. ■

Lemma 3: A relation $R[\Omega]$ is in BCNF iff every atomic filter of $L(R[\Omega])$ is normal.

Proof: (Necessity) Trivial.

(Sufficiency) Suppose $X \rightarrow Y$ and X is not a superkey, i.e., $\theta(X) \neq \mathbf{0}$. Then there must exist an atom π , such that

$$\mathbf{0} < \pi \leq \theta(X) \leq \theta(Y).$$

It follows that $X, Y \subseteq \Omega_\pi$ and that $X \rightarrow Y$ is valid in the atomic projection $R[\Omega_\pi]$, which is assumed normal. Therefore $Y \subseteq X$. ■

The join operation in the Boolean algebra $(\mathbf{2}^\Omega, \cdot, +, \bar{})$ is not always preserved by the θ mapping. But for a relation $R[\Omega]$ in BCNF, if $X, Y \subseteq \Omega$ and neither X nor Y is a superkey of $R[\Omega]$, then the join $X + Y$ is preserved by θ . We have

Corollary 2: If $R[\Omega]$ is in BCNF, $X, Y \subseteq \Omega$, $\theta(X) \neq \mathbf{0}$, and $\theta(Y) \neq \mathbf{0}$, then

$$\theta(X + Y) = \theta(X) + \theta(Y).$$

Proof: Since $X + Y \subseteq X$ and $X + Y \subseteq Y$, we have

$$\theta(X) \leq \theta(X + Y) \quad \text{and} \quad \theta(Y) \leq \theta(X + Y).$$

By definition of the join operation, we have

$$\theta(X) + \theta(Y) \leq \theta(X + Y).$$

Suppose there is a $Z \subseteq \Omega$ such that

$$\theta(X) \leq \theta(Z) \quad \text{and} \quad \theta(Y) \leq \theta(Z).$$

Given $\theta(X) \neq \mathbf{0}$ and $\theta(Y) \neq \mathbf{0}$, we have

$$Z \subseteq X \quad \text{and} \quad Z \subseteq Y.$$

Thus,

$$Z \subseteq X + Y,$$

so that

$$\theta(X + Y) \leq \theta(Z).$$

By the definition of least upper bound, we have

$$\theta(X + Y) = \theta(X) + \theta(Y). \quad \square$$

The most important characteristic of the BCNF is given in the following theorem.

Theorem 4: The relation $R[\Omega]$ is in BCNF iff every atomic filter $[\pi]$ of $L(R[\Omega])$ is isomorphic to the Boolean algebra $(\mathbf{2}^{\Omega_\pi}, \cdot, +, \bar{})$.

Proof: (Necessity) Since $[\pi]$ is a meet-morphic image of θ restricted to $\mathbf{2}^{\Omega_\pi}$, it is sufficient to show that θ is a one-to-one mapping on $\mathbf{2}^{\Omega_\pi}$. Let $X, Y \in \mathbf{2}^{\Omega_\pi}$, and $\theta(X) = \theta(Y)$. It follows that

$$\theta(X) = \theta(Y - X)\theta(Y + X) \leq \theta(Y - X),$$

which implies

$$X \rightarrow Y - X.$$

Since $\mathbf{0} < \pi \leq \theta(X)$ and $[\pi]$ is normal, we have

$$Y - X \subseteq X.$$

Hence,

$$Y \subseteq X.$$

Similarly,

$$X \subseteq Y.$$

Therefore, $X = Y$, and θ is a one-to-one mapping on 2^{Ω_π} .

(Sufficiency) Suppose $X \rightarrow Y$ is valid in $R[\Omega_\pi]$. Then $\theta(X) \subseteq \theta(Y)$. Since the inverse of an isomorphism is also order-preserving, it follows that $X \supseteq Y$. Therefore, $[\pi]$ is normal and $R[\Omega]$ is in BCNF. ■

The above theorem implies that if $[\pi]$ is normal, the only key of $R[\Omega_\pi]$ is $\theta^{-1}(\pi) = \Omega_\pi$.

It is known that any relation has a lossless-join decomposition into Boyce-Codd Normal Form, and an algorithm for determining the decomposition is given in Ref. 6. We will show how the concept of the principal filter can be used to modify this algorithm. In the algorithm for synthesizing the Third Normal Form,⁵ a concept similar to the principal filter is used implicitly by Bernstein when he partitions the functional dependencies (Step 2). Before describing the improved algorithm, we need the following:

Lemma 4: Let R be a relation on Ω . Let $\pi \in L(R[\Omega])$ be an atom of the relation lattice, and let K be a key of the atomic projection $R[\Omega_\pi]$. Then,

$$R[\Omega] = R[(\Omega - \Omega_\pi)K] \times R[\Omega_\pi].$$

Proof: $K \subseteq \Omega_\pi$ and $K \rightarrow \Omega_\pi$. ■

The algorithm for determining the lossless-join decomposition into BCNF is simply to construct a sequence of decompositions $D_i = (R_1, \dots, R_m)$ of R , each with lossless join: Initially, let D_0 consist of R alone. If $T[\Omega]$ is a relation in D_i , and $T[\Omega]$ is not in BCNF, let π be an atom of $L(T[\Omega])$ for which the principal filter $[\pi]$ is abnormal. Let K

Table II—Relation $R[MSPCNY]$

	Model Number	Serial Number	Price	Color	Name	Year
1	1234	342	13.25	blue	pot	1974
2	1234	347	13.25	red	pot	1974
3	1234	410	14.23	red	pot	1975
4	1465	347	9.45	black	pan	1974
5	1465	390	9.82	black	pan	1976
6	1465	392	9.82	red	pan	1976
7	1465	401	9.82	red	pan	1976
8	1465	409	9.82	blue	pan	1976
9	1623	311	22.34	blue	kettle	1973
10	1623	390	30.21	blue	kettle	1976
11	1623	410	28.55	black	kettle	1975
12	1623	423	28.55	black	kettle	1975
13	1623	428	28.55	blue	kettle	1975
14	1654	435	28.55	red	kettle	1975

be a key of the atomic projection $T[\Omega_\pi]$. Now replace $T[\Omega]$ in D_i by $T[\Omega - \Omega_\pi]K$ and $T[\Omega_\pi]$ to obtain D_{i+1} . Continue the process until all the relations in the decomposition D_k are in BCNF.

Example 3: Let us consider the relation $R[MSPCNY]$ from Ref. 23, where M = model number, S = serial number, P = price, C = color, N = name, and Y = year. The tuples of the relation $R[MSPCNY]$ are shown in Table II.

The Hasse diagram of the relation lattice $L(R[\Omega])$ is illustrated in Fig. 2, where

$$\pi_1 = \{\overline{1, 8, 9, 10, 13}; \overline{2, 3, 6, 7, 14}; \overline{4, 5, 11, 12}\},$$

$$\pi_2 = \{\overline{1, 2, 3}; \overline{4, 5, 6, 7, 8}; \overline{9, 10, 11, 12, 13, 14}\},$$

$$\pi_3 = \{\overline{1, 2, 4}; \overline{3, 11, 12, 13, 14}; \overline{5, 6, 7, 8, 10}; \overline{9}\},$$

$$\pi_4 = \{\overline{1, 2, 3}; \overline{4, 5, 6, 7, 8}; \overline{9, 10, 11, 12, 13}; \overline{14}\},$$

$$\pi_5 = \{\overline{1, 2}; \overline{3}; \overline{4}; \overline{5, 6, 7, 8}; \overline{9}; \overline{10}; \overline{11, 12, 13, 14}\},$$

$$\pi_6 = \{\overline{1}; \overline{2, 3}; \overline{4, 5}; \overline{6, 7}; \overline{8}; \overline{9, 10, 13}; \overline{11, 12}; \overline{14}\},$$

$$\pi_7 = \{\overline{1, 2}; \overline{3}; \overline{4}; \overline{5, 6, 7, 8}; \overline{9}; \overline{10}; \overline{11, 12, 13}; \overline{14}\},$$

$$\pi_8 = \{\overline{1}; \overline{2, 4}; \overline{3, 11}; \overline{5, 10}; \overline{6}; \overline{7}; \overline{8}; \overline{9}; \overline{12}; \overline{13}; \overline{14}\},$$

$$\pi_9 = \{\overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6, 7}; \overline{8}; \overline{9}; \overline{10}; \overline{11, 12}; \overline{13}; \overline{14}\},$$

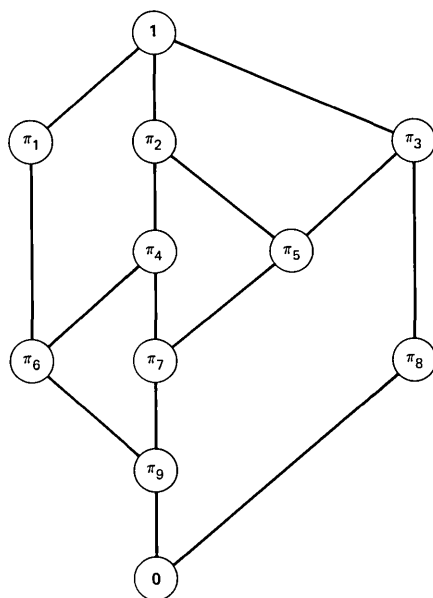


Fig. 2—Relation lattice $L(R[MSPCNY])$.

and $\theta(C) = \pi_1, \theta(N) = \pi_2, \theta(Y) = \pi_3, \theta(M) = \pi_4, \theta(P) = \pi_5, \theta(S) = \pi_8$. For $X \subseteq \Omega$, $\theta(X)$ can be obtained easily by carrying out the meet operations on the attributes in X .

The principal ideal functions of $R[MSPCNY]$ are

$$f_C(M, S, P, C, N, Y) = C + MS + NS + PS,$$

$$f_N(M, S, P, C, N, Y) = N + M + P + CY + CS,$$

$$f_P(M, S, P, C, N, Y) = P + MY + CS + MS + NS,$$

$$f_M(M, S, P, C, N, Y) = M + CN + CP + CY + NS + PS + CS,$$

$$f_Y(M, S, P, C, N, Y) = Y + P + S,$$

$$f_S(M, S, P, C, N, Y) = S,$$

and the key Boolean function is

$$\begin{aligned} F(M, S, P, C, N, Y) &= (C + MS + NS + PS) \cdot (N + M + P + CY \\ &\quad + CS) \\ &\quad \cdot (P + MY + CS + MS + NS) \\ &\quad \cdot (M + CN + CD + CY + NS + PS + CS) \\ &\quad \cdot (Y + P + S) \cdot S \\ &= CS + MS + NS + PS. \end{aligned}$$

The keys of $R[MSPCNY]$ are $\{CS, MS, NS, PS\}$, and Y is the only nonprime attribute.

Initially, let $D_0 = \{R[MSPCNY]\}$. Since both atomic filters $[\pi_3]$ and $[\pi_9]$ are abnormal, we arbitrarily choose π_9 , and let $\Sigma = \Omega_{\pi_9} = MPCNY$. The relation lattice of $R[\Sigma]$ is isomorphic to $[\pi_9]$. The principal ideal functions of $R[\Sigma]$ are

$$g_C(M, P, C, N, Y) = f_C(M, \mathbf{0}, P, C, N, Y) = C,$$

$$g_N(M, P, C, N, Y) = f_N(M, \mathbf{0}, P, C, N, Y) = N + M + P + CY,$$

$$g_P(M, P, C, N, Y) = f_P(M, \mathbf{0}, P, C, N, Y) = P + MY,$$

$$g_M(M, P, C, N, Y) = f_M(M, \mathbf{0}, P, C, N, Y) = M + CN + CP + CY,$$

$$g_Y(M, P, C, N, Y) = f_Y(M, \mathbf{0}, P, C, N, Y) = Y + P,$$

and the key Boolean function is

$$\begin{aligned} G(M, P, C, N, Y) &= C \cdot (N + M + P + CY) \cdot (P + MY) \\ &\quad \cdot (M + CN + CP + CY) \cdot (Y + P) \\ &= CP + CMY. \end{aligned}$$

We choose the key CP and replace $R[MSPCNY]$ in D_0 by $R[(\Omega - \Sigma)K] = R[SPC]$ and $R[\Sigma] = R[MPCNY]$ to obtain $D_1 = \{R[SPC], R[MPCNY]\}$. The relation $R[SPC]$ and its lattice are shown in Table III and Fig. 3, respectively.

The relation $R[SPC]$ is in BCNF, but the relation $R[MPCNY]$ is not. The relation lattice of $R[MPCNY]$ is isomorphic to the filter $[\pi_9]$. We will not duplicate the figure. Both “atoms” π_6 and π_7 of $[\pi_9]$ are abnormal. We choose the filter $[\pi_7]$. The principal ideal functions of $R[\Sigma_{\pi_7}] = R[MPNY]$ are

$$\begin{aligned} h_M(M, P, N, Y) &= g_M(M, P, \mathbf{0}, N, Y) = M, \\ h_N(M, P, N, Y) &= g_N(M, P, \mathbf{0}, N, Y) = N + M + P, \\ h_P(M, P, N, Y) &= g_P(M, P, \mathbf{0}, N, Y) = P + NY, \\ h_Y(M, P, N, Y) &= g_Y(M, P, \mathbf{0}, N, Y) = Y + P, \end{aligned}$$

and the key Boolean function of $R[MPNY]$ is given by

$$\begin{aligned} H(M, P, N, Y) &= M \cdot (N + M + P) \cdot (P + MY) \cdot (Y + P) \\ &= MP + MY. \end{aligned}$$

We choose the key $K' = MP$ and replace $R[MPCNY]$ in D_1 by $R[(\Sigma - \Sigma_{\pi_7})K'] = R[MPC]$ and $R[MPNY]$ to obtain $D_2 = \{R[SPC], R[MPC], R[MPNY]\}$. The relation $R[MPC]$ and its relation lattice are illustrated in Table IV and Fig. 4, respectively.

Now we have to decompose the relation $R[MPNY]$ in D_2 . The relation lattice of $R[MPNY]$ is isomorphic to $[\pi_7]$ of $L(R[MSPCNY])$. We choose the abnormal filter that is isomorphic to $[\pi_5]$. Since $\Sigma_{\pi_5} = PNY$ and the only key is P , we can replace $R[MPNY]$ in D_2 by $R[MP]$ and $R[PNY]$ to obtain $D_3 = \{R[SPC], R[MPC], R[MP], R[PNY]\}$. All the relations in D_3 are in BCNF. The relations $R[MP]$, $R[PNY]$ and

Table III—Relation $R[SPC]$

	Serial Number	Price	Color
1	342	13.25	blue
2	347	13.25	red
3	410	14.23	red
4	347	9.45	black
5	390	9.82	black
6	392	9.82	red
7	401	9.82	red
8	409	9.82	blue
9	311	22.34	blue
10	390	30.21	blue
11	410	28.55	black
12	423	28.55	black
13	428	28.55	blue
14	435	28.55	red

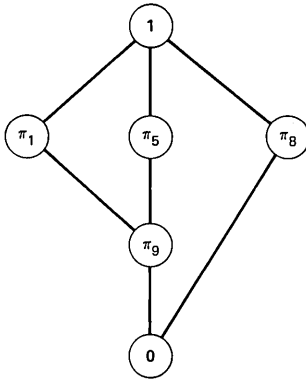


Fig. 3—Relation lattice $L(R[SPC])$.

Table IV—Relation $R[MPC]$

	Model Number	Price	Color
1	1234	13.25	blue
2	1234	13.25	red
3	1234	14.23	red
4	1465	9.45	black
5	1465	9.82	black
(6, 7)	1465	9.82	red
8	1465	9.82	blue
9	1623	22.34	blue
10	1623	30.21	blue
(11, 12)	1623	28.55	black
13	1623	28.55	blue
14	1654	28.55	red

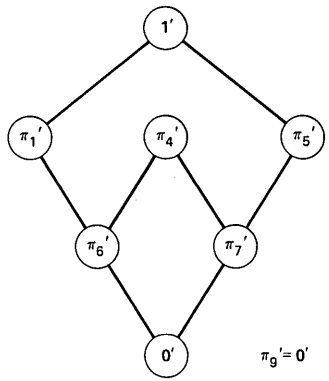


Fig. 4—Relation lattice $L(R[MPC])$.

their respective lattices are shown in Tables V and VI, and Figs. 5 and 6. \square

VI. MULTIVALUED DEPENDENCIES

Multivalued dependency (MVD) proposed by Fagin⁷ and Zaniolo⁸ is the necessary and sufficient condition for a (binary) lossless-join decomposition. A similar concept, called hierarchical dependency, was defined by Delobel.²⁴ A bit later, the concept of multivalued dependency was generalized to join dependency by Rissanen.^{10,11} A set of “axioms” or inference rules for multivalued dependencies was given by Beeri, Fagin, and Howard.²⁵ We know from our previous discussion that functional dependency is equivalent to partial ordering in the partition lattice. In this section we show that multivalued dependency

Table V—Relation $R[MP]$

	Model Number	Price
(1, 2)	1234	13.25
3	1234	14.23
4	1465	9.45
(5, 6, 7, 8)	1465	9.82
9	1623	22.34
10	1623	30.21
(11, 12, 13)	1623	28.55
14	1654	28.55

Table VI—Relation $R[PNY]$

	Price	Name	Year
(1, 2)	13.25	pot	1974
3	14.23	pot	1975
4	9.45	pan	1974
(5, 6, 7, 8)	9.82	pan	1976
9	22.34	kettle	1973
10	30.21	kettle	1976
(11, 12, 13, 14)	28.55	kettle	1975

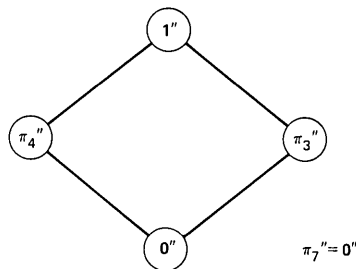


Fig. 5—Relation lattice $L(R[MP])$.

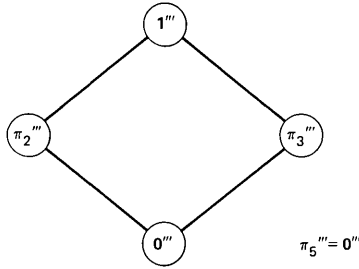


Fig. 6—Relation lattice $L(R[PNY])$.

is equivalent to a lattice equation. First, however, we state the definition of MVD and show that MVD guarantees information-lossless join decomposition.

Definition 5: Let R be a relation on the set of attributes $\Omega = XYZ$, where X , Y , and Z are disjoint subsets of Ω . We say there is a multivalued dependency $X \twoheadrightarrow Y$ if

$$R_{xz}[Y] = R_x[Y], \quad \forall(x) \in R[X], \quad (z) \in R[Z]. \quad \blacksquare$$

Lemma 5: Let R be a relation on $\Omega = XYZ$, where X , Y , and Z are disjoint subsets. Then,

$$R[XYZ] = R[XY] \mid \times \mid R[XZ]$$

iff

$$R_x[YZ] = |R_x[Y]| \cdot |R_x[Z]|, \quad \forall(x) \in R[X].$$

Proof: (Necessity) $R[XYZ] = R[XY] \mid \times \mid R[XZ]$ implies

$$R_x[YZ] = R_x[Y] \times R_x[Z], \quad \forall(x) \in R[X].$$

Hence,

$$|R_x[YZ]| = |R_x[Y]| \cdot |R_x[Z]|.$$

(Sufficiency) It is easy to verify that

$$R_x[YZ] \subseteq R_x[Y] \times R_x[Z], \quad \forall(x) \in R[X].$$

The given cardinal identity assures that

$$R_x[YZ] = R_x[Y] \times R_x[Z], \quad \forall(x) \in R[X]. \quad \blacksquare$$

Theorem 5: Let R be a relation on the set of attributes $\Omega = XYZ$, where X , Y , and Z are disjoint subsets.* Then,

$$R[XYZ] = R[XY] \mid \times \mid R[XZ] \text{ iff } X \twoheadrightarrow Y.$$

* For convenience, we assume X , Y , and Z to be disjoint. It will later become clear that this assumption is not necessary.

Proof: (Necessity) From Lemma 5, it is sufficient to show that

$$|R_x[YZ]| = |R_x[Y]| \cdot |R_x[Z]|, \quad \forall(x) \in R(X)$$

iff

$$R_{xz}[Y] = R_x[Y], \quad \forall(x) \in R(X), \quad (z) \in R(Z).$$

Since

$$R[XYZ] = R[XY] \mid \times \mid R[XZ]$$

implies

$$R_{xz}[Y] \times (x, z) = R_x[Y] \times (x, z), \quad \forall(x) \in R[X], \quad (z) \in R[Z].$$

Hence,

$$R_{xz}[Y] = R_x[Y].$$

(Sufficiency) For every $(x) \in R[X]$, we have

$$\begin{aligned} (x) \times R_x[Z] &= \{(x, z_i) \mid (x, z_i) \in R[XZ]\} \\ &= \{(x, z_i) \mid (x, z_i) \in R[XZ]\} \\ &= (x) \times R_x[Z] \times R_x[Y]. \end{aligned}$$

Since $|x| = 1$, it follows that

$$|R_x[YZ]| = |R_x[Y]| \cdot |R_x[Z]|, \quad \forall(x) \in R[X]. \quad \blacksquare$$

We need the commutative property of the product of two equivalence relations (partitions) to establish the lattice equation of multivalued dependency. The product of two equivalence relations may not be an equivalence relation; if it is an equivalence relation then the product must be commutative and vice versa.

Definition 6: Two binary relations ρ and ρ' and S are *permutable* (commute) if and only if $\rho \circ \rho' = \rho' \circ \rho$. This means that if $a \rho x \rho' b$ for some $x \in S$, then $a \rho' y \rho b$ for some $y \in S$, and conversely.¹⁸ \blacksquare

Lemma 6: Let ρ and ρ' be equivalence relations (partitions) on S . Then the following are equivalent:

1. $\rho \circ \rho' = \rho \circ \rho$
2. $\rho \circ \rho' = \rho \oplus \rho'$
3. $\rho \circ \rho'$ is an equivalence relation
4. $\rho \circ \rho'$ is symmetric.

Proof: The proof of Lemma 6 is given in Ref. 21. \blacksquare

Lemma 7: Let R be a relation on the set of attributes Ω and let $X, Y, Z \subseteq \Omega$. Then,

$$\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY)$$

iff

$$\theta(X) \subseteq \theta(XY) \circ \theta(XZ).$$

Proof: (Necessity) Trivial.

(Sufficiency) Suppose $t_1\theta(XY) \circ \theta(XZ)t_2$. Then there exists $t_3 \in R[\Omega]$ such that

$$t_1\theta(XT)t_3\theta(XZ)t_2,$$

which implies

$$t_1\theta(X)t_3\theta(X)t_2.$$

Therefore,

$$t_1\theta(X)t_2.$$

Hence,

$$\theta(XY) \circ \theta(XZ) \subseteq \theta(X).$$

It follows that

$$\theta(X) = \theta(XY) \circ \theta(XZ).$$

From Lemma 6, we have

$$\theta(X) = \theta(XY) \oplus \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY).$$

Since

$$\theta(XY) \leq \theta(XY) + \theta(XZ) \quad \text{and} \quad \theta(XZ) \leq \theta(XY) + \theta(XZ),$$

by the definition of the join operation \oplus in $\prod(R[\Omega])$, we must have

$$\theta(X) = \theta(XY) \oplus \theta(XZ) \leq \theta(XY) + \theta(XZ).$$

But,

$$\theta(XY) + \theta(XZ) \leq \theta(X) + \theta(X) = \theta(X),$$

so it follows that

$$\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY). \quad \blacksquare$$

The following theorem shows that the multivalued dependency can be formulated as a lattice equation.

Theorem 6: Let R be a relation on the set of attributes $\Omega = XYZ$, where X , Y , and Z are disjoint subsets. Then, $R[XYZ] = R[XY] \mid \times \mid R[XZ]$ iff

$$\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY).$$

Proof: (Necessity) Since $R[XYZ] = R[XY] \mid \times \mid R[XZ]$ implies

$$R_x[YZ] = R_x[Y] \times R_x[Z], \quad \forall (x) \in R[X],$$

there is a one-to-one and onto mapping $\phi_x: R_x[YZ] \rightarrow R_x[Y] \times R_x[Z]$, which takes every tuple $(y, z) \in R_x[YZ]$ into $\phi_x((y, z)) = ((y), (z)) \in R_x[Y] \times R_x[Z]$, $\forall (x) \in R[X]$. Suppose $t_1, t_2 \in R[XYZ]$ and $t_1\theta[X]t_2$, and assume $t_1 = (x, y_1, z_1)$ and $t_2 = (x, y_2, z_2)$. As

$$(y_1, z_1), (y_2, z_2) \in R_2[YZ] = R_x[Y] \times R_x[Z],$$

we have

$$(y_1), (y_2) \in R_x[Y] \quad \text{and} \quad (z_1), (z_2) \in R_x[Z].$$

Since ϕ_x is an onto mapping, there must exist two tuples $t_3 = (x, y_1, z_2)$, and $t_4 = (x, y_2, z_1) \in R[XYZ]$. Hence,

$$t_1\theta(XY)t_3\theta(XZ)t_2,$$

which means

$$t_1\theta(XY) \circ \theta(XZ)t_2.$$

It follows that

$$\theta(X) \subseteq \theta(XY) \circ \theta(XZ).$$

From Lemma 7, we have

$$\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY).$$

(Sufficiency) We know $R[XYZ] \subseteq R[XY] \mid \times \mid R[XZ]$. Suppose $t = (x, y, z) \in R[XY] \mid \times \mid R[XZ]$. Then there exist $t_1 = (x, y, z')$, $t_2 = (x, y', z) \in R[XYZ]$. Thus,

$$t_1\theta(X)t_2,$$

which implies

$$t_1\theta(XY) \circ \theta(XZ)t_2.$$

There must exist $t_3 \in R[XYZ]$ such that

$$t_1\theta(XY)t_3\theta(XZ)t_2.$$

Therefore,

$$t_3 = (x, y, z) = t \in R[XYZ],$$

and thus

$$R[XY] \mid \times \mid R[XZ] \subseteq R[XYZ].$$

Hence,

$$R[XYZ] = R[XY] \mid \times \mid R[XZ]. \quad \square$$

It should be noted that in the above proof we use the fact that $\Omega = XYZ$ and $\theta(\Omega) = \theta(XYZ) = \mathbf{0}$, i.e., there are no duplicated tuples in

$R[\Omega]$. The inference rules of MVD are given and proved in Appendix B.

Example 4: Consider the relation $R[ECSY]$ of Example 1. We have the MVD: $E \twoheadrightarrow SY$, where $\theta(E) = \pi_1 = \{1, 2; 3, 4, 5, 6; 7, 8\}$, $\theta(EC) = \pi_2 = \{1, 2; 3, 4; 5, 6; 7, 8\}$, $\theta(ESY) = \pi_5 = \{1; 2; 3, 5; 4, 6; 7; 8\}$. It is easy to verify that

$$\pi_1 = \pi_2 + \pi_5 = \pi_2 \circ \pi_5 = \pi_5 \circ \pi_2. \quad \blacksquare$$

It is known that if R is a relation on $\Omega = XYZ$, and $X \twoheadrightarrow Y$ then $X \twoheadrightarrow Z$. The symmetricity of the MVD can easily be seen in the lattice equation of Lemma 7.

If $XYZ \subset \Omega$ and $\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY)$ holds, then $X \twoheadrightarrow Y|Z$ is called an embedded multivalued dependency (EMVD)⁷; this is simply a multivalued dependency in the projection $R[XYZ]$ of $R[\Omega]$.

Theorem 6 clearly indicates that the MVD is actually a condition pertaining to data *independency* rather than data dependency. For this reason, we introduce the notion of decomposition of two sets of attributes in a relation as follows.

Definition 7: Let R be a relation on the set of attributes Ω . The two sets of attributes $\Omega_1, \Omega_2 \subseteq \Omega$ are decomposable in R if

$$\theta(\Omega_1 + \Omega_2) = \theta(\Omega_1) + \theta(\Omega_2) = \theta(\Omega_1) \circ \theta(\Omega_2) = \theta(\Omega_2) \circ \theta(\Omega_1). \quad \blacksquare$$

It is easy to see that Ω_1 and Ω_2 are decomposable in Ω iff $\Omega_1 + \Omega_2 \rightarrow \Omega_1 - \Omega_2 | \Omega_2 - \Omega_1$ is an EMVD in R . Furthermore, if $\Omega_1 \Omega_2 = \Omega$ then $\Omega_1 + \Omega_2 \twoheadrightarrow \Omega_1 - \Omega_2$ (or $\Omega_1 + \Omega_2 \twoheadrightarrow \Omega_2 - \Omega_1$) is an MVD in R . In the latter case, (Ω_1, Ω_2) is called a *decomposition pair* by Armstrong and Delobel.²⁶

We feel that decomposition is a basic concept in the study of the structure of databases. It can be naturally generalized to the concepts of projective decomposition and mutual decomposition. Projective composition concerns the data independence of two sets of attributes on the projection of a relation. Mutual decomposition extends the concept of decomposition to more than two sets of attributes.

Let ρ be a partition on the set S , the function $\rho^* = S \rightarrow S/\rho$ maps $a \in S$ into $(a)\rho^* = a_\rho$ is called the *canonical function* of ρ . For $S = \{a, b, \dots, e\}$, we will use the notation

$$\rho^* = \begin{pmatrix} a & b & \dots & e \\ a_\rho & b_\rho & \dots & e_\rho \end{pmatrix}$$

to illustrate the canonical function ρ^* . The equivalence relation

$$\ker \rho^* = \rho^* \circ \rho^{*-1} = \{(a, b) \in S \times S | \rho^*(a) = \rho^*(b)\}.$$

is called the *kernel* of ρ^* . Notice that $\ker \rho^* = \rho$.

Let ρ and σ be partitions on $S = \rho \subseteq \sigma$; then there is a unique function f from S/ρ onto S/σ such that $(a_\rho)f = a_\sigma$. The kernel of f ,

$$\ker f = f \circ f^{-1} = \{(a_\rho, b_\rho) \in S/\rho \times S/\rho \mid a \sigma b\},$$

is an equivalence on S/ρ . It is usual to write $\ker f$ as σ/ρ , the quotient of σ and ρ . Note that $a_\rho(\sigma/\rho)b_\rho$ if and only if $a \sigma b$ and the mapping $g: (S/\rho)/(\sigma/\rho) \rightarrow S/\sigma$ defined by $((a_\rho)_{\sigma/\rho})g = a_\sigma$ is one-to-one and onto. Thus the function f defined above is in fact the canonical function of σ/ρ , i.e., $f = (\sigma/\rho)^*$. It is easy to see the diagram in Fig. 7 commutes, that is $\rho^* \circ (\sigma/\rho)^* = \sigma^*$.

Example 5: Let ρ, σ be partitions on the set $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$, such that $\rho = \{\overline{1, 2}, \overline{3, 4}, \overline{5, 6}, \overline{7, 8}\}$ and $\sigma = \{\overline{1, 2, 3, 4}, \overline{5, 6, 7, 8}\}$ with $\rho \subseteq \sigma$. Then $S/\rho = \{I, II, III, IV\}$, where $I = \overline{1, 2}, II = \overline{3, 4}, III = \overline{5, 6}, IV = \overline{7, 8}$, and $S/\sigma = \{\alpha, \beta\}$, where $\alpha = \overline{1, 2, 3, 4}, \beta = \overline{5, 6, 7, 8}$. The canonical functions of ρ and σ are

$$\rho^* = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ I & I & II & II & III & III & IV & IV \end{pmatrix}$$

and

$$\sigma^* = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \alpha & \alpha & \alpha & \alpha & \beta & \beta & \beta & \beta \end{pmatrix}.$$

It follows that

$$\sigma/\rho = \{\overline{I, II}, \overline{III, IV}\}$$

and

$$(\sigma/\rho)^* = \begin{pmatrix} I & II & III & IV \\ \alpha & \alpha & \beta & \beta \end{pmatrix}. \quad \square$$

Lemma 8: Let ρ, σ_1, σ_2 be partitions on S such that $\rho \subseteq \sigma_1, \rho \subseteq \sigma_2$, and $\sigma_1 \circ \sigma_2 = \sigma_2 \circ \sigma_1$. Then

$$(\sigma_1 \circ \sigma_2)/\rho = (\sigma_1/\rho) \circ (\sigma_2/\rho).$$

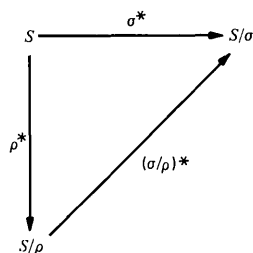


Fig. 7—Canonical function of quotient partition.

Proof: It is clear that $\sigma_1 \circ \sigma_2$ is a partition on S and $\rho \subseteq \sigma_1 \circ \sigma_2 = \sigma_2 \circ \sigma_1$. It follows from the definition of quotient partition that the lemma is true. \square

Lemma 9: Let ρ, σ_1, σ_2 be partitions on S , such that $\rho \subseteq \sigma_1, \rho \subseteq \sigma_2$. Then

$$\sigma_1 \circ \sigma_2 = \sigma_2 \circ \sigma_1$$

iff

$$(\sigma_1/\rho) \circ (\sigma_2/\rho) = (\sigma_2/\rho) \circ (\sigma_1/\rho).$$

Proof: (Necessity)

$$(\sigma_1/\rho) \circ (\sigma_2/\rho) = (\sigma_1 \circ \sigma_2)/\rho = (\sigma_2 \circ \sigma_1)/\rho = (\sigma_2/\rho) \circ (\sigma_1/\rho).$$

(Sufficiency) Suppose $a\sigma_1 \circ \sigma_2 b$. Then there is a $c \in S$ such that $a\sigma_1 c \sigma_2 b$. It follows that

$$a_\rho(\sigma_1/\rho)c_\rho(\sigma_2/\rho)b_\rho.$$

There must be a $d \in S$ such that

$$a_\rho(\sigma_2/\rho)d_\rho(\sigma_1/\rho)b_\rho.$$

Thus

$$a\sigma_2 d \sigma_1 b,$$

and

$$a\sigma_2 \circ \sigma_1 b.$$

Hence

$$\sigma_1 \circ \sigma_2 \subseteq \sigma_2 \circ \sigma_1.$$

Similarly, we have $\sigma_2 \circ \sigma_1 \subseteq \sigma_1 \circ \sigma_2$. Then $\sigma_1 \circ \sigma_2 = \sigma_2 \circ \sigma_1$. \square

Definition 8: Let R be a relation on the set of attributes Ω . For $\Omega_1, \Omega_2 \subseteq \Omega$, the *projective partition* defined by

$$\theta(\Omega_1 | \Omega_2) = \theta(\Omega_1 + \Omega_2) / \theta(\Omega_2)$$

is a partition on the set of tuples of $R[\Omega] / \theta(\Omega_2) = R[\Omega_2]$. The canonical function of $\theta(\Omega_1 | \Omega_2)$ is denoted by $\theta^*(\Omega_1 | \Omega_2) = (\theta(\Omega_1 + \Omega_2) / \theta(\Omega_2))^*$, which satisfies $\theta^*(\Omega_2) \circ \theta^*(\Omega_1 | \Omega_2) = \theta^*(\Omega_1 + \Omega_2)$. \square

Certain properties of the projective partition are demonstrated in the following theorems and their proof directly follows from the definition of projective partition.

Theorem 7: Let R be a relation on the set of attributes Ω and $\Omega_1, \dots, \Omega_n \subseteq \Omega$. Then

$$\theta^* \left(\bigcap_{k=1}^n \Omega_k \right) = \theta^*(\Omega_1) \circ \theta^*(\Omega_2 | \Omega_1) \circ \dots \circ \theta^*(\Omega_n | \bigcap_{k=1}^{n-1} \Omega_k). \quad \square$$

Theorem 8: Let R be a relation on the set of attributes Ω and X , $\Omega_1, \dots, \Omega_n \subseteq \Omega$, such that

$$\bigcup_{k=1}^n \Omega_k = \Omega.$$

Then

1. $\theta(X) = \prod_{k=1}^n \ker(\theta^*(\Omega_k) \circ \theta^*(X|\Omega_k))$
2. $\theta(\Omega_m|X) = \ker(\theta^*(\Omega_m) \circ \theta^*(X|\Omega_m)) / \prod_{k=1}^n \ker(\theta^*(\Omega_k) \circ \theta^*(X|\Omega_k))$. \square

Definition 9: Let R be a relation on the set of attributes Ω and $\Omega_1, \Omega_2, \Sigma \subseteq \Omega$. We say Ω_1 and Ω_2 are projectively decomposable on Σ if

$$\begin{aligned} \theta(\Omega_1 + \Omega_2|\Sigma) &= \theta(\Omega_1|\Sigma) + \theta(\Omega_2|\Sigma) \\ &= \theta(\Omega_1|\Sigma) \circ \theta(\Omega_2|\Sigma) \\ &= \theta(\Omega_2|\Sigma) \circ \theta(\Omega_1|\Sigma). \quad \square \end{aligned}$$

The EMVD is a special case of projective decomposition, which can be seen from the following theorem.

Theorem 9: Let R be a relation on Ω , and let $\Omega_1, \Omega_2, \Sigma \subseteq \Omega$. Then Ω_1 and Ω_2 are projectively decomposable on Σ iff

$$\begin{aligned} \theta(\Omega_1 + \Omega_2 + \Sigma) &= \theta(\Omega_1 + \Sigma) + \theta(\Omega_2 + \Sigma) \\ &= \theta(\Omega_1 + \Sigma) \circ \theta(\Omega_2 + \Sigma) = \theta(\Omega_2 + \Sigma) \circ \theta(\Omega_1 + \Sigma). \end{aligned}$$

Proof: The proof follows from Lemma 8 and 9. \square

Example 6: Consider the relation R on $\Omega = ABCDE$ in Table VII. The Hasse diagram of the relation lattice $L(R[\Omega])$ is shown in Fig. 8, where

$$\begin{aligned} \pi_1 &= \{\overline{1, 2, 3, 5, 6, 7}; \overline{4}\} = \theta(A), \\ \pi_2 &= \{\overline{1, 3, 4}; \overline{2, 5, 6, 7}\} = \theta(B), \\ \pi_3 &= \{\overline{1, 6, 7}; \overline{2, 3, 4, 5}\} = \theta(C), \\ \pi_4 &= \{\overline{1, 3}; \overline{2, 5, 6, 7}; \overline{4}\} = \theta(AB), \\ \pi_5 &= \{\overline{1, 6, 7}; \overline{2, 3, 5}; \overline{4}\} = \theta(AC), \\ \pi_6 &= \{\overline{1}; \overline{2, 5}; \overline{3, 4}; \overline{6, 7}\} = \theta(BC), \\ \pi_7 &= \{\overline{1, 6, 7}; \overline{2, 5}; \overline{3}; \overline{4}\} = \theta(D), \\ \pi_8 &= \{\overline{1, 3}; \overline{2, 6}; \overline{4}; \overline{5, 7}\} = \theta(E), \\ \pi_9 &= \{\overline{1}; \overline{2, 5}; \overline{3}; \overline{4}; \overline{6, 7}\} = \theta(ABC) = \theta(ABD). \end{aligned}$$

Let $\Omega_1 = ABD$, $\Omega_2 = ACE$, $\Sigma = ABC$. We find that

Table VII—Relation lattice $L(R[\Omega])$

	A	B	C	D	E
1	a_1	b_1	c_1	d_1	e_1
2	a_1	b_2	c_2	d_2	e_2
3	a_1	b_1	c_2	d_3	e_1
4	a_2	b_1	c_2	d_4	e_3
5	a_1	b_2	c_2	d_2	e_4
6	a_1	b_2	c_1	d_1	e_2
7	a_1	b_2	c_1	d_1	e_4

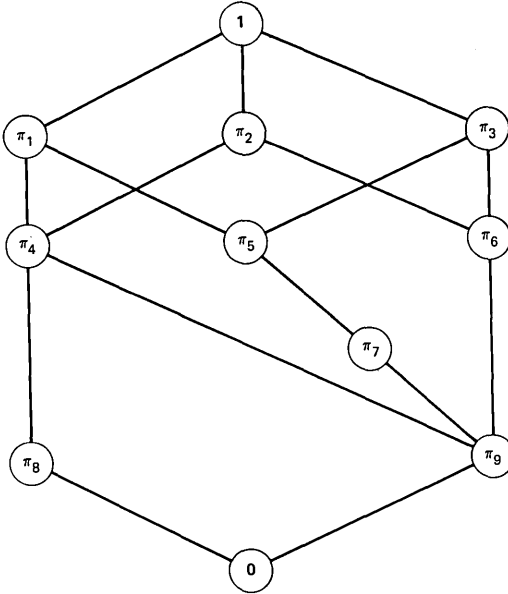


Fig. 8—Relation lattice $L(R[\Omega])$.

$$\theta(\Omega_1 + \Omega_2 | \Sigma) = \theta(A | ABC) = \theta(A) / \theta(ABC) = \{\overline{I}, \overline{II}, \overline{III}, \overline{V}; \overline{IV}\},$$

$$\theta(\Omega_1 | \Sigma) = \theta(ABD | ABC) = \theta(AB) / \theta(ABC) = \{\overline{I}, \overline{III}; \overline{II}, \overline{V}; \overline{IV}\},$$

$$\theta(\Omega_2 | \Sigma) = \theta(ACE | ABC) = \theta(AC) / \theta(ABC) = \{\overline{I}, \overline{V}; \overline{II}, \overline{III}; \overline{IV}\},$$

and

$$\theta(ABC) = \{I, II, III, IV, V\},$$

where $I = \overline{1}$, $II = \overline{2, 5}$, $III = \overline{3}$, $IV = \overline{4}$, $V = \overline{6, 7}$.

It is easy to see that ABD and ACE are projectively decomposable on ABC , i.e.,

$$\begin{aligned} \theta(A | ABC) &= \theta(ABD | ABC) + \theta(ACE | ABC) \\ &= \theta(ABD | ABC) \circ \theta(ACE | ABC) \\ &= \theta(ACE | ABC) \circ \theta(ABD | ABC). \end{aligned}$$

But the MVD: $A \twoheadrightarrow BD$ (or $A \twoheadrightarrow CE$) does not hold in $R[\Omega]$. Nevertheless, the EMVD: $A \rightarrow B | C$ does hold in $R[\Omega]$. \square

So far we have discussed the properties of decomposition of two sets of attributes. The concept of decomposition certainly can be extended to any $n > 2$ sets of attributes. We define the notion of mutual decomposition as follows:

Definition 10: Let R be a relation on the set of attributes Ω . The sets of attributes $\Omega_1, \Omega_2, \dots, \Omega_n \subseteq \Omega$ are mutually decomposable, if for any $I \subseteq N = \{1, \dots, n\}$ and $J \subseteq N - I$, the two sets of attributes $\Omega_I = \cup_{i \in I} \Omega_i$ and $\Omega_J = \cup_{j \in J} \Omega_j$ are decomposable. \square

Theorem 10: Let R be a relation on the set of attributes Ω and $\Omega_1 \dots \Omega_n = \Omega$. Suppose $\Omega_1, \dots, \Omega_n$ are mutually decomposable. Then

$$R[\Omega] = R[\Omega_1 \dots \Omega_n] = R[\Omega_1] | \times | \dots | \times | R[\Omega_n].$$

Proof: It follows from the definition of mutual decomposition that

$$R[\Omega_1 \dots \Omega_m] = R[\Omega_1 \dots \Omega_{m-1}] | \times | R[\Omega_m], \quad m = 2, \dots, n.$$

Therefore the assertion is true by induction. \square

The above theorem states that mutual decomposition implies an information-lossless join. The converse is not true in general. The necessary and sufficient condition of an information-lossless join is called join dependency, which will be discussed in the next section.

VII. JOIN DEPENDENCIES

Join dependency (JD)^{10,11,14} is a generalization of MVD. It refers to a collection $\{\Omega_1, \dots, \Omega_n\}$ of subsets of Ω such that

$$\Omega = \Omega_1 \dots \Omega_n$$

and

$$R[\Omega] = R[\Omega_1] | \times | \dots | \times | R[\Omega_n].$$

Join dependency can be considered as a “set of coordinates” of the relation. The connection between join dependencies and multivalued dependencies is given by the following lemma:

Lemma 10: Let $R[\Omega] = R[\Omega_1] | \times | \dots | \times | R[\Omega_n]$, let N_0 be a subset of $\{1, \dots, n\}$, and let $N_1 = \{1, \dots, n\} - N_0$. Then $(\Omega_{N_0}, \Omega_{N_1})$ is a decomposition pair, where

$$\Omega = \Omega_1 \dots \Omega_n, \quad \Omega_{N_0} = \bigcup_{i \in N_0} \Omega_i, \quad \text{and} \quad \Omega_{N_1} = \bigcup_{i \in N_1} \Omega_i.$$

Proof: Since

$$R \left[\bigcup_{i \in N_0} \Omega_i \right] \subseteq | \times |_{i \in N_0} R[\Omega_i],$$

and

$$R \left[\bigcup_{i \in N_1} \Omega_i \right] \subseteq \prod_{i \in N_1} R[\Omega_i],$$

it follows that

$$R[\Omega_{N_0}] \times \prod_{i \in N_1} R[\Omega_i] \subseteq \left(\prod_{i \in N_0} R[\Omega_i] \right) \times \left(\prod_{i \in N_1} R[\Omega_i] \right).$$

Since the natural join operation is commutative and associative,⁶ we have

$$R[\Omega_{N_0}] \times \prod_{i \in N_1} R[\Omega_i] \subseteq R[\Omega_1] \times \cdots \times R[\Omega_n] = R(\Omega).$$

But we know

$$R[\Omega] \subseteq R[\Omega_{N_0}] \times \prod_{i \in N_1} R[\Omega_i].$$

Hence,

$$R[\Omega] = R[\Omega_{N_0}] \times \prod_{i \in N_1} R[\Omega_i]. \quad \blacksquare$$

Let x be an X -value, and assume $Y \subseteq X$. We shall denote the Y -value in x as $x[Y]$. Let $t \in R[\Omega]$ be a tuple and let $\Omega = \Omega_1 \cdots \Omega_n$. The notation $t \triangleq (w_1, \dots, w_n)$ will be used to indicate that $t[\Omega_i] = w_i$, $\forall i \in N$, where $N = \{1, \dots, n\}$ denotes the index set.

Before we state the necessary and sufficient conditions for join dependency, we first introduce the concepts of a *set of consistent values* and an *indexed family of tuples*.

Definition 11: Let R be a relation on the set of attributes Ω , and let $\{X_i \mid i \in N\}$ be a collection of subsets of Ω . The set of values $\{x_i \mid x_i$ is an X_i -value, $i \in N\}$ is called a *set of consistent values* of $\{X_i \mid i \in N\}$ if the values of $X_i \cap X_j$ in x_i and x_j agree, i.e., if

$$x_i[X_i \cap X_j] = x_j[X_i \cap X_j], \quad \forall i, j \in N.$$

The set of tuples $\{t_i \mid i \in N\}$ of $R[\Omega]$ is called an *indexed family of tuples* with respect to $\{X_i \mid i \in N\}$ if $\{x_i \mid t_i[X_i] = x_i, i \in N\}$ is a set of consistent values. \blacksquare

Theorem 11: Let R be a relation on the set of attributes Ω , and let $\Omega = \Omega_1 \cdots \Omega_n$. Then

$$R[\Omega] = R[\Omega_1] \times \cdots \times R[\Omega_n]$$

iff for every indexed family of tuples $\{t_i \mid i \in N\}$ with respect to $\{X_i \mid i \in N\}$ there is a tuple $t \in R[\Omega]$ such that $t[\Omega_i] = t_i[\Omega_i]$, $\forall i \in N$, where $X_i = \Omega_i \cap \hat{\Omega}_i$, and $\hat{\Omega}_i = \bigcup_{j \neq i} \Omega_j$.

Proof: (Necessity) Let $\{t_i \mid i \in N\}$ be an indexed family of tuples of $R[\Omega]$ with respect to $\{X_i \mid i \in N\}$. Thus, $\{x_i \mid t_i[X_i] = x_i, i \in N\}$ is a set of consistent values. Suppose $t_i[\Omega_i] = w_i, i \in N$. We want to show that

there exists a tuple $t \triangleq (w_1, \dots, w_n) \in R[\Omega]$. We will prove this by mathematical induction. We know that $s_1 = (w_1) \in R[\Omega_1]$ and $(w_2) \in R[\Omega_2]$. Thus,

$$w_1[X_1] = x_1 \quad \text{and} \quad w_2[X_2] = x_2.$$

Since $\{x_i \mid i \in N\}$ is consistent, it follows that

$$w_1[X_1 \cap X_2] = x_1[X_1 \cap X_2] = x_2[X_1 \cap X_2] = w_2[X_1 \cap X_2].$$

It is known that

$$X_i \cap X_j = (\Omega_i \cap \hat{\Omega}_i) \cap (\Omega_j \cap \hat{\Omega}_j) = \Omega_i \cap \Omega_j, \quad i \neq j.$$

Therefore,

$$w_1[\Omega_1 \cap \Omega_2] = w_2[\Omega_1 \cap \Omega_2].$$

By the definition of natural join, we know that there exists a tuple $s_2 \triangleq (w_1, w_2) \in R[\Omega_1] \times | R[\Omega_2]$.

Suppose there is a tuple $s_{n-1} \triangleq (w_1, \dots, w_{n-1}) \in R[\Omega_1] \times | \dots \times | R[\Omega_{n-1}]$. Then

$$\begin{aligned} s_{n-1}[X_i \cap X_n] &= w_i[X_i \cap X_n] = x_i[X_i \cap X_n] \\ &= x_n[X_i \cap X_n] = w_n[X_i \cap X_n], \quad i = 1, \dots, n-1. \end{aligned}$$

Hence,

$$\begin{aligned} s_{n-1}[\Omega_n \cap \hat{\Omega}_n] &= s_{n-1}[\Omega_n \cap (\Omega_1 \cup \dots \cup \Omega_{n-1})] \\ &= s_{n-1}[(\Omega_1 \cap \Omega_n) \cup \dots \cup (\Omega_{n-1} \cap \Omega_n)] \\ &= s_{n-1}[(X_1 \cap X_n) \cup \dots \cup (X_{n-1} \cap X_n)] \\ &= w_n[(X_1 \cap X_n) \cup \dots \cup (X_{n-1} \cap X_n)] \\ &= w_n[\Omega_n \cap \hat{\Omega}_n]. \end{aligned}$$

It follows that there exists a tuple t such that

$$t = s_n \triangleq (w_1, \dots, w_n) \in R[\Omega_1] \times | \dots \times | R[\Omega_n] = R[\Omega].$$

(Sufficiency) We know that

$$R[\Omega] \subseteq R[\Omega_1] \times | \dots \times | R[\Omega_n].$$

For any $t \triangleq (w_1, \dots, w_n) \in R[\Omega_1] \times | \dots \times | R[\Omega_n]$, there exists an indexed family of tuples $\{t_i \mid t_i[\Omega_i] = w_i, i \in N\}$ of $R[\Omega]$ with respect to $\{X_i \mid i \in N\}$ that has a set of consistent values $\{x_i \mid w_i[X_i] = x_i, i \in N\}$. It follows that $t \triangleq (w_1, \dots, w_n) \in R[\Omega]$. Hence,

$$R[\Omega] = R[\Omega_1] \times | \dots \times | R[\Omega_n]. \quad \square$$

The necessary and sufficient conditions for JD given in the above

theorem are similar to the notion of template dependency introduced by Sadri and Ullman.²⁷ The following condition can be considered as an extension of the binary natural join operation.

Corollary 3: Let R be a relation on the set of attributes Ω , and $\Omega = \Omega_1 \dots \Omega_n$. Then,

$$R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$$

iff

$$R_{x_1, \dots, x_n}[\Omega] = R_{x_1}[\Omega_1] \mid \times \mid \dots \mid \times \mid R_{x_n}[X_n]$$

for every set of consistent values $\{x_i \mid i \in N\}$ of $\{X_i \mid i \in N\}$, where $X_i = \Omega_i \cup \hat{\Omega}_i$, $\forall i \in N$, and $R_{x_1, \dots, x_n}[\Omega] = \{t \mid t \in R[\Omega], t[X_i] = x_i, \forall i \in N\}$.

Proof: The proof follows from Theorem 11. ■

Clearly, for any $t \in R[\Omega] = R[\Omega_1 \dots \Omega_n]$, the set of values $\{x_i \mid t[X_i] = x_i, X_i = \Omega_i \cap \hat{\Omega}_i, i \in N\}$ is always consistent. The converse is not necessarily true. Suppose for any set of consistent values $\{x_i \mid x_i \text{ is an } X_i\text{-value}, i \in N\}$ there is a tuple $t \in R[\Omega]$ such that $t[X_i] = x_i, \forall i \in N$; in this case we say $\{\Omega_i \mid i \in N\}$ is *complete*.

Corollary 4: Let R be a relation on the set of attributes Ω , and $\Omega = \Omega_1 \dots \Omega_n$. Then $\{\Omega_i \mid i \in N\}$ is complete iff

$$R[X_1 \dots X_n] = R[X_1] \mid \times \mid \dots \mid \times \mid R[X_n],$$

where $X_i = \Omega_i \cap \hat{\Omega}_i, i \in N$.

Proof: The proof follows directly from Theorem 11. ■

The necessary and sufficient conditions for JD may be stated in a different form, as follows:

Theorem 12: Let R be a relation on the set of attributes Ω , and $\Omega = \Omega_1 \dots \Omega_n$. Then

$$R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$$

iff

1. $\{\Omega_i, \hat{\Omega}_i\}$ is a decomposition pair, $i \in N$,
2. $\{\Omega_i \mid i \in N\}$ is complete, i.e.,

$$R[X_1 \dots X_n] = R[X_1] \mid \times \mid \dots \mid \times \mid R[X_n], \quad X_i = \Omega_i \cap \hat{\Omega}_i, \quad i \in N.$$

Proof: (Necessity) Condition 1 follows from Lemma 10. Condition 2 is a consequence of Theorem 11.

(Sufficiency) We know that

$$R[\Omega] \subseteq R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n].$$

Suppose $t \triangleq (w_1, \dots, w_n) \in R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$. Then there is an indexed family of tuples $\{t_i \mid t_i[\Omega_i] = w_i, i \in N\}$ of $R[\Omega]$ with respect

to $\{X_i | i \in N\}$ and the set of consistent values $\{x_i | w_i[X_i] = x_i, X_i = \Omega_i \cap \hat{\Omega}_i, i \in N\}$. We will prove by mathematical induction that $t \triangleq (w_1, \dots, w_n) \in R[\Omega]$.

Since $\{\Omega_i | i \in N\}$ is complete, there exists a tuple $s \triangleq (y_1, \dots, y_n) \in R[\Omega]$ such that

$$s[X_i] = x_i, \quad \forall i \in N.$$

We know that

$$t_1[\Omega_1 \cap \hat{\Omega}_1] = t_1[X_1] = w_1[X_1] = x_1 = s[X_1] = s[\Omega_1 \cap \hat{\Omega}_1],$$

which means

$$t_1\theta(\Omega_1 \cap \hat{\Omega}_1)s.$$

Since $(\Omega_1, \hat{\Omega}_1)$ is a decomposition pair, there exists a tuple $s_1 \in R[\Omega]$ such that

$$t_1\theta(\Omega_1)s_1\theta(\hat{\Omega}_1)s.$$

Hence,

$$s_1 \triangleq (w_1, y_2, \dots, y_n) \in R[\Omega].$$

Suppose there is a tuple $s_{n-1} \triangleq (w_1, \dots, w_{n-1}, y_n) \in R[\Omega]$. It follows that

$$t_n[\Omega_n \cap \hat{\Omega}_n] = t_n[X_n] = w_n[X_n] = x_n = s_{n-1}[X_n] = s_{n-1}[\Omega_n \cap \hat{\Omega}_n].$$

Thus there is a tuple $s_n \in R[\Omega]$ such that

$$t_n\theta(\Omega_n)s_n\theta(\hat{\Omega}_n)s_{n-1}.$$

Hence

$$t = s_n \triangleq (w_1, \dots, w_n) \in R[\Omega]. \quad \square$$

It is known that a special class of JD, called *acyclic join dependency*, has many desirable properties; this class makes operations like updates and joins especially easy.^{15, 28} A collection of subsets $\{\Omega_i | i \in N\}$ of the set of attributes Ω is called *acyclic* if all the attributes can be deleted by repeatedly applying the following two operations:^{15, 28}

1. Delete from some Ω_i an attribute A that appears in no other Ω_j
2. Delete one Ω_i if there is an $\Omega_j, i \neq j$, such that $\Omega_i \subseteq \Omega_j$.

A reduction $\{Y_j | j \in J \subseteq N, \text{ and } \forall i \in N - J \exists j \in J \text{ such that } Y_i \subseteq Y_j\}$ is obtained by removing from $\{Y_i | i \in N\}$ each Y_i that is contained in another Y_j .

Definition 12: Let $\mathbf{S} = \{\Omega_i | i \in N\}$ be a collection of subsets of Ω . The *core* of \mathbf{S} , denoted by $\hat{\mathbf{S}}$, is defined as follows:

1. $\hat{\mathbf{S}} = \emptyset$, for $|\mathbf{S}| = N = 1$
2. $\hat{\mathbf{S}}$ is the reduction of $\{\Omega_i \cap \hat{\Omega}_i | i \in N\}$, for $|\mathbf{S}| = N > 1$. ■

There are many different but equivalent conditions that characterize a collection of subsets as acyclic.¹⁵ We will use the following one:

Lemma 11: A collection $\mathbf{S} = \{\Omega_i \mid i \in N\}$ of subsets of Ω is acyclic iff its core $\hat{\mathbf{S}}$ is acyclic.

Proof: $\hat{\mathbf{S}}$ can be obtained from \mathbf{S} by performing the operations 1 and 2 defined above. It follows that if \mathbf{S} is acyclic then $\hat{\mathbf{S}}$ is acyclic and vice versa. ■

Corollary 5: Let $\mathbf{S} = \{\Omega_i \mid i \in N\}$ be an acyclic collection of subsets of Ω . Then $|\mathbf{S}| > |\hat{\mathbf{S}}|$.

Proof: For $|\mathbf{S}| = 1$, $|\hat{\mathbf{S}}| = |\emptyset| = 0$. For $|\mathbf{S}| \geq 2$, we know that $|\mathbf{S}| \geq |\hat{\mathbf{S}}|$. Suppose $|\mathbf{S}| = |\hat{\mathbf{S}}|$. Then any attribute A in $\hat{\mathbf{S}}$ must be contained in at least two distinct subsets of $\hat{\mathbf{S}}$. Let $A \in \Omega_i \cap \hat{\Omega}_i$. Then $A \in \Omega_i$ and $A \in \hat{\Omega}_i$. There is a $j \neq i$ such that $A \in \Omega_j$. Since $\Omega_i \subseteq \hat{\Omega}_j = \cup_{k \neq j} \Omega_k$ it follows that

$$A \in \Omega_j \cap \hat{\Omega}_j \in \hat{\mathbf{S}}.$$

Since $|\hat{\mathbf{S}}| = |\mathbf{S}| \geq 2$, $\hat{\mathbf{S}}$ is not empty. Now, neither operation 1 nor 2 can be applied to reduce $\hat{\mathbf{S}}$. From Lemma 11 we know this contradicts the assumption that \mathbf{S} is acyclic. Thus, $|\mathbf{S}| > |\hat{\mathbf{S}}|$. ■

A JD $R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$, $\Omega = \Omega_1 \dots \Omega_n$, is an acyclic join dependency if $\{\Omega_i \mid i \in N\}$ is acyclic. A recursive condition for acyclic join dependency is as follows:

Corollary 6: Let R be a relation on the set of attributes $\Omega = \Omega_1 \dots \Omega_n$. Then

$$R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$$

is an acyclic join dependency iff

1. $(\Omega_i, \hat{\Omega}_i)$ is a decomposition pair of $R[\Omega]$, $i = 1, \dots, n$,
2. $R[X_1 \dots X_m] = R[X_1] \mid \times \mid \dots \mid \times \mid R[X_m]$ is an acyclic join dependency over the set $X_1 \dots X_m \subseteq \Omega$, where $\{X_i \mid i = 1, \dots, m\}$ is the core of $\{\Omega_i \mid i \in N\}$.

Proof: The join dependency of a collection of sets and the join dependency of its reduction are equivalent.²⁵ The proof easily follows from Theorem 12 and Lemma 11. ■

The above corollary simply states that acyclic join dependency is equivalent to a set of MVDs and EMVDs, i.e., a set of simultaneous lattice equations that can be derived recursively. It has been shown by hypergraph theory that an acyclic join dependency is equivalent to a set of MVDs.^{15,16} That is, the converse of Lemma 10 is true for acyclic join dependency; we will prove that the converse of Lemma 10 is a consequence of Corollary 6.

Theorem 10: Let R be a relation on the set of attributes $\Omega = \Omega_1 \dots \Omega_n$ such that $\{\Omega_i \mid i \in N\}$ is acyclic. Suppose for any $N_0 \subseteq N = \{1, \dots, n\}$, $N_1 = N - N_0$, and

$$R[\Omega] = R[\Omega_{N_0}] \mid \times \mid R[\Omega_{N_1}].$$

Then

$$R[\Omega] = R[\Omega_1] \mid \times \mid \dots \mid \times \mid R[\Omega_n]$$

is an acyclic join dependency.

Proof: This theorem will be proved by mathematical induction on n . For the smallest nontrivial case $n = 3$, let the core set $\{X_i \mid i = 1, \dots, m\}$ of $\{\Omega_i \mid i = 1, 2, 3\}$ be the reduction of $\{Y_i = \Omega_i \cap \hat{\Omega}_i \mid i = 1, 2, 3\}$. First we want to show that

$$R[X_1 \dots X_m] = R[X_1] \mid \times \mid \dots \mid \times \mid R[X_m].$$

We know $m < 3$ from Corollary 5. There is nothing to be proved if $m < 2$. For $m = 2$, without loss of generality, let $X_1 = Y_1$, $X_2 = Y_2$, and $Y_3 \subseteq Y_2 = X_2$. Then

$$\begin{aligned} X_1 \cap X_2 &= Y_1 \cap Y_2 = Y_1 \cap Y_2 Y_3 = (Y_1 \cap Y_2) \cup (Y_1 \cap Y_3) \\ &= (\Omega_1 \cap \Omega_2) \cup (\Omega_1 \cap \Omega_3) = \Omega_1 \cap \Omega_2 \Omega_3. \end{aligned}$$

Since $(\Omega_1, \Omega_2 \Omega_3)$ is a decomposition pair,

$$\theta(X_1 \cap X_2) = \theta(\Omega_1 \cap \Omega_2 \Omega_3) \subseteq \theta(\Omega_1) \circ \theta(\Omega_2 \Omega_3).$$

Also we have

$$\Omega_1 \supseteq Y_1 = X_1,$$

and

$$\Omega_2 \Omega_3 \supseteq Y_2 Y_3 = Y_2 = X_2.$$

Thus

$$\theta(\Omega_1) \subseteq \theta(X_1),$$

$$\theta(\Omega_2 \Omega_3) \subseteq \theta(X_2),$$

and

$$\theta(\Omega_1) \circ \theta(\Omega_2 \Omega_3) \subseteq \theta(X_1) \circ \theta(X_2).$$

It follows that

$$\theta(X_1 \cap X_2) \subseteq \theta(X_1) \circ \theta(X_2).$$

Hence

$$R[X_1 X_2] = R[X_1] \mid \times \mid R[X_2].$$

It follows from Corollary 6 that

$$R[\Omega] = R[\Omega_1] \mid \times \mid R[\Omega_2] \mid \times \mid R[\Omega_3].$$

Suppose the theorem is true for all $k < n$. Let the core set $\{X_i \mid i = 1,$

$\dots, m\}$ of $\{\Omega_i | i \in N\}$ be the reduction of $\{Y_i = \Omega_i \cap \hat{\Omega}_i | i \in N\}$. We know $m < n$ and for any $M_0 \subseteq M = \{1, \dots, m\}$ and $M_1 = M - M_0$ there is an $N_0 \subseteq N$ and $N_1 = N - N_0$ such that

$$M_0 \subseteq N_0, \quad M_1 \subseteq N_1,$$

and

$$X_{M_0} = Y_{N_0}, \quad X_{M_1} = Y_{N_1}.$$

Then,

$$\begin{aligned} X_{M_0} \cap X_{M_1} &= Y_{N_0} \cap Y_{N_1} = \bigcup_{i \in N_0, j \in N_1} (Y_i \cap Y_j) \\ &= \bigcup_{i \in N_0, j \in N_1} (\Omega_i \cap \Omega_j) = \Omega_{N_0} \cap \Omega_{N_1}. \end{aligned}$$

Since $(\Omega_{N_0}, \Omega_{N_1})$ is a decomposition pair, we have

$$\theta(X_{M_0} \cap X_{M_1}) = \theta(\Omega_{N_0} \cap \Omega_{N_1}) \subseteq \theta(\Omega_{N_0}) \circ \theta(\Omega_{N_1}).$$

Also, we know

$$\Omega_{N_0} \supseteq Y_{N_0} = X_{M_0},$$

and

$$\Omega_{N_1} \supseteq Y_{N_1} = X_{M_1}.$$

Thus

$$\theta(\Omega_{N_0}) \subseteq \theta(X_{M_0}),$$

$$\theta(\Omega_{N_1}) \subseteq \theta(X_{M_1}),$$

and

$$\theta(\Omega_{N_0}) \circ \theta(\Omega_{N_1}) \subseteq \theta(X_{M_0}) \circ \theta(X_{M_1}).$$

It follows that

$$\theta(X_{M_0} \cap X_{M_1}) \subseteq \theta(X_{M_0}) \circ \theta(X_{M_1}).$$

Hence

$$R[X_1 \dots X_m] = R[X_{M_0}] | \times | R[X_{M_1}]$$

for any $M_0 \subseteq M, M_1 = M - M_0$.

Since the theorem is true for $m < n$, we have

$$R[X_1 \dots X_m] = R[X_1] | \times | \dots | \times | R[X_m].$$

It follows from Corollary 6 that

$$R[\Omega_1 \dots \Omega_n] = R[\Omega_1] | \times | \dots | \times | R[\Omega_n]. \quad \blacksquare$$

Further discussion of the properties of acyclic join dependencies can be found in Refs. 15 and 16. A linear-time algorithm for testing acyclicity is given in Ref. 28.

VIII. CONCLUSIONS

We have shown that lattice theory is a powerful tool in the analysis of the structure of relational database systems. Using this tool, we have established a unified theory of relations. As we have seen, almost every concept in the existing relational database theory has a counterpart in the lattice theory. This suggests that further study of relations should be carried out within the framework of lattice theory. The independency theory of lattices, which is a generalization of the familiar notion of independency in the geometries,^{18,21} is especially important and relevant to the structure of relational database systems if its relation lattice is modular. This approach may lead to a geometric interpretation of data dependencies and independencies, which would make the theory more intuitive and also more useful for practical application.

The establishment of this algebraic theory of relational databases is done in the same spirit as the construction of probability theory. A probability space is a triple (Ω, Σ, P) , where Ω is the sample space, Σ is a σ -algebra of the subsets of Ω , and P is a real-valued function, called a *probability measure*, defined on the σ -algebra Σ .^{17,29} The notion

Table VIII—Comparison of probability theory and the theory of relational databases

Probability Theory	Theory of Relational Databases
Sample space Ω	Set of attributes Ω
Σ , the σ -Algebra of subsets of Ω	2^Ω , the Boolean algebra of subsets of Ω
Probability measure $P: \Sigma \rightarrow R[0, 1]$	Partition function $\theta: 2^\Omega \rightarrow \Pi[R(\Omega)]$
σ -additivity: $\{X_k\}$ is an denumerable union of disjoint events	Meet-morphism: $\{X_k\}$ is a finite collection of sets of attributes
$P\left(\bigcup_{k=1}^{\infty} X_k\right) = \sum_{k=1}^{\infty} P(X_k)$	$\theta\left(\bigcup_{k=1}^n X_k\right) = \theta(X_1) \cdots \theta(X_n)$
$P(\Omega) = 1$ $P(\emptyset) = 0$	$\theta(\Omega) = \mathbf{0}$ $\theta(\emptyset) = \mathbf{1}$
$0 \leq P(X) \leq 1, \forall X \in \Sigma$	$\mathbf{0} \leq \theta(X) \leq \mathbf{1}, \forall X \in 2^\Omega$
If $X \subseteq Y$, $P(X) \leq P(Y)$	If $X \supseteq Y$, $\theta(X) \leq \theta(Y)$
If Ω_1 and Ω_2 are independent, $P(\Omega_1 \cap \Omega_2) = P(\Omega_1)P(\Omega_2)$	If Ω_1 and Ω_2 are decomposable, $\theta(\Omega_1 \cap \Omega_2) = \theta(\Omega_1) + \theta(\Omega_2)$ $= \theta(\Omega_1) \circ \theta(\Omega_2) = \theta(\Omega_2) \circ \theta(\Omega_1)$

of a σ -algebra of sets also has an abstract generalization, namely it is a particular case of a Boolean σ -algebra.³⁰ A comparison of the algebraic theory of relational databases and probability theory is shown in Table VIII.

We feel that this theory of relational databases can be used to analyze the nonquantitative aspects of data dependencies (or independencies), whereas probability theory is the basis of quantitative data analysis, namely statistics. This comparison is not meant to imply that there is a one-to-one correspondence between the theory of relational databases and the theory of probability. Nevertheless, we are convinced that the lattice theory could play a role in the theory of relational databases similar to the role measure theory plays in the theory of probability.¹⁷

The computational algorithms for meet and join operations of partitions are given in Ref. 31, which provides the basic tools for future development of algorithms for relations.

IX. ACKNOWLEDGMENT

I am grateful to Dr. M. Eisenberg for his careful review of the manuscript and his helpful comments and suggestions.

REFERENCES

1. E. F. Codd, "A relational model of data for large shared data banks," *Comm. ACM*, 3, No. 6 (June 1970), pp. 377-87.
2. E. F. Codd, "Further normalization of the database relational model," in *Database Systems*, R. Rustin, ed., Englewood Cliffs, NJ: Prentice Hall, 1972, pp. 33-64.
3. C. Beeri, P. A. Bernstein, and N. Goodman, "A sophisticated introduction to database normalization theory," *Proc. Int. Conf. on Very Large Databases*, West Berlin, Germany (September 1978), pp. 113-24.
4. P. A. Bernstein and N. Goodman, "What does Boyce-Codd normal form do?" *Proc. 6th Int. Conf. on Very Large Databases*, Montreal, Canada (1980), pp. 245-59.
5. P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Trans. Database Syst.*, 1, No. 4 (December 1976), pp. 277-98.
6. J. D. Ullman, *Principles of Database Systems*, Rockville, MD: Computer Science Press, Inc., 1980.
7. R. Fagin, "Multivalued dependencies and a new normal form for relational databases," *ACM Trans. Database Syst.*, 2, No. 3 (September 1977), pp. 262-78.
8. C. Zaniolo, "Analysis and design of relational schemata for database systems," Ph.D. Diss., Tech. Rep. UCLA-ENG-7661, U. of California, Los Angeles, CA, July, 1976.
9. A. K. Anora, and C. R. Carlson, "The information preserving properties of relational database transformations," *Proc. 4th Int. Conf. on Very Large Databases*, West Berlin, Germany (September 1978), pp. 352-9.
10. J. Rissanen, "Independent components of relations," *ACM Trans. Database Syst.*, 2, No. 4 (December 1977), pp. 317-25.
11. J. Rissanen, "Theory of relations for databases—a tutorial survey," *Proc. 7th Symp Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64*, J. Winkowski, ed., Berlin, Heidelberg: Springer-Verlag, 1978, pp. 537-51.
12. W. W. Armstrong, "Dependency structure of database relationships," *Proc. IFIP*, 74, Amsterdam: North-Holland Publ. Co. (1979), pp. 580-3.
13. C. Beeri, R. Fagin, and J. H. Howard, "A complete axiomization for functional and multivalued dependencies in database relations," *Proc. ACM SIGMOD. Int. Conf. on Management of Data*, Toronto, Canada (1977), pp. 47-61.

14. A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Trans. Database Syst.*, 4, No. 3 (September 1979), pp. 297-314.
15. C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. D. Ullman, and M. Yannakakis, "Properties of acyclic database schemes, Proc. Thirteenth Annual ACM Symp. on Theory of Computing (Milwaukee 1981), pp. 355-62.
16. R. Fagin, A. O. Mendelzon, and J. D. Ullman, "A simplified universal relation assumption and its properties," *ACM Trans. Database Syst.*, 7, No. 3 (September 1982), pp. 343-360.
17. M. Loeve, *Probability Theory*, 3rd ed., Princeton, NJ: Van Nostrand, 1963.
18. G. Birkhoff, *Lattice Theory*, 3rd ed., Providence, RI: American Mathematical Society Colloquium Publ. XXV, 1967.
19. C. Delobel and R. G. Casey, "Decomposition of a database and the theory of Boolean switching functions," *IBM J. Res. and Develop.*, 17, No. 5 (September 1973), pp. 374-86.
20. P. M. Cohen, *Algebra*, Vol. 2, London: John Wiley, 1977.
21. A. Rosenfeld, *An Introduction to Algebraic Structures*, San Francisco, CA: Holden-Day, 1968.
22. H. C. Torng, *Switching Circuits Theory and Logic Design*, Reading, MA: Addison-Wesley Publ. Co., 1972.
23. T. W. Lin, F. W. Tompa, and T. Kameda, "An improved third normal form for relational databases," *ACM Trans. Database Syst.*, 6, No. 2 (June 1981), pp. 329-46.
24. C. Delobel, "Normalization and hierarchical dependencies in the relational data model," *ACM Trans. Database Syst.*, 3, No. 3 (September 1978), pp. 201-222.
25. C. Berri, A. O. Mendelzon, Y. Sagiv, and J. D. Ullman, "Equivalence of relational database schemes, Proc. Eleventh Annual ACM Symposium on the Theory of Computing (1979), pp. 319-29.
26. W. W. Armstrong, and C. Delobel, "Decompositions and functional dependencies in relations," *ACM Trans. Database Syst.*, 5, No. 4 (December 1980), pp. 404-30.
27. F. Sadri and J. D. Ullman, "Template dependency: a large class of dependencies is relational databases and its complete axiomatization," *J. ACM*, 29, No. 2 (April 1982), pp. 363-72.
28. R. E. Tarjan and M. Yannakakis, unpublished work.
29. A. N. Kolmogorov, *Foundation of Probability*, New York: Chelsea, 1950.
30. P. R. Halmos, *Lectures on Boolean Algebra*, New York: Springer-Verlag, 1974.
31. T. T. Lee, "Order-preserving representations of the partitions on the finite set," *J. Combinatorial Theory, Series A.31*, No. 2 (September 1981), pp. 136-45.

APPENDIX A

Properties of Meet and Join Operations

In any lattice $(L, \cdot, +)$, the operations of meet and join satisfy the following laws:

$$L1 - a \cdot a = a, a + a = a; \text{ (Idempotent)}$$

$$L2 - a \cdot b = b \cdot a, a + b = b + a; \text{ (Commutative)}$$

$$L3 - a \cdot (b \cdot c) = (a \cdot b) \cdot c,$$

$$a + (b + c) = (a + b) + c; \text{ (Associative)}$$

$$L4 - a \cdot (a + b) = a + (a \cdot b) = a; \text{ (Absorption)}$$

$$L5 - a \leq b \text{ iff } a \cdot b = a,$$

$$a \leq b \text{ iff } a + b = b; \text{ (Consistency)}$$

$$L6 - b \leq c \text{ implies } a \cdot b \leq a \cdot c$$

$$b \leq c \text{ implies } a + b \leq a + c; \text{ (Isotone)}$$

$$L7 - a \cdot (b + c) \geq (a \cdot b) + (a \cdot c)$$

$$a + (b \cdot c) \leq (a + b) \cdot (a + c); \text{ (Distributive Inequalities)}$$

$$L8 - a \leq c \text{ implies } a + (b \cdot c) \leq (a + b) \cdot c. \text{ (Modular Inequality)}$$

A lattice is called *distributive* if equality holds in L7 and is called

modular if equality holds in L8. A Boolean algebra is a lattice $(L, \cdot, +, \bar{})$ with the following additional properties:³⁰

L9— $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$,

$a + (b \cdot c) = (a + b) \cdot (a + c)$; (*Distributive Identities*)

L10— $a \leq c$ implies $a + (b \cdot c) = (a + b) \cdot c$; (*Modular Identity*)

L11— L contains universal bounds $\mathbf{0}, \mathbf{1}$, which satisfy

$\mathbf{0} \cdot a = \mathbf{0}, \mathbf{0} + a = a$,

$\mathbf{1} \cdot a = a, \mathbf{1} + a = \mathbf{1}$;

L12— $\forall a \in L, \exists \bar{a} \in L$ such that

$a \cdot \bar{a} = \mathbf{0}, a + \bar{a} = \mathbf{1}, \bar{\bar{a}} = a$,

$\overline{(a \cdot b)} = \bar{a} + \bar{b}, \overline{(a + b)} = \bar{a} \cdot \bar{b}$.

APPENDIX B

The Proofs of Axioms for Functional and Multivalued Dependencies

The first three of the following are Armstrong's axioms for functional dependencies:¹²

B1. (Reflexivity for functional dependencies)

If $X \subseteq Y \subseteq Z$, then $X \rightarrow Y$.

Proof: $\theta(X) = \theta(Y(X - Y)) = \theta(Y)\theta(X - Y) \leq \theta(Y)$. ■

B2. (Augmentation for functional dependencies)

If $X \rightarrow Y$ and $Z \subseteq W$, then $XZ \rightarrow YZ$.

Proof: $\theta(XZ) = \theta(X)\theta(Z) \leq \theta(Y)\theta(Z) = \theta(YZ)$. ■

B3. (Transitivity for functional dependencies)

If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Proof: $\theta(X) \leq \theta(Y)$ and $\theta(Y) \leq \theta(Z)$ imply $\theta(X) \leq \theta(Z)$. ■

The next three axioms apply to multivalued dependencies:¹³

B4. (Complementation for multivalued dependencies)

If $X \twoheadrightarrow Y$ then $X \twoheadrightarrow \Omega - X - Y$.

Proof: $\theta(X) = \theta(XY) + \theta(XZ) = \theta(XY) \circ \theta(XZ) = \theta(XZ) \circ \theta(XY)$, where $Z = \Omega - X - Y$. ■

B5. (Augmentation for multivalued dependencies)

If $X \twoheadrightarrow Y$, and $V \subseteq W$, then $WX \twoheadrightarrow VY$.

Proof: Without loss of generality,* we can let $\Omega = ABCDEFGHIJKL$, $X = ABCDEF$, $Y = BCGHFI$, $W = CDEFHIJK$, $V = EFIJ$ (see Fig. 9). Then $\Omega - X - Y = JKL$ and $\Omega - WX - VY = L$.

We want to show that

$$\theta(ABCDEF) \subseteq \theta(ABCDEFGHI) \circ \theta(ABCDEFJKL)$$

* This proof is carried out in terms of equivalence relations (partitions). It is irrelevant here whether an equivalence relation is the image of a single attribute or the image of a set of attributes.

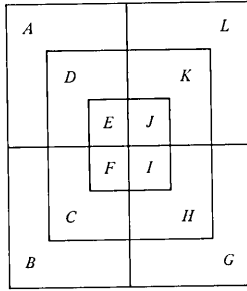


Fig. 9—Set of attributes Ω for B5.

implies

$$\theta(ABCDEFHIJK) \subseteq \theta(ABCDEFGHIIJK) \circ \theta(ABCDEFHIJKL).$$

Suppose

$$t_1\theta(ABCDEFHIJK)t_2. \quad (1)$$

Then

$$t_1\theta(ABCDEF)t_2.$$

There exists t_3 , such that

$$t_1\theta(ABCDEFGHI)t_3\theta(ABCDEFJKL)t_2. \quad (2)$$

From (1) and (2), we have

$$t_3\theta(JK)t_2\theta(JK)t_1.$$

It follows from (2) that

$$t_1\theta(JKG)t_3. \quad (3)$$

Combining (2) and (3), we have

$$t_1\theta(ABCDEFGHIIJK)t_3. \quad (4)$$

Relation (2) also implies that

$$t_1\theta(HI)t_3.$$

From (1), we know

$$t_1\theta(HI)t_2,$$

and therefore

$$t_3\theta(HI)t_2. \quad (5)$$

It follows from (2) and (5) that

$$t_3\theta(ABCDEFHIJKL)t_2. \quad (6)$$

Combining (4) and (6), we have

$$t_1\theta(ABCDEFGH) t_3\theta(ABCDEFHIJKL)t_2.$$

It follows that

$$\theta(ABCDEFHIJK) \subseteq \theta(ABCDEFGH) \circ \theta(ABCDEFHIJKL). \quad \square$$

B6. (Transitivity for multivalued dependencies)

If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow Z - Y$.

Proof: Again, without loss of generality, we can let $\Omega = ABCDEFGH$, $X = AFGH$, $Y = BCFG$, $Z = CDGH$ (see Fig. 10). Then $Z - Y = DH$, $\Omega - XY = DE$, $\Omega - YZ = AE$, $\Omega - X(Z - Y) = BCE$.

We want to show that

$$\theta(AFGH) \subseteq \theta(ADEFGH) \circ \theta(ABCFGH)$$

and

$$\theta(BCFG) \subseteq \theta(ABCEFG) \circ \theta(BCDFGH)$$

imply

$$\theta(AFGH) \subseteq \theta(ADFGH) \circ \theta(ABCEFGH).$$

Suppose $t_1\theta(AFGH)t_2$. Then there exists t_3 such that

$$t_1\theta(ADEFGH)t_3\theta(ABCFGH)t_2. \quad (7)$$

Since $t_2\theta(BCFG)t_3$, there exists t_4 such that

$$t_2\theta(ABCEFG)t_4\theta(BCDFGH)t_3. \quad (8)$$

It follows that

$$t_1\theta(AFG)t_3\theta(AFG)t_2\theta(AFG)t_4. \quad (9)$$

From (7) and (8), we have

$$t_1\theta(DH)t_3\theta(DH)t_4. \quad (10)$$

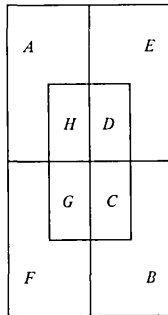


Fig. 10—Set of attributes Ω for B6.

Combining (9) and (10) yields

$$t_1\theta(ADFGH)t_4. \quad (11)$$

From (7) and (8), we have

$$t_4\theta(H)t_3\theta(H)t_2.$$

It follows from (8) that

$$t_4\theta(ABCEFGH)t_2. \quad (12)$$

Relations (11) and (12) yield

$$t_1\theta(ADFGH)t_4\theta(ABCEFGH)t_2.$$

Hence

$$\theta(AFGH) \subseteq \theta(ADFGH) \circ \theta(ABCEFGH). \quad \square$$

The last two axioms relate functional and multivalued dependencies.

B7. If $X \rightarrow Y$ then $X \twoheadrightarrow Y$.

Proof: Let $Z = \Omega - XY$. We want to show that

$$\theta(X) \subseteq \theta(Y) \text{ implies } \theta(X) \subseteq \theta(XY) \circ \theta(XZ).$$

Suppose $t_1\theta(X)t_2$. Since $\theta(X) \subseteq \theta(Y)$ implies $\theta(XY) = \theta(X)$, then $t_1\theta(XY)t_2$. It follows that

$$t_1\theta(XY)t_2\theta(XZ)t_2.$$

Hence

$$\theta(X) \subseteq \theta(XY) \circ \theta(XZ). \quad \square$$

B8. If $X \twoheadrightarrow Y$, $Z \subseteq Y$, and for some W disjoint from Y , we have $W \rightarrow Z$, then $X \rightarrow Z$.

Proof: Again, without loss of generality, we can let $\Omega = ABCDEFGH$,

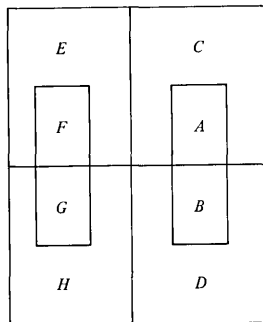


Fig. 11—Set of attributes Ω for B8.

$X = ACEF$, $Y = EFGH$, $Z = FG$, and $W = AB$ (see Fig. 11). Then $\Omega - XY = BD$.

We want to show that

$$\theta(ACEF) \subseteq \theta(ACEFGH) \circ \theta(ABCDEF)$$

and

$$\theta(AB) \cong \theta(FG)$$

imply

$$\theta(ACEF) \cong \theta(FG).$$

Suppose $t_1\theta(ACEF)t_2$. Then there exists t_3 such that

$$t_1\theta(ACEFGH)t_3\theta(ABCDEF)t_2.$$

Since

$$t_1\theta(FG)t_3 \quad \text{and} \quad t_3\theta(AB)t_2,$$

we have

$$t_1\theta(FG)t_3 \quad \text{and} \quad t_3\theta(FG)t_2,$$

and thus

$$t_1\theta(FG)t_2.$$

Hence

$$\theta(ACEF) \cong \theta(FG). \quad \blacksquare$$

AUTHOR

Tony T. Lee, B.S. (Electrical Engineering), 1971, National Cheng Kung University; M.S. (Mathematics), 1973, Cleveland State University, M.S., 1976, Ph.D. (Electrical Engineering), 1977, Polytechnic Institute of New York; Bell Laboratories, 1977—. At Bell Laboratories, Mr. Lee has worked on Common Control Interoffice Signaling network planning, teletraffic usage forecasting, 5ESS™ message switch performance analysis, and queueing modeling of computer and communication systems. He is currently working in the Teletraffic Theory and Application Department at Bell Laboratories. Member, Sigma Xi.

Generation of Syntax-Directed Editors With Text-Oriented Features

By B. A. BOTTOS* and C. M. R. KINTALA†

(Manuscript received March 16, 1983)

Often, syntax-directed editors rely solely on menu selection for program construction. We describe here the generation of *hybrid editors* that give a programmer the option of either (1) using menu selection and tree navigation as in a syntax-directed editor, or (2) entering text for parsing and navigating through the text as in a conventional editor at *any* stage during the expansion of a program. A prototype system, HEG (Hybrid Editor Generator), has been built to automatically generate such a hybrid editor from a high-level specification of a grammar for an application language. Each such generated hybrid editor is called an AGE (Automatically Generated Editor). We describe the HEG meta-language and briefly summarize the editing process in AGEs. We also describe possible extensions to the meta-language to describe program semantics, and the generation of the procedures to check those semantics during program construction.

I. INTRODUCTION

In the past, there has been a dichotomy between the way a development tool such as a text editor would manipulate the text of a computer program and the way another development tool such as a parser would manipulate the same text. The advent of syntax-directed editing has removed this difference by introducing the use of editors that store and manipulate programs entirely as (partially) instantiated syntax trees.¹⁻⁸ The question, though, of whether a program should be manipulated *only* in terms of its syntax tree, or also in terms of its

*Carnegie-Mellon University. †Bell Laboratories.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

textual representation, is yet to be resolved.^{8,9} Some editing systems purport to allow the programmer both points of view, but provide only one, predetermined, choice for the manipulation of each construct in the language.⁷ We introduce here the concept of a *hybrid editor*, which integrates the tree-navigation and menu-selection capabilities of a syntax-directed editor with the text-navigation and the string-entry capabilities of a conventional editor. Either of these views of a program can be taken at *any* stage during the expansion of the program in a hybrid editor.

One of the features of the hybrid editors discussed herein is the fact that a complete editing system is automatically generated from a high-level description of a language. The concrete and abstract syntaxes of the language are both described by one context-free grammar; and both the syntax-directed editor, and the incremental parser for use with the editor, are generated from this grammar.

In this paper, we outline the overall design of a hybrid editor generator and describe briefly the interface presented to a programmer by the hybrid editor. We define the meta-language in which the application language is specified for the generator, and describe the operations performed on the language specification in the process of generating the editor. Based on this design, a prototype hybrid editor generator, called HEG, has been built and is being tested. We describe possible extensions to the meta-language to describe program semantics, and the generation of the procedures to check those semantics during program construction.

II. BACKGROUND

Although syntax-directed editing can be very helpful to a programmer who is unfamiliar with the language he is using, one who knows the syntax of the language may easily become frustrated with the plethora of menu choices that must be made to "write" a program substructure as simple as an assignment statement. For this reason, allowing the programmer the option of giving the editor a string to parse and insert into the partially expanded program is a desirable feature to have in such a menu-driven editor. We define a hybrid editor to be a syntax-directed editor with the ability to parse and to integrate into the program tree a string given at any time during the expansion of a program.

There have been attempts to generate syntax-directed editors automatically for several programming languages from specifications of the syntactic (given by a context-free grammar)⁵ and the semantic (given using attributes for the symbols in the grammar) aspects of a language.¹⁰ There is also a system in use that requires only the addition of a context-free grammar for any new language for which it is to

operate.⁴ (In this latter system, detailed knowledge of the structure of the grammar seems to be required in order to use the editor with any grace.) Usually, however, such editors are built individually for each programming language. This is because (1) high-level programming languages have many context-sensitive semantic constraints that cannot be expressed by a context-free grammar, (2) programmers using high-level languages usually need more local context-sensitive aid than syntactic help, and (3) the external interface that such an editor is expected to provide depends heavily upon the language it supports.

Little attention, however, has been paid to syntax-directed editors for special-purpose languages such as the input specification languages for Yet Another Compiler-Compiler (YACC)¹¹ and database interface languages such as HISEL.¹² Yet it is for these seldom used, but numerous, languages that a hybrid editor would be most useful. Users of such application programs often write their input in a file using a regular text editor and manually check conformity with the syntactic constraints imposed by the particular application program. Some of these application programs have parsers in their front ends to check the syntactic correctness of their input. (Sometimes these parsers are automatically generated from utilities such as YACC.¹¹) Others just assume that their input is syntactically correct and abort when it is not. If, in place of the parser, a hybrid editor is available within the application program, the user can be guided by the program's editor during input construction.

A tool to automatically generate such editors for application languages can be quite useful for several reasons: (1) A special-purpose application language, unlike a programming language, is likely to change more rapidly with an evolving application program. (2) Many application programs are not frequently used and hence their idiosyncratic syntactic constraints are likely to be forgotten. (3) The amount of syntactic help from a hybrid editor can be determined by the programmer. (4) The derivation tree built by the hybrid editor may be used by the application program to process its input in a more structured manner than is often possible when only a textual view of the input is available to the program. It is for these types of languages that HEG-generated hybrid editors, called AGEs (Automatically Generated Editors), are most valuable.

III. THE EDITING PROCESS

In a HEG-generated hybrid editor for a given language, a program is internally represented by a derivation tree of the program in that language, along with a symbol table containing the programmer-defined character strings. (See Appendix A for an example of a short session with an AGE.) When the programmer wishes to expand a

particular nonterminal in the tree by syntax-directed menu selection, and if there are two or more production rules for that nonterminal, then the AGE creates a one-item-at-a-time wraparound menu, on the bottom line of the screen, displaying the menu strings associated with those production rules. After a production rule is chosen for expansion, the internal node for the nonterminal is grown in such a way that the frontier of the subtree rooted at that internal node corresponds to the right-hand side of the production rule selected. If, however, the programmer wishes to forego menu selection, he/she may expand a nonterminal by entering a string that is derivable from the nonterminal. The editor will parse the string, build a subtree representing the string, and graft the subtree into the program tree at the node corresponding to the nonterminal. The string given by the programmer may be any combination of terminal strings and nonterminal names.

Even though the internal representation of a program is a tree, both tree and text interfaces are provided to the programmer for navigation through the program. At any moment, a tree cursor navigating around the internal nodes of the tree is available. The programmer sees the cursor spanning the entire frontier of the subtree rooted at the tree-cursor node. He can move the tree cursor along the internal nodes of the tree (using the commands up [^], down [V], left-sibling under the same parent [<], right-sibling under the same parent [>], next internal node of the same type under the same parent [N], etc). Or, he can navigate textually (using the commands n for next word, b for back word, CR for next line, - for previous line) through the program. The commands 'e' and 'u' provide syntax-driven expansion and unexpansion facilities. Commands 'p' and 'r' allow parser-driven nonterminal expansion and file-reading facilities.

When the programmer quits editing, both the text and the tree versions of the program are saved. The tree is saved by storing the leftmost derivation sequence of the production rules in the derivation tree and the symbol table. This sequence is used to reconstruct the tree for a later editing session on that program. The text is saved for possible use by other tools.

IV. ARCHITECTURE OF HEG

HEG produces a generalized table-driven syntax-directed editor, linked with a parser, for a given application language. The application language is specified by an AGE grammar in the meta-language described in the next section. A parser generator front end, named 'GENPAR', generates a YACC file and a LEX file¹³ from this specification of a language. The parser generated from these two files is capable of parsing strings, which may contain nonterminal names,

starting from any nonterminal of the grammar. The grammar tables used by the editor, the parser, and the generic editing routines are then linked together to generate an AGE for the given language.

Figure 1 shows the interactions between the different components of an AGE.

V. THE META-LANGUAGE

The “meta-language” for HEG is the grammar specification language in which the production rules of any context-free grammar are specified, along with the pretty-printing information required by the display utilities of the editor (e.g., indentation and color codes, if available) and the strings for the menu lists. As an example of the use of the language,

```

query      : <QUERY>
           | “select” list “where” conds
           ;
    
```

is a normal production rule. Here, <QUERY> is a “user-friendly” name for the nonterminal symbol ‘query’. The sequence of strings following the character ‘|’ specifies a production for the nonterminal ‘query’. Strings within double quotes in the production specification are terminal characters and others are nonterminals. The terminal character strings may also contain the following special characters denoting pretty-printing information: ‘\n’ for a new-line, ‘~’ for a blank character, ‘\t’ for tabbing one position to the right and increasing the tab count by one for the starting positions of the subsequent lines, and ‘\T’ for decreasing the tab count by one for the current and the subsequent lines.

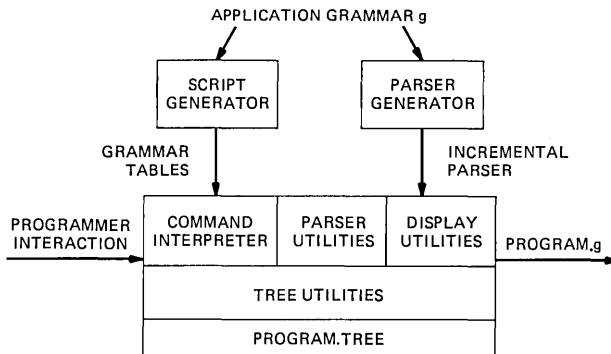


Fig. 1—Interactions between the components of an AGE.

Another example is

```
list      : ITEM_LIST
          | +item “,”
          ;
```

This is an iterative production rule. It specifies that the nonterminal, ‘list’, can be expanded into one or more occurrences of the nonterminal, ‘item’, separated by the terminal character string ‘,’. A ‘*’, in place of the ‘+’ above, would denote zero or more occurrences of the nonterminal.

There may be more than one production rule for a nonterminal; if so, all of the rules for the nonterminal must be specified in one rule set, each preceded by a ‘|’. For example, the following two sets of rules show several possible expansions for the corresponding nonterminals:

```
clause   : CLAUSE
          | field_name “~” op “~” constant
          | constant “~” op “~” field_name
          | field_name “~” op “~” field_name
          ;

op       : OPERATOR
          | “=”
          | “!”
          | “>=”
          | “<=”
          | “>”
          | “<”
          ;
```

For each production rule, the grammar specification must include an associated “expansion character”. This expansion character must be unique within the set of rules for that nonterminal. If there are two or more rules for the expansion of a nonterminal and if an instance of that nonterminal in the program is to be expanded, the programmer must indicate his choice by typing the expansion character associated with the desired rule. However, if the programmer wishes to examine all the choices for that nonterminal, the AGE creates a one-item-at-a-time wraparound menu on the bottom line of the screen, from which the programmer may choose a production. Therefore, the implementor (the person defining the grammar) must provide a “menu-item-string”, which is displayed in the menu for each production rule. These menu items should be suggestive of the associated production rule for better communication with the programmer. When a particular rule is chosen for the expansion, the right-hand side of that rule replaces the left-hand side nonterminal node in the program’s syntax tree.

To illustrate the provision of menu information in a grammar for HEG as above and to clarify the grammar specifications further, a complete set of rules for a database query language HISEL¹² is given below. Sentences (queries) in this language have the form:

select x1.fld1,x2.fld2, ... where <CLAUSES> or <CLAUSES> ...

where 'x' is a cursor name, 'fld' is a field name, and <CLAUSES> is a conjunctive sequence of field conditions. The single characters following the menu items in the rules are the expansion characters. For the rule sets having just one production rule, meaningless menu-item strings (e.g., 'xxx') and expansion characters (e.g., 'x') are used for uniformity in the meta-syntax. Observe that there is no white space in the menu-item strings.

```

query      : <QUERY>
           | xxx x "select" list "\nwhere~" conds
           ;
list       : <ITEM_LIST>
           | xxx x + item ","
           ;
item       : ITEM
           | xxx x cursor "." field_name
           ;
conds      : <CONDITIONS>
           | xxx x *disjunct "\n~~~or~"
           ;
disjunct   : <CLAUSES>
           | xxx x +clause "~,"
           ;
clause     : CLAUSE
           | field_op_const 1 field_name "~" op "~" constant
           | const_op_field 2 constant "~" op "~" field_name
           | field_op_field 3 field_name "~" op "~" field_name
           ;
op         : OPERATOR
           | equal = "="
           | not_equal ! "!="
           | greater_equal 1 ">="
           | less_equal 2 "<="
           | greater > ">"
           | less < "<"
           ;

```

The left-hand side nonterminal of the first rule in the grammar specification (i.e., 'query', above) is the starting nonterminal. If no

production rule is available for a nonterminal (e.g., 'field_name'), and that nonterminal appears during the derivation of a program, then it is assumed to expand into a character string to be provided by the programmer at the time of expansion. Such nonterminals are said to derive "identifiers". A regular expression specification for any such nonterminal can be used to restrict the format of the identifier strings that a programmer may supply at the time of expansion. HEG provides a default specification for all nonterminals deriving identifiers and having no regular expression specification.

As another example, the following is a grammar in the meta-language (meta-grammar) for the meta-language described in this section. In fact, one can invoke an AGE for this meta-language to enter a grammar specification for any user-defined language.

```

gram      : <AN_AGE_GRAMMAR>
           | xxx x cfrules "\n%%\n" rerules
           ;
cfrules   : <CONTEXT_FREE_RULES>
           | xxx x +ruleset "\n"
           ;
ruleset   : <A_RULE_SET>
           | xxx x nonterm-name "\t:~" user_name "\n" rules
             "\n;\n\T"
           ;
rules     : <RHS_OF_A_RULE_SET>
           | xxx x +rule "\n"
           ;
rule      : <A_RULE>
           | normal-rule n "|~" menu_string "~" exp_char "~"
             tklist
           | nonempty-rec-rule + "|~xxx~x~+" nonterm_name
             "~\" separator "\"
           | empty-rec-rule * "|~xxx~x~*" nonterm_name "~\"
             separator "\"
           ;
tklist    : <A_SEQ_OF_TOKENS>
           | xxx x +token "~"
           ;
token     : TOKEN
           | nonterminal-token n nonterm_name
           | terminal-token t "\" terminal_str "\"
           ;
rerules   : <REGULAR_EXP_RULES>
           | xxx x *rerule "\n"
           ;

```

```

rerule      : <REG_EXP_RULE>
             | xxx x nonterm-name “~\~##!” reg_expr
             ;
%%
exp_char    ~##![a-zA-Z0-9+*^<>&]

```

VI. THE PARSER GENERATOR FOR HEG

The tools YACC and LEX are used to produce a parser and a scanner for an AGE system. The parser generator front end, GENPAR, takes, as input, the AGE grammar specification for the desired language and produces YACC and LEX specification files. (Note that the use of YACC implies that the input grammar must be LALR(1) in order to generate a parser for an AGE system.)

The generated parser can be invoked to parse an input string representing the expansion of any nonterminal node in the program derivation tree. The string to be parsed can be any combination of terminal strings and nonterminal names that is derivable from the nonterminal at the “current” position of the editing cursor.

6.1 *The parsing of an input string*

When the programmer provides a string to be parsed, rather than making a menu selection, the AGE system opens a temporary file, writes the prefix `$$NONTERMINAL_NAME$$` to the file (where `NONTERMINAL_NAME` represents the nonterminal from which the programmer’s input string is to be derived), and appends the input string. The parser is called to process the entire string contained in the file. The prefix is considered to be an integral part of the input string. If the portion of the string entered by the programmer cannot be derived from the nonterminal indicated by the prefix, the input will be considered syntactically incorrect. If a syntax error is found, the temporary file is saved in case the programmer should wish to edit the string and resubmit it to the parser.

The parser will parse its input in a “backtracking bottom-up” fashion, constructing a syntax tree to represent the derivation of the input string. After the tree is constructed, a preorder traversal of the tree is performed, producing a list of production numbers representing the leftmost derivation of the input string. This list is then passed back to the controlling program in AGE.

In an AGE, the YACC-generated parser is run subordinate to a “parser monitor” to provide a certain amount of parser backtracking, made necessary by the conflict resolution scheme of the LEX-generated scanner. For such a scanner, the lexical tokens desired are specified by regular expressions. When two or more of the given regular expressions match equal-length segments of the input (starting at the

“current input position” of the scanner), and these are the longest matches possible, the scanner will select the regular expression that had been listed first (textually) in the input specification file as the “correct” match, and return the corresponding token number. If the parser is expecting one of the other possible matches at the current input position, the parser will find a “syntax error” where no error may, in fact, exist.

The parsers produced by the YACC program are standard shift/reduce (“bottom-up”) parsers. The required backtracking is accomplished through the interaction of code at two levels of the parsing scheme. At the lowest level, the handling of multiple matches in the scanner is slightly modified by forcing the scanner to “reject” an “identifier” match, after linking the token number associated with the match into a “token map.” In this manner, all possible matches for an “identifier” are linked into the token map for the parse. At the highest level, the parser monitor runs the shift/reduce parser, handing the parser token numbers from either the scanner or the token map, depending upon the current state of the parse. Then, if the YACC-generated portion of the parser discovers a syntax error in the input, the monitor can rotate the last set of entries in the token map to (temporarily) “forget” the “preferred” token number for the last “choice position” in the input stream, allowing the use of another possible token number for that position.* In this manner, no syntactically correct input will be declared to contain a syntax error, and an incorrect input will only be rejected after trying the allowable combinations of token numbers for the “identifier” positions.

6.2 What the parser generator does

Several transformations must be applied to the AGE grammar to produce a grammar specification that is acceptable to YACC, and that will specify a parser offering the features and enforcing the constraints desired. All the transformations are performed and the YACC grammar specification is produced in a single pass over the input grammar.

6.2.1 Starting the derivations from arbitrary nonterminals

For every nonterminal in the AGE grammar, two additional productions are generated. One such set of productions makes it possible to

*Due to the relative simplicity of most application languages (in terms of permissible “identifier” combinations and possible token map complexity), the erroneous “identifier” token number will usually be in the last set of entries in the token map. So, in most cases, only the last one or two “identifiers” need to be retried (if the input really is syntactically correct). In all cases, the LALR(1) property of the grammar should aid in the isolation of groups of points in the “identifier” cross-product space that could not possibly be characteristic of a correct parse. The points in these groups, then, need not be considered individually during parsing.

start a derivation from any nonterminal in the original grammar, even though YACC will allow the specification of only one start symbol for a grammar. Each such added production is of the form:

```
ppstart : AGE_TERM_20 clause
```

where “ppstart” is a newly defined start symbol for the grammar, “clause” is a nonterminal in the original grammar, and “AGE_TERM_20” is the token returned by the scanner when it reads the prefix (e.g., “\$\$clause\$\$”) encoding the nonterminal from which the remainder of the input is to be derived. This type of production also allows the enforcement of the rule that the input string must be derivable from the nonterminal at the “current” node in the program tree.

6.2.2 Use of nonterminal names

The second set of productions generated for each nonterminal allows the acceptance of a “user-name” for a nonterminal, in place of a string that could be derived from that nonterminal, wherever the nonterminal may appear in an input sentence. Each of these productions is of the form:

```
clause : AGE_TERM_22
```

where “AGE_TERM_22” is the token returned by the scanner when it reads the user-name for the nonterminal on the left-hand side of the production (e.g., “clause”).

6.2.3 Iteration in the grammars

The iterative specifications in the AGE grammar must be transformed into explicit left recursions, with the separators appropriately treated. Since the most general form of iteration in AGE grammars is that of a list item repeated zero or more times, with a nonwhite-space separator, that case is considered here. The AGE grammar specification for such a list might be:

```
clauses : *clause “,”
```

where “clause” is the nonterminal to be repeated in the list structure and “,” is the terminal separator to appear between list elements.

The YACC specification corresponding to the above production would be:

```
clauses : Xclause_2_0X
```

where Xclause_2_0X is a new nonterminal encoding the nonterminal to be repeated as the list elements (e.g. “clause”), the number of the terminal string to be used as the list element separator (e.g., “2”), and

the minimum number of times the nonterminal must appear in the list (e.g., "0"). Such encoding permits the use of the list item nonterminal, "clause," as the repeated element in other lists with different separators and/or different minimal numbers of occurrences.

To derive the required number of occurrences of "clause," sets of productions of the following form must be added to the YACC grammar:

```
Xclause_2_0X : e
              | Xclause_2_1X
```

where "e" represents the empty string. The second new nonterminal, Xclause_2_1X, denotes one or more occurrences of "clause" separated by occurrences of terminal number 2. For this second new nonterminal, sets of productions of the following form are added to the YACC grammar:

```
Xclause_2_1X : clause
              | Xclause_2_1X AGE_TERM_2 clause
```

where "AGE_TERM_2" denotes the required separator ("terminal number 2") between elements of the list.

Note that other iterative specifications are specializations of the above case. If the list must be nonempty, only the second new nonterminal, with its associated productions, is generated. If the list element separator is not significant (i.e., if it is any form of white space), then no terminal is encoded in the new nonterminal(s) or included in any of the new productions.

6.2.4 Treatment of identifiers

In an AGE input grammar, there are no explicit productions for the derivation of character strings representing identifiers. Any nonterminal that does not appear on the left-hand side of any production, in an AGE grammar specification, may produce an identifier. Since there are no such implied rules in a YACC grammar specification, GENPAR must add explicit productions to the grammar to allow the reduction of terminal identifier strings to appropriate nonterminals. These productions are of the form:

```
cursor : AGE_IDENTIFIER_5
```

where an identifier number (e.g., "5") is specified only if the default lexical specification is overridden for the nonterminal on the left-hand side of the production (see below).

To handle the lexical specification for each terminal identifier, the grammar designer has two choices. The designer may use GENPAR's default specification, which allows identifiers to be any combination

of letters, digits, and the characters “-”, “_”, and “.”, which begins with a letter, and is not a reserved terminal string in the language. He may, instead, associate an arbitrary LEX specification with any non-terminal that may derive an identifier. The given specification would then be used to override the default identifier specification for that particular nonterminal.

6.2.5 Treatment of white space

In the generation process, all types of white space in the AGE grammar specifications of terminal strings are treated identically, and are considered insignificant to the specification of the generated grammar. For example, if a “PROGRAM” is specified as a “list of statements separated by new-lines”, the YACC-generated parser will accept any “list of statements separated by any white space (e.g., blanks, tabs, or new-lines)” as a “PROGRAM”.

6.3 Generation of YACC actions

If the input string to the parser is valid, AGE requires the output of the parser to be a list of rule numbers, symbol table indices, and list item occurrence counts representing the leftmost derivation of the input string. Since the YACC-generated parser is a shift/reduce parser, the order in which that parser uses the grammar productions will not represent a leftmost derivation of the input. To produce the proper input for the AGE monitor, the parser builds a tree to represent the derivation of its input string as it parses the input string, and, as part of the last production applied (reduction to the start symbol), performs the preorder traversal of the tree, generating the required list of numbers.

The building of this tree is the main activity of the “actions” generated for each production in the YACC specification. To generate code to appropriately link nonterminal sub-trees, the positions of all the nonterminals in a given production rule must be saved as the production is processed by the generator. This is accomplished by counting the tokens on the right-hand side of the given production rule and stacking the position numbers that correspond to nonterminals. Actions are then generated that call precoded subroutines that link the tree nodes together. These subroutines, and their associated data and type declarations, are constant across all grammars, and are included in the appropriate sections of each generated YACC grammar.

6.4 The size of the processed grammar

The growth of the grammar in the generation process is actually not as large as may be imagined. If n = the total number of nonterminals and r = the number of iterative nonterminals in the original

AGE grammar, at most $3n + 3r < 6n$ productions, $2n - 1$ nonterminals, and between $2n + 1$ and $3n - r$ terminals are added to the grammar before YACC generates the parser.

VII. ATTRIBUTES AND SEMANTIC PROCESSING

One possible extension to this work would involve the addition of attributes to the grammars described above. These attributes could allow a certain amount of semantic checking of user programs, in addition to allowing interpretation and/or code generation (for simple application languages) during program construction. The basic principles behind attribute grammars are described elsewhere;¹⁴ we shall only discuss the required extensions to our meta-language and the generation of evaluation functions for the attributes.

7.1 Extensions to the meta-language

The meta-language described in Section V can be enhanced to include attributes and their evaluation routines in each production. An example of the use of the enhanced meta-language is shown below.

```
clause      : CLAUSE
             inherited: int temp_loc;;
             synthesized: int next_temp;
             | field_op_const 1 field_name "~" op "~" constant
             {
               $0.next_temp = $0.temp_loc;
             }
             | const_op_field 2 constant "~" op "~" field_name
             {
               $0.next_temp = $0.temp_loc;
             }
             | field_op_field 3 field_name "~" op "~" field_name
             {
               $0.next_temp = $0.temp_loc + 2;
             }
             ;
```

In general, the attributes and their evaluation rules are specified in a pseudo-C language notation. We provide data typing facilities for the attribute variables.* The inherited attributes of a nonterminal and their data types are listed after the key word "inherited", which appears after the user-name of that nonterminal. The synthesized attributes

*As with the attributed grammars for programming languages, our experience in using attributed grammars for application languages suggests that facilities that include standard libraries of functions, user-defined types, and global attributes are needed.

of that nonterminal are then listed in a similar fashion. The attribute evaluation rules are listed after each production specification, within braces. “\$0.attribute_name” in these rules refers to the correspondingly named attribute of the nonterminal in the left-hand side of the production. “\$i.attribute_name” refers to the correspondingly named attribute of the *i*th nonterminal in the right-hand side of the production.

For each inherited attribute of each nonterminal appearing in the right-hand side of a production rule, there must be an evaluation rule (a function call or an arithmetic expression) assigning a value to that attribute associated with that production rule. Similarly, for each synthesized attribute of the left-hand side nonterminal of a production rule, there must be a rule (a function call or an arithmetic expression) assigning a value to that attribute. These rules may take as arguments the inherited attributes of the left-side nonterminal and/or synthesized attributes of the nonterminals in the right-hand side. These rules must not, however, violate the properties defining *L*-attributedness¹⁴ of the grammar in order for the proposed evaluation scheme to work.

For iterative production rules, the semantic specifications appear as:

```

disjunct      : (CLAUSES)
               inherited;;
               synthesized;;
               |xxx x +clause “~,~”
               {
               int current_temp;
               init: {
                   current_temp = 0;
               }
               repeat: {
                   $1.temp_loc = current_temp;
                   current_temp = $1.next_temp;
               }
               }
               ;

```

The routine following the key word “init:” is executed when this production rule is initially chosen. Then, for each instance of the nonterminal “clause” under the parent “disjunct”, the specification following the key word “repeat:” is used. The “\$1” in the latter specification refers to the instance of the nonterminal “clause” whose attributes are being evaluated. These evaluation specifications must follow the same guidelines as those of the regular productions. Nonterminals deriving “identifiers” (i.e., those having no production rules)

are assumed to have only one synthesized attribute, "value", whose value is the string entered by the programmer at the time of expansion.

7.2 Generation of the attribute evaluators

In any specification of an application language as above, each implementor-defined attribute evaluation function is associated with a specific production. For each distinct attribute in the grammar, the calls to these functions are collected into one generated evaluation function, which determines the appropriate implementor-defined function to call, based on the rule that produced the associated nonterminal. To make the required environment information accessible during the attribute evaluation, a pointer to the tree node at which the generated function is to be evaluated is passed as a parameter to the generated function. The evaluation functions for inherited attributes must be passed a pointer to the parent node of the nonterminal node with which the attribute is associated (i.e., inherited attributes are evaluated when the associated nonterminal appears on the right-hand side of a production). Synthesized attribute evaluation functions must be passed a pointer to the nonterminal node with which the attribute is associated (i.e., synthesized attributes are evaluated when the associated nonterminal is expanded).

Each generated evaluation function for an inherited attribute contains a section of code for each production in which the associated nonterminal can appear on the right-hand side. Similarly, each generated evaluation function for a synthesized attribute contains a section of code for each production in which the associated nonterminal appears on the left-hand side. Within each of these sections of code is the code for the appropriate implementor-defined function, along with the function calls required to evaluate the actual parameters of the implementor's function. The attributes are evaluated only as required, and the evaluation nesting is managed automatically by the C (target programming language) procedure calling mechanism.

VIII. REMARKS

The editor generator HEG, as described here (excluding the semantic analysis), currently exists and has been used for a variety of application languages. The novel aspects of this system include: (1) the ability to give a programmer the option to use menu selection or to enter strings containing terminal characters and nonterminal names at *any* stage during the expansion of a program in the user-language, and (2) the ability to generate a hybrid editor from a high-level specification of a user grammar. This system was an experiment to investigate only these aspects of editing. Various other related issues such as building a robust syntax-sensitive editor for higher-level

languages (e.g., C language), experimenting with different ways of exhibiting the menus, and human-factors issues are being addressed separately.⁵

REFERENCES

1. V. Donzeau-Gouge et al., "A Structure-Oriented Program Editor: a First Step Towards Computer-Assisted Programming," Proc. of Int. Computing Symp., Antibes, June 1975.
2. P. H. Feiler and R. Medina-Mora, "An Incremental Programming Environment," Carnegie-Mellon University, Computer Science Department Report, December 1980.
3. C. N. Fischer, G. Johnson, and J. Mauney, "An Introduction to Release 1 of Editor Allan Poe," University of Wisconsin, Technical Report 453, 1981.
4. C. W. Fraser, "Syntax-Directed Editing of General Data Structures," Proc. ACM SIGPLAN-SIGOA Symp. Text Manipulation, Portland, Oregon, June 1981, pp. 17-21.
5. E. R. Gansner et al., "SYNED—A Language Based Editor for an Interactive Programming Environment," Spring COMPCON 83 San Francisco, California, February 1983, pp. 406-10.
6. M. R. Horton, "Design of a Multi-Language Editor with Static Error Detection Capabilities," Ph.D. Dissertation, Computer Science, University of California, Berkeley, July 1981, p. 158.
7. T. Teitelbaum et al., "The Why and Wherefore of the Cornell Program Synthesizer," Proc. ACM SIGPLAN-SIGOA Symp. on Text Manipulation, Portland, Oregon, June 1981, pp. 8-16.
8. S. R. Wood, "Z—The 95% Program Editor," Proc. ACM SIGPLAN-SIGOA Symp. on Text Manipulation, Portland, Oregon, June 1981, pp. 1-7.
9. R. C. Waters, "Program Editors Should Not Abandon Text Oriented Commands," ACM SIGPLAN Notices, 17, No. 7 (July 1982), pp. 39-46.
10. T. Reps, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors," Proc. 9th Annual Symp. on Principles of Programming Languages, Albuquerque, New Mexico, January 1982, pp. 169-76.
11. Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," Computer Science Technical Report #32, Bell Laboratories, Murray Hill, 1976.
12. E. R. Gansner et al., "Semantics and Correctness of a Query Language Translation," Ninth Symp. Principles of Programming Languages, Albuquerque, New Mexico, January 1982, pp. 289-98.
13. M. E. Lesk, "Lex—A Lexical Analyzer Generator," Computer Science Technical Report #39, Bell Laboratories, Murray Hill, 1975.
14. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, "Attributed Translations," J. Computer and System Sciences, 9 (December 1974), pp. 279-307.

APPENDIX A

An Editing Session With 'Hisel.age'

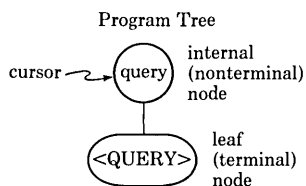
\$hisel.age

program? test

Initializing the tree/text for test..

Terminal Screen

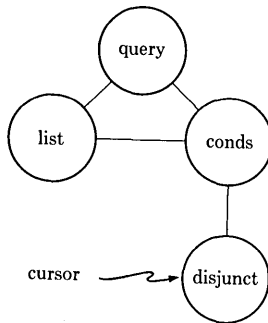
1) <QUERY>



```
:r <- user command - invisible
file name? temp
```

Assume that there is a file named 'temp' in the working directory and that it has the string "select <ITEM_LIST> where <CLAUSES>". AGE will read it, recognize the two nonterminal names, parse the string, create the subtree, and graft it to the root as shown below.

```
2) select <ITEM_LIST>
   where <CLAUSES>
```

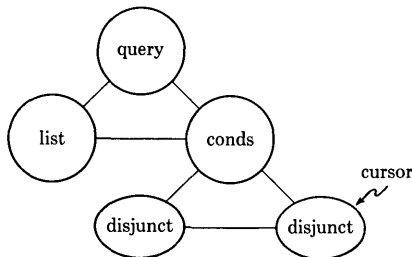


(Leaves are not shown)

```
:a
```

This will add one more instance of CLAUSES. The screen after this command will be:

```
3) select <ITEM_LIST>
   where <CLAUSES>
      or <CLAUSES>
```



```
:e
```

```
:e
```

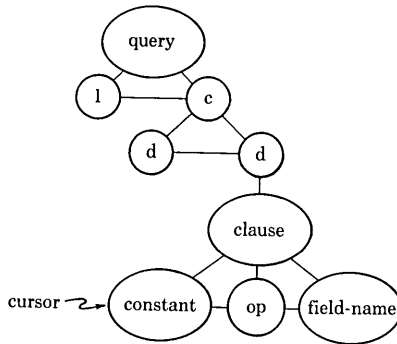
AGE will install a CLAUSE in place of <CLAUSES> after the first e command. Since there is a choice for the expansion corresponding to the next e command, the following menu items will appear one after

another if the user asks for a menu:

expansion character (type ? for menu): ? <- user types this
field_op-const 1 y(es), n(ext), q(uit)? n <- user types this
const-op-field 2 y(es), n(ext), q(uit)? y

The screen after this will be:

- 4) select <ITEM_LIST>
where <CLAUSES>
or constant OPERATOR field_name



:w

This command will write the above text into test.hisel file and the tree representation in test.lmd file.

:q

AGE will then print:

BYE! test hasn't been fully expanded; call me back later.

APPENDIX B

AGE Command Summary

Text-navigation commands:

- cr - a carriage return : go to the next line
- - the character minus: go back one line
- space-bar or n : go to the next word
- b : go back one word

Tree-navigation commands:

- ^ : go to the parent of the current nonterminal
- V : go to the first son of the current nonterminal
- > : go to the right neighboring nonterminal under the same parent
- < : go to the left neighboring nonterminal under the same parent
- N : go the next unexpanded nonterminal
- B : go backwards for the next unexpanded nonterminal

Nonterminal expansion commands:

p : parse text for the current nonterminal
r : read text from a file and parse it for the current nonterminal
e : expand the current nonterminal by menu selection
a : append an instance of a 'listy' (i.e., iterative) nonterminal
i : insert an instance of a 'listy' nonterminal
d : delete the current instance of the 'listy' nonterminal

Other commands:

U : unexpand nonterminal
w : write program
q : quit editing with AGE

AUTHORS

Beth A. Bottos, B.S.E.E., 1979, Purdue University; M.S. (Electrical Engineering), 1980, Stanford University; M.S. (Computer Science), 1983, Carnegie-Mellon University. From 1976 to 1979, Ms. Bottos was a participant in the Bell Laboratories Engineering Scholarship Program (BLESP). During that time, her work included telephone ringer characterization and modification and hardware design for a home appliance control system. From 1979 to 1981, she was involved in the development of the 1AVSS (Voice Storage System) and LADT (Local Area Data Transport) systems. Since 1981, Ms. Bottos has been a student in Computer Science at Carnegie-Mellon University. During that time, she has also worked for Bell Laboratories in the Advanced Software Department and in the Advanced Software Technology Laboratory.

Chandra M. R. Kintala, B.Sc. (Honors), 1970, Regional Engineering College, Rourkela, India; M.Tech. (Electrical Engineering), 1973, Indian Institute of Technology, Kanpur, India; Ph.D. (Computer Science), 1977, Pennsylvania State University; Assistant Professor, Computer Science Department, University of Southern California, 1977-1980; Bell Laboratories, 1980—. Mr. Kintala is a member of the Advanced Software Department. He has worked on the complexity of nondeterminism in various classes of Automata and Turing machines. His current interests include programming environments, compiler techniques for database query language translators and software graphics. Member, ACM, IEEE Computer Society, Sigma Xi, Phi Kappa Phi, and Who's Who in the East.

Performance Analysis of a Preemptive Priority Queue with Applications to Packet Communication Systems

By M. G. HLUCHYJ,* C. D. TSAO,* and R. R. BOORSTYN†

(Manuscript received May 6, 1983)

In this paper we analyze the performance of a preemptive priority queue. We give the model description in the context of a packet communication system where message sources, having different priorities, share a common communication channel. Each source generates, as an independent Poisson process, messages consisting of an arbitrarily distributed, random number of fixed-length packets. The channel server can only begin service at integer multiples of the packet transmission time (i.e., a time-slotted channel is assumed), and the server will preempt an ongoing message transmission at the next packet boundary whenever there is a message arrival from a higher-priority source. The average in-queue waiting time for each packet in any given source message and the average message delay are derived along with the corresponding moment-generating functions. Also, comparisons are made with the first-come first-served queueing discipline.

I. INTRODUCTION

We analyze the performance of a preemptive priority queueing system. To make clear at the outset the importance of the particular queueing system studied, we describe the system model in a packet communication context. Specifically, as Fig. 1 illustrates, a number of data sources share a single communication channel. Each source generates, according to a Poisson process, messages consisting of a

*AT&T Information Systems. †Polytechnic Institute of New York.

©Copyright 1983, American Telephone & Telegraph Company. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

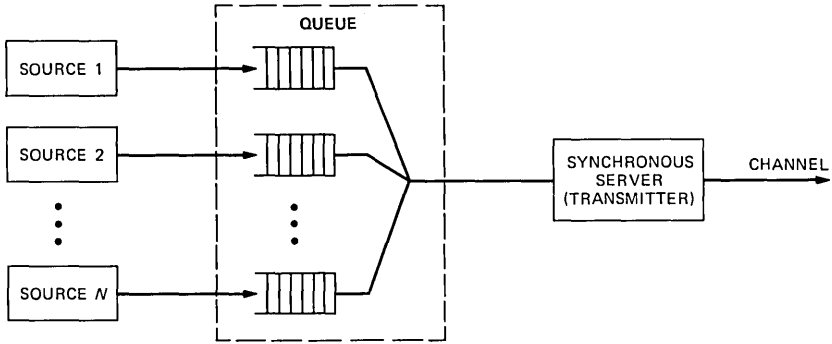


Fig. 1—Queueing model for a packet communication system.

random number of fixed-length data packets. The packets comprising a message arrive in bulk to be transmitted on the communication channel. For clarity, we view each source as having its own separate buffer to queue packets. Here, packets generated by the source wait for access to the channel.

Packet transmissions on the channel are synchronized. More precisely, time is divided into a sequence of fixed-length intervals or time slots. Each time slot is just large enough to allow the transmission of one packet, and packet transmissions must occur within time-slot boundaries. Hence, a packet arriving at the queue, at the very least, must wait until the start of the next time slot before its transmission can begin.

Packets from any given source are served (i.e., transmitted) on a first-come first-serve basis. The sources, however, are assigned fixed priorities: the first source has the highest priority, the last has the lowest. At the start of each time slot, the first packet queued from the highest-priority source is served. That is, a packet at the head of the source k buffer is transmitted if and only if the buffers associated with sources 1 to $k - 1$ are empty. Hence, an ongoing packet transmission cannot be preempted; however, an ongoing message transmission will be preempted (at the next slot boundary) whenever there is a message arrival from a higher-priority source.

Such a priority queueing discipline arises naturally in many packet communication systems. The channel might be a link in a data communication network, or may simply be a shared data bus. The use of priority may be required to give more urgent messages lower delay. For example, one might choose to give network control messages higher priority than interactive data messages, which in turn are given higher priority than long file transfers. In some situations, the priority structure is inherent in the mechanism for sharing the channel among the independent messages sources. This is the case with Datakit,¹

where the module (i.e., the interface between the source and the channel) with the highest address always wins the channel contention. It is also true of some slotted ring systems, where the physical order of sources along the ring imposes a priority ordering for access to the channel.²

The first results on queues with preemptive priority appear to be due to White and Christie.³ Shortly thereafter, others studied the problem using different assumptions about the service time distribution. A comprehensive treatment of some of the early work is given in Jaiswal,⁴ and a more up-to-date, but less comprehensive, discussion may be found in Kleinrock.⁵ The models examined, however, all assume an "asynchronous" server where service starting times and preemption times are not constrained to certain periodically recurring points. The use of a synchronous service facility in queueing models arises in the context of computer and data communication systems where there is a natural elementary unit of time such as the machine cycle of a processor, or the bit, byte, or packet transmission time on a channel. Many such models are reviewed, and references given, in Kobayashi and Konheim.⁶ As we indicated, the model we have selected for study has applications to slotted ring systems, and it is here that one finds analysis of other models similar to ours. The model that seems to come closest is by Konheim and Meister,² where the main differences have to do with the arrival process. Konheim and Meister assume discrete arrivals (between slots) of packets, whereas we assume continuous arrivals of messages with each message containing an arbitrarily distributed number of packets. In this way, we are better able to examine message delays in the system.

In this paper we analyze the performance of the above preemptive priority queueing system. We begin in Section II by summarizing the queueing model and introducing performance measures that are of interest. In Section III we derive the average in-queue waiting time for each packet in any given source message. From this result we easily obtain the average delay in transporting a message. The corresponding moment-generating functions are derived in the appendix. Finally, in Section IV, we compare performance with the First-Come First-Served (FCFS) queueing discipline.

II. QUEUEING MODEL

In this section we briefly summarize the important points of the queueing model, and indicate the steady-state statistics that are of interest. Notation established here is used in the performance analysis that follows.

The queueing system under study has the following properties:

1. N sources of messages.

2. Priorities are assigned to sources in decreasing order (i.e., source k has higher priority than source $k + 1$, $k = 1, 2, \dots, N - 1$).

3. Source k generates messages as an independent Poisson process with rate λ_k messages per time slot. Each such message has its length (in packets) selected independently from the distribution $P_{m_k}(\cdot)$ with first and second moments, \bar{m}_k and m_k^2 , respectively.

4. During busy periods, one packet is transmitted in each time slot and is always selected at the beginning of the time slot from the head of the highest-priority, nonempty source buffer.

5. Each source buffer is assumed infinite, and packets enter and are removed from the buffer on a first-in first-out basis.

We define W_{kj} as the steady-state in-queue waiting time for the j th packet in a message from source k , $k = 1, 2, \dots, N$. In addition, we define

$$\rho_k = \lambda_k \bar{m}_k,$$

where ρ_k is interpreted as the fraction of time the server is busy with source k packets. We also find it convenient to define

$$\sigma_k = \sum_{i=1}^k \rho_i.$$

Other notation is introduced as needed in the analysis.

III. PERFORMANCE ANALYSIS

We begin this section by deriving \bar{W}_{kj} , the average in-queue waiting time for the j th packet in a message from source k . Using this result we then obtain the average delay in transporting a message from source k . Included in the discussion are specific numerical examples to illustrate the derived results.

3.1 Average waiting time analysis

In Fig. 2, observe that we may express the waiting time for the j th packet in a source k message as

$$W_{kj} = W_{k1} + \sum_{\ell=1}^{j-1} w_{k\ell}, \quad (1)$$

where the incremental waiting time $w_{k\ell}$ is defined by

$$w_{k\ell} = W_{k,\ell+1} - W_{k\ell}.$$

For a given message length, m_k , the random variables $\{w_{k1}, w_{k2}, \dots, w_{k,m_k-1}\}$ are independent and identically distributed. We observe that at the beginning of a slot during which a packet from source k is in service, there are no packets from sources 1 to $k - 1$ in the system.

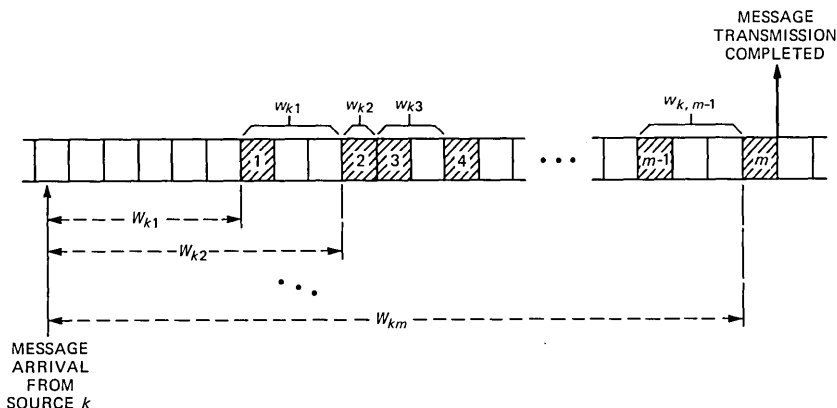


Fig. 2—Waiting times for packet transmissions.

Any messages that arrive from sources 1 to $k - 1$ while this source k packet is in service spawn a busy period of service, starting in the next slot, for sources 1 to $k - 1$. All such busy periods are independent and identically distributed, and hence so are the random variables $\{w_{k1}, w_{k2}, \dots, w_{k, m_k - 1}\}$.

Note that the incremental waiting time $w_{k\ell}$ consists of one slot time to transmit the ℓ th packet in the source k message plus the time to serve all messages from sources 1 to $k - 1$ that arrive in the interval $w_{k\ell}$. Hence, the average incremental waiting time $\bar{w}_{k\ell}$ satisfies

$$\bar{w}_{k\ell} = 1 + \sigma_{k-1} \bar{w}_{k\ell},$$

from which we obtain

$$\bar{w}_{k\ell} = \frac{1}{1 - \sigma_{k-1}}.$$

It then follows from (1) that the average in-queue waiting time for the j th packet in a source k message is given by

$$\bar{W}_{kj} = \bar{W}_{k1} + \frac{j-1}{1 - \sigma_{k-1}}. \quad (2)$$

Hence we are left with having to determine \bar{W}_{k1} , the average waiting time for the first packet in the message.

By applying standard queueing arguments, we have

$$\bar{W}_{k1} = \frac{1}{2} + \sum_{i=1}^k \sum_{j=1}^{\infty} \rho_{ij} \bar{W}_{ij} + \sum_{i=1}^{k-1} \rho_i \bar{W}_{k1}, \quad (3)$$

where

$$\rho_{ij} \triangleq \lambda_i \Pr\{m_i \geq j\}. \quad (4)$$

The first term on the right-hand side of (3) is simply the average time between the arrival of a message and the start of the next slot. The second term is, by Little's result, the average number of packets of equal or higher priority awaiting transmission at the moment the message arrives. Finally, the last term corresponds to the average number of packets of higher priority that arrive while the first packet in the source k message waits on queue.

Now substituting (2) into (3) yields

$$\bar{W}_{k1} = \frac{1}{2} + \sum_{i=1}^k \sum_{j=1}^{\infty} \rho_{ij} \left(\bar{W}_{i1} + \frac{j-1}{1-\sigma_{i-1}} \right) + \sum_{i=1}^{k-1} \rho_i \bar{W}_{k1}. \quad (5)$$

Note from the definition of ρ_{ij} in (4) that

$$\begin{aligned} \sum_{j=1}^{\infty} \rho_{ij} &= \lambda_i \sum_{j=1}^{\infty} \Pr[m_i \geq j] = \lambda_i \sum_{j=1}^{\infty} \sum_{\ell'=j}^{\infty} \Pr[m_i = \ell'] \\ &= \lambda_i \sum_{\ell'=1}^{\infty} \sum_{j=1}^{\ell'} \Pr[m_i = \ell'] = \lambda_i \sum_{\ell'=1}^{\infty} \ell' \Pr[m_i = \ell'] \\ &= \lambda_i \bar{m}_i = \rho_i. \end{aligned} \quad (6)$$

Similarly, we have that

$$\sum_{j=1}^{\infty} (j-1) \rho_{ij} = \frac{\lambda_i}{2} (\bar{m}_i^2 - \bar{m}_i). \quad (7)$$

Hence, using (6) and (7), we may rewrite (5) as

$$\bar{W}_{k1} = \frac{\frac{1}{2} + \sum_{i=1}^{k-1} \rho_i \bar{W}_{i1} + \sum_{i=1}^k \lambda_i (\bar{m}_i^2 - \bar{m}_i) / 2(1 - \sigma_{i-1})}{(1 - \sigma_k)}.$$

Solving recursively, we obtain

$$\bar{W}_{k1} = \frac{1 + \sum_{i=1}^k \lambda_i (\bar{m}_i^2 - \bar{m}_i)}{2(1 - \sigma_k)(1 - \sigma_{k-1})}. \quad (8)$$

Finally, substituting (8) into (2) yields

$$\bar{W}_{kj} = \frac{1 + \sum_{i=1}^k \lambda_i (\bar{m}_i^2 - \bar{m}_i)}{2(1 - \sigma_k)(1 - \sigma_{k-1})} + \frac{j-1}{1 - \sigma_{k-1}}. \quad (9)$$

This concludes the derivation of the average in-queue waiting time \bar{W}_{kj} . The derivation of the moment-generating function for W_{kj} (from which \bar{W}_{kj} can be obtained directly) is given in the appendix.

To illustrate the performance, we begin by considering a homoge-

neous system where $\lambda_k = \lambda$, $\bar{m}_k = \bar{m}$, and $\bar{m}_k^2 = \bar{m}^2$ for $k = 1, 2, \dots, N$. For this case, (8) becomes

$$\bar{W}_{k1} = \frac{1 + \frac{k}{N} \rho (\bar{m}^2/\bar{m} - 1)}{2 \left(1 - \frac{k}{N} \rho\right) \left(1 - \frac{(k-1)}{N} \rho\right)}, \quad (10)$$

where ρ is the total system utilization (or load) defined by

$$\begin{aligned} \rho &= \sum_{i=1}^N \rho_i \\ &= N\lambda\bar{m} \quad (\text{for a homogeneous system}). \end{aligned}$$

If we take $N = 10$ and assume a constant message length of 10 packets (i.e., $\bar{m} = 10$, $\bar{m}^2 = 100$), Fig. 3 is a plot of \bar{W}_{k1} vs. ρ for k varying from

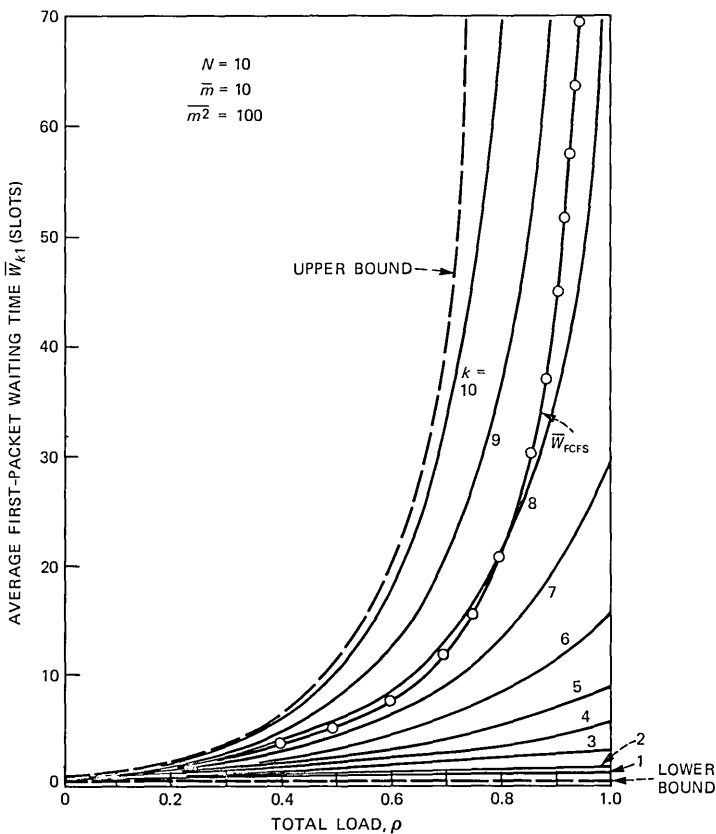


Fig. 3—Average first packet waiting time \bar{W}_{k1} vs. total load ρ .

1 to 10. Also shown in Fig. 3 is the average waiting time for the first-come first-served (FCFS) queueing discipline, which is derived in Section IV. Note from (10) that if we allow $N \rightarrow \infty$, then

$$\overline{W}_{11} \rightarrow \frac{1}{2}$$

$$\overline{W}_{N1} \rightarrow \frac{1 + \rho(\overline{m^2}/\overline{m} - 1)}{2(1 - \rho)^2}.$$

These two expressions represent, respectively, lower and upper bounds on the average first packet waiting time for all sources and arbitrary N . These bounds are plotted as dashed lines in Fig. 3. Finally, if we assume the same values for N , \overline{m} , and $\overline{m^2}$ as in Fig. 3, Fig. 4 is a plot of the average incremental waiting time \overline{w}_{k1} vs. ρ for k varying from 1 to 10. Also shown in Fig. 4 is the upper bound $1/(1 - \rho)$ on \overline{w}_{k1} , valid for all parameter values.

3.2 Average message delay analysis

We now consider the average message delay. Defining $\overline{D}_k(m)$ as the average delay (in slots) from the arrival to the queue of an m -packet

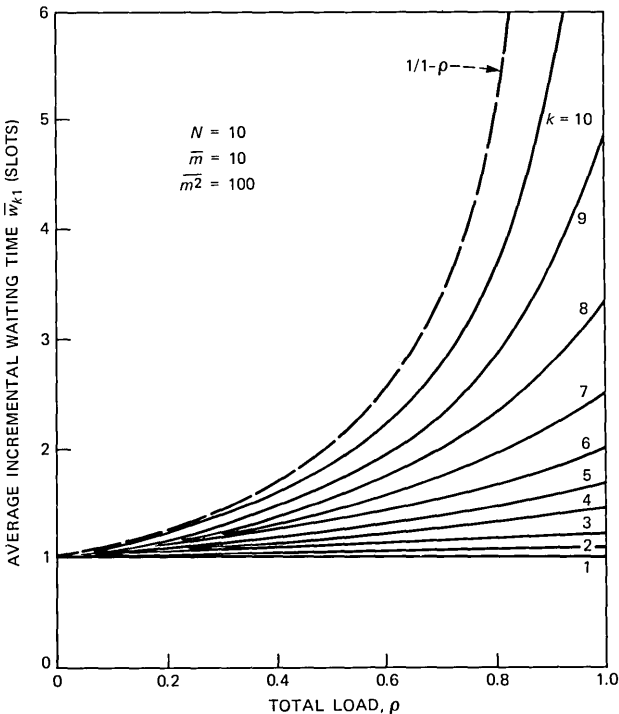


Fig. 4—Average incremental waiting time \overline{w}_{k1} vs. total load ρ .

message from source k until the end of its transmission, we have

$$\bar{D}_k(m) = \bar{W}_{km} + 1.$$

Letting \bar{D}_k denote the average delay over all messages from source k , it follows, since \bar{W}_{km} is linear in m , that

$$\begin{aligned} \bar{D}_k &= \sum_m \bar{D}_k(m) P_{m_k}(m) \\ &= \bar{W}_{k, \bar{m}_k} + 1. \end{aligned} \quad (11)$$

If we assume the same homogeneous system as represented in Figs. 3 and 4, Fig. 5 is a plot of \bar{D}_k vs. ρ for k varying from 1 to 10. Also shown in Fig. 5 is an upper bound on \bar{D}_k , obtained from the upper bounds on \bar{W}_{k1} and $\bar{w}_{k'}$. Specifically, we have

$$\bar{D}_k \leq \frac{1 + \rho[\bar{m}^2/\bar{m} - 1]}{2(1 - \rho)^2} + \frac{\bar{m} - 1}{(1 - \rho)} + 1,$$

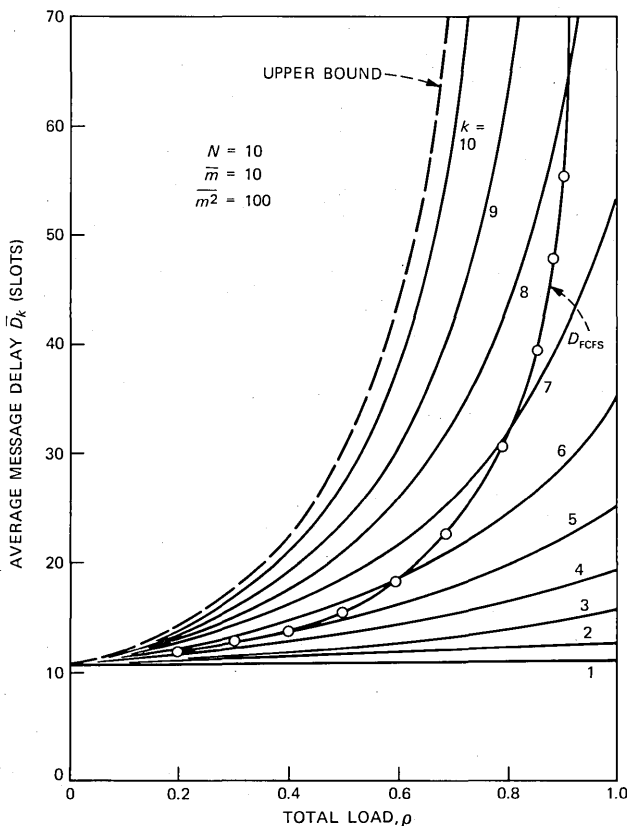


Fig. 5—Average message delay \bar{D}_k vs. total load ρ .

which depends on \bar{m} and \bar{m}^2 , but is valid for all sources and arbitrary N .

To complete this section, we consider a nonhomogeneous system consisting of 10 host computers and 300 terminals. The terminals and hosts correspond to the message sources and may be viewed as sharing a common time-slotted bus. There is a priority ordering of the terminals and hosts, with terminals having priority over hosts (i.e., the terminals correspond to sources 1 to 300 and the hosts correspond to sources 301 to 310). Each host is assumed to generate two types of traffic: host-to-host file transfers consisting of fixed-length 32-packet messages, and host-to-terminal messages with an average message length of 2 packets and a standard deviation of 1. Each terminal, on the other hand, only generates messages that are one packet in length and destined to a host. The message generation rates for each of the two types of host traffic are the same for all hosts. Similarly, all terminals generate messages at the same rate. The specific generation rate of each traffic type is such that the total load on the channel is divided as follows: 30 percent host-to-host, 60 percent host-to-terminal, and 10 percent terminal-to-host. The average delay performance for this system is plotted in Fig. 6. Observe that the results obtained allow us to distinguish between different types of traffic generated by the same source. In particular, in Fig. 6, the average message delay performance for the host-to-host and host-to-terminal traffic are shown separately.

From the moment-generating function for D_k derived in the appendix, one can obtain the second moment of the message delay. This, in turn, may be used to compute the message delay standard deviation. For hosts 1 and 10 (i.e., the two extremes), shown in Fig. 7 for the host-to-host messages and in Fig. 8 for the host-to-terminal messages, we see the mean delay and mean delay plus one, two, and three standard deviations (denoted by 1σ , 2σ , and 3σ). The second-moment-of-message delay depends on the first three moments of message length, and in Fig. 8 we set $\bar{m}^3 = 15$.

IV. COMPARISONS WITH FCFS

In this section we compare the average delay performance of the priority queueing discipline studied in the previous section with that of the First-Come First-Served (FCFS) discipline. With the FCFS discipline, messages are served in the order in which they are generated, independent of the source from which they originate. In this way, the FCFS discipline allocates the communication channel more fairly than does the priority discipline. For simplicity, we assume in the analysis a homogeneous system where $\lambda_k = \lambda$, $\bar{m}_k = \bar{m}$, and $\bar{m}_k^2 = \bar{m}^2$ for $k = 1, 2, \dots, N$.

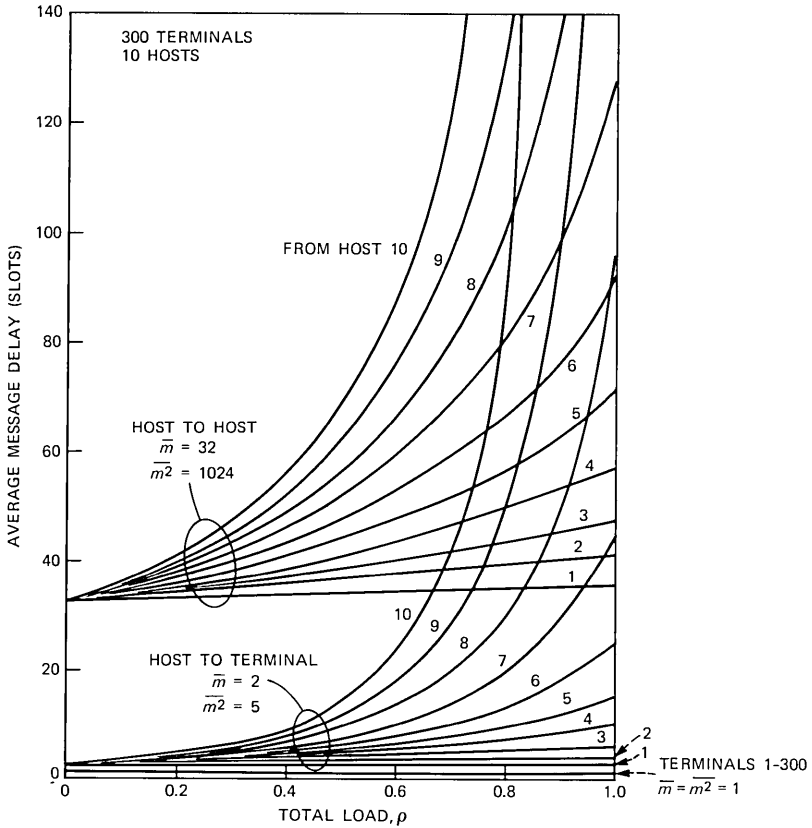


Fig. 6—Average message delay vs. total load ρ .

The performance analysis of the FCFS queueing discipline is a special case of the results obtained for the priority discipline. Specifically, we combine the N independent Poisson streams into a single Poisson stream (using the well-known result that the sum of independent Poisson processes is a Poisson process) with rate $N\lambda$. From (10) we have that the average in-queue waiting time for a message generated by this combined (single) source is given by

$$\bar{W}_{\text{FCFS}} = \frac{\rho(\bar{m}^2/\bar{m})}{2(1-\rho)} + \frac{1}{2}, \quad (12)$$

where again $\rho = N\lambda\bar{m}$ is the total system utilization. The average message delay for the FCFS system is given by

$$\begin{aligned} \bar{D}_{\text{FCFS}} &= \bar{W}_{\text{FCFS}} + \bar{m} \\ &= \frac{\rho(\bar{m}^2/\bar{m})}{2(1-\rho)} + \frac{1}{2} + \bar{m}. \end{aligned} \quad (13)$$

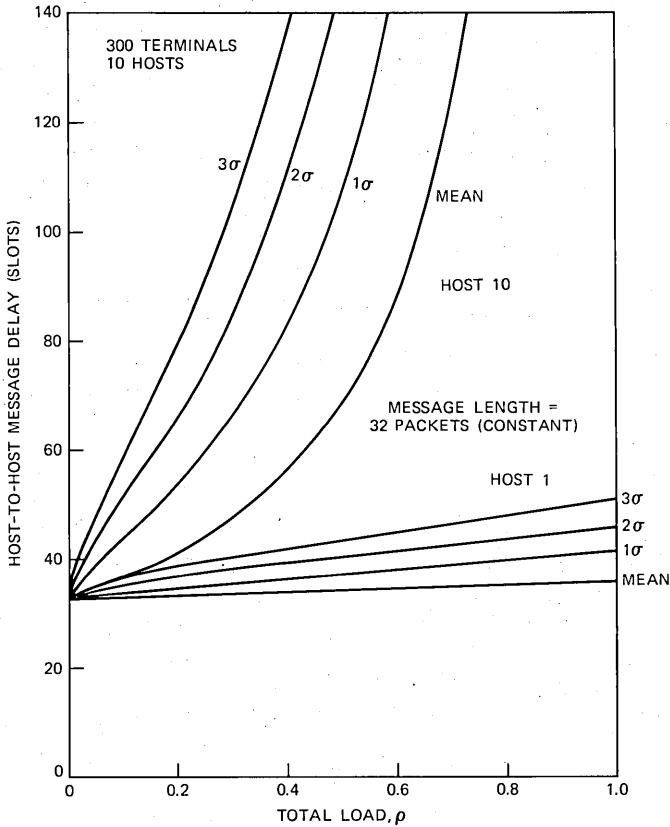


Fig. 7—Host-to-host message delay vs. total load ρ .

\bar{W}_{FCFS} is plotted in Fig. 3 and \bar{D}_{FCFS} is plotted in Fig. 5 for the assumed system parameter values.

It is worth noting that the waiting time and delay results given by (12) and (13), respectively, differ from those corresponding to the standard M/G/1 queueing system by the additional term $1/2$. This added term results from the synchronous nature of the server and represents the average time an arriving message must wait before the start of the next time slot.

We continue the priority and FCFS comparison by focusing on the unfairness issue. Specifically, we consider the ratio of the average message delay for source N to that of source 1, \bar{D}_N/\bar{D}_1 . Since all sources encounter the same average delay in the FCFS discipline, $\bar{D}_N/\bar{D}_1 = 1$. With the priority discipline, source N has the lowest priority and source 1 the highest; hence $\bar{D}_N/\bar{D}_1 > 1$ for $\rho > 0$. In particular, we have from (9) and (11) that

$$\bar{D}_k = \frac{1 + \frac{k}{N} \rho [(1 + c_m^2) \bar{m} - 1]}{2 \left(1 - \frac{k}{N} \rho\right) \left(1 - \frac{(k-1)}{N} \rho\right)} + \frac{\bar{m} - 1}{\left(1 - \frac{(k-1)}{N} \rho\right)} + 1, \quad (14)$$

where c_m^2 is the squared coefficient of variation for the message-length distribution defined by

$$c_m^2 = \frac{\text{variance}(m)}{(\bar{m})^2}.$$

Hence, for large N we have from (14) that

$$\begin{aligned} \bar{D}_1 &= \frac{\frac{\rho}{N} (1 + c_m^2) \bar{m}}{2 \left(1 - \frac{\rho}{N}\right)} + \frac{1}{2} + \bar{m} \\ &\approx \frac{\bar{m}}{2} \left[\frac{1}{\bar{m}} + \frac{\rho}{N} (1 + c_m^2) + 2 \right] \end{aligned}$$

and

$$\begin{aligned} \bar{D}_N &= \frac{1 + \rho [(1 + c_m^2) \bar{m} - 1]}{2(1 - \rho) \left(1 - \frac{(N-1)}{N} \rho\right)} + \frac{\bar{m} - 1}{\left(1 - \frac{(N-1)}{N} \rho\right)} + 1 \\ &= \frac{\bar{m} \left[(1 + c_m^2) \rho + \left(2 - \frac{1}{\bar{m}}\right) (1 - \rho) + \frac{2}{\bar{m}} (1 - \rho)^2 \right]}{2(1 - \rho)^2}. \end{aligned}$$

It follows then that

$$\frac{\bar{D}_N}{\bar{D}_1} \approx \begin{cases} \frac{1 + c_m^2 \rho + 2(1 - \rho)^2}{3(1 - \rho)^2} & \text{for } \bar{m} = 1 \\ \frac{2 + (c_m^2 - 1)\rho}{2(1 - \rho)^2} & \text{for } \bar{m} \gg 1. \end{cases}$$

Observe that for large N and fixed ρ , the increase in \bar{D}_N/\bar{D}_1 is approximately linear with the squared coefficient of variation c_m^2 . In Fig. 9, the ratio \bar{D}_N/\bar{D}_1 is plotted against total utilization ρ for the FCFS and priority disciplines with $c_m^2 = 0$ and 1.

To complete this section, we compare the average delay performance of the FCFS discipline with the overall average delay of the priority discipline. That is, we compare \bar{D}_{FCFS} as given by (13) to the quantity

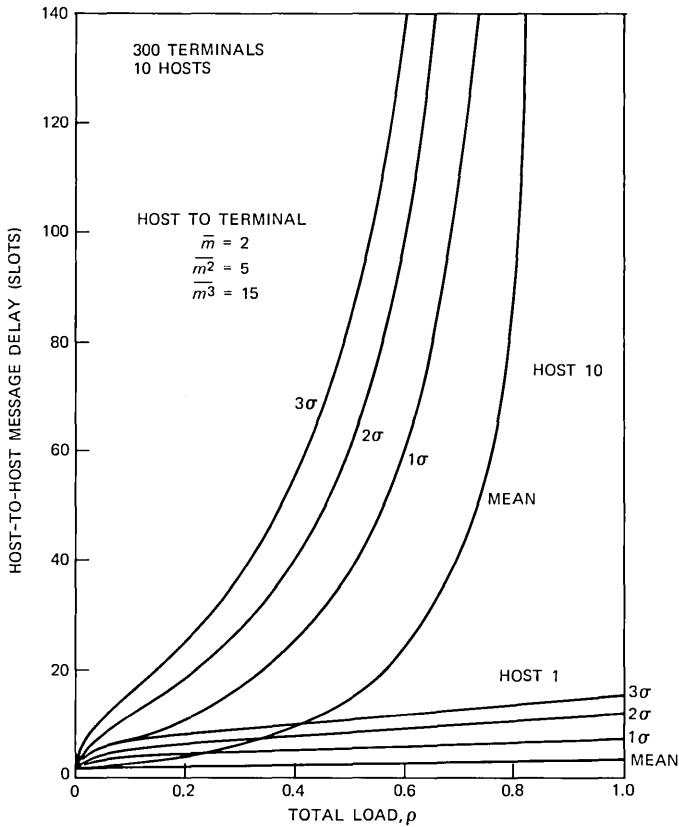


Fig. 8—Host-to-terminal message delay vs. total load ρ .

$$\bar{D} \triangleq \frac{1}{N} \sum_{k=1}^N \bar{D}_k.$$

Using the expression for \bar{D}_k given in (14), we obtain after some manipulation

$$\bar{D} = \frac{(1 + c_m^2)\bar{m}}{2(1 - \rho)} + \frac{1}{2N} [2\bar{m} - 1 - (1 + c_m^2)\bar{m}]\gamma + 1,$$

where

$$\gamma = \sum_{k=1}^N \left[1 - \frac{(k-1)}{N} \rho \right]^{-1}.$$

From this, one may show that

$$\bar{D}_{\text{FCFS}} \leq \bar{D} \quad \text{for} \quad 0 \leq c_m^2 \leq \frac{\bar{m} - 1}{\bar{m}}$$

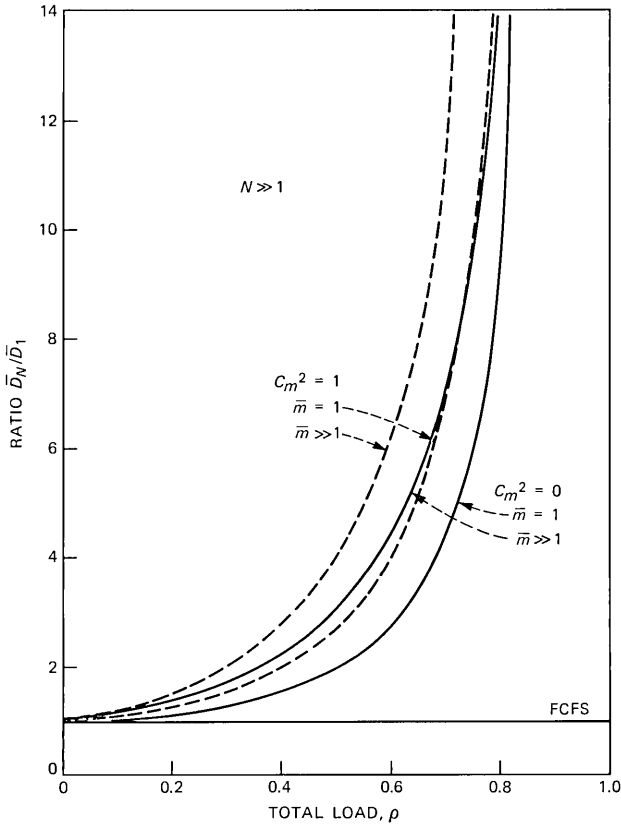


Fig. 9—Ratio \bar{D}_N/\bar{D}_1 vs. total load ρ .

and

$$\bar{D}_{\text{FCFS}} \geq \bar{D} \quad \text{for} \quad c_m^2 \geq \frac{\bar{m} - 1}{\bar{m}}.$$

Hence, for sufficiently large message-length coefficient of variation c_m^2 , the overall average delay for the priority discipline is less than the average delay for the first-come first-served discipline. Of course, as we saw earlier, as c_m^2 increases so does the relative unfairness of the priority discipline over the FCFS discipline.

V. CONCLUSIONS

We analyzed the performance of a preemptive priority queue, which has direct applications to packet communication systems. The main distinguishing feature of the system studied compared to the standard M/G/1 preemptive resume priority queue⁵ is that the server can only

begin serving a "customer" (and preemptions take place) at integer multiples of time corresponding to packet slot boundaries in the communication context. Mean value formulas for in-queueing waiting time and average message delay were derived and comparisons made to the FCFS queueing discipline. A derivation of the waiting time and delay moment-generating functions is given in the appendix.

REFERENCES

1. A. G. Fraser, "Datakit—A Modular Network for Synchronous and Asynchronous Traffic," Proc. ICC (June 1979), pp. 20.1.1-3.
2. A. G. Konheim and B. Meister, "Service in a Loop System," JACM, 19, No. 1 (January 1972), pp. 92-108.
3. H. White and L. S. Christie, "Queueing with Preemptive Priorities or with Break-down," Oper. Res., 6, No. 1 (January-February 1958), pp. 79-95.
4. N. K. Jaiswal, *Priority Queues*, New York, NY: Academic Press, 1968.
5. L. Kleinrock, *Queueing Systems, Vol. II: Computer Applications*, New York, NY: John Wiley and Sons, 1976.
6. H. Kobayashi and A. G. Konheim, "Queueing Models for Computer Communications System Analysis," IEEE Trans. Commun., COM-25, No. 1 (January 1977), pp. 2-29.

APPENDIX

Derivation of the Waiting Time and Message Delay Moment-Generating Functions

As we introduced in Section II, W_{kj} is the steady-state in-queue waiting time for the j th packet in a message from source k , $k = 1, 2, \dots, N$. Its Moment-Generating Function (MGF), defined as

$$G_{W_{kj}}(\nu) = E[e^{\nu W_{kj}}]$$

is derived in this appendix. From this result, the message delay MGF is easily obtained. The approach taken parallels in many respects the analysis given in Section III.

We begin the derivation by examining the duration of a busy period for sources 1 to $k - 1$, denoted by Y_k . Such a busy period starting in a slot is initialized by one or more message arrivals from sources 1 to $k - 1$ in the previous slot (which contains no packet from sources 1 to $k - 1$). Let A_k denote the total number of packets that arrive from sources 1 to $k - 1$ in this previous slot. For the i th packet in this set, we define the "sub-busy" period $X_k(i)$ to consist of the duration of the "virtual" busy period (i.e., as if $i = A_k = 1$) initiated by the messages (if any) that arrive from sources 1 to $k - 1$ while this i th packet is in service. In other words, we conceptually reorder the priorities so that each of the A_k packets, and the sub-busy period it spawns, is served in turn. This does not change Y_k and is a standard approach to busy-period analysis.

Due to the memoryless property of the arrival process, the sub-busy period random variables $X_k(i)$, $i = 1, 2, \dots, A_k$, are independent and

identically distributed (*iid*). In addition, note that Y_k has the same distribution as the generic random variable X_k , and satisfies the relation

$$Y_k = A_k + \sum_{i=1}^{A_k} X_k(i). \quad (15)$$

The Probability-Generating Function (PGF) for the discrete random variable X_k is defined as

$$\Phi_{X_k}(z) = E[z^{X_k}] = \sum_{i=0}^{\infty} z^i \Pr[X_k = i].$$

Using (15) and the result that X_k is distributed as Y_k , we obtain

$$\begin{aligned} \Phi_{X_k}(z) &= E[z^{Y_k}] \\ &= E \left[z^{A_k + \sum_{i=1}^{A_k} X_k(i)} \right] \\ &= E[(z\Phi_{X_k}(z))^{A_k}] \\ &= \Phi_{A_k}(z\Phi_{X_k}(z)), \end{aligned} \quad (16)$$

where $\Phi_{A_k}(z)$ is the PGF for the random variable A_k .

Now, A_k is equal to the total number of packets arriving from sources 1 to $k - 1$ in one time slot. Recall that each source i generates messages as an independent Poisson process with rate λ_i ; and each such message has its length selected independently from the distribution $P_{m_i}(\cdot)$, whose PGF we denote by $\Phi_{m_i}(z)$. It follows then that

$$\begin{aligned} \Phi_{A_k}(z) &= \prod_{i=1}^{k-1} \left\{ \sum_{r=0}^{\infty} \frac{\lambda_i^r e^{-\lambda_i}}{r!} \cdot [\Phi_{m_i}(z)]^r \right\} \\ &= \prod_{i=1}^{k-1} e^{\lambda_i[\Phi_{m_i}(z)-1]}. \end{aligned} \quad (17)$$

Hence, substituting (17) into (16), we obtain

$$\Phi_{X_k}(z) = \prod_{i=1}^{k-1} e^{\lambda_i[\Phi_{m_i}(z\Phi_{X_k}(z))-1]}. \quad (18)$$

As we shall see, $\Phi_{X_k}(z)$, the PGF for the duration of a busy period for sources 1 to $k - 1$, plays an important role in the derivation of $G_{W_{kj}}(\nu)$, the waiting time MGF.

Returning to eq. (1) in Section III, we note that W_{kj} is the sum of W_{k1} and the $j - 1$ *iid* random variables $w_{k1}, w_{k2}, \dots, w_{kj-1}$. Observe, however, that $w_{k\ell}$, $\ell = 1, 2, \dots, j - 1$, is distributed as $X_k + 1$. That is, $w_{k\ell}$ is composed of the service time for the ℓ th packet plus the busy

period for sources 1 to $k - 1$ initiated during this service time. In addition, it follows that the waiting time for the first packet in a source k message, W_{k1} , is statistically independent of $w_{k\ell}$, $\ell = 1, 2, \dots, j - 1$. Hence we may write

$$G_{W_{kj}}(\nu) = G_{W_{k1}}(\nu) \cdot [e^\nu \Phi_{X_k}(e^\nu)]^{j-1}. \quad (19)$$

This leaves us with having to determine the MGF for W_{k1} .

Let us for the moment consider the time-dependent behavior for the number of packets queued from sources 1 to k . For time slot n , we let $Q_k(n)$ denote the number of such packets queued just after the beginning of the slot and, to be consistent with our previous notation, we let $A_{k+1}(n)$ denote the number of packets that arrive from sources 1 to k during the n th slot. It follows that

$$Q_k(n + 1) = [Q_k(n) + A_{k+1}(n) - 1]^+, \quad (20)$$

where

$$[\epsilon]^+ = \begin{cases} \epsilon & \text{if } \epsilon \geq 0 \\ 0 & \text{if } \epsilon < 0. \end{cases}$$

From (20) we obtain the relation

$$E[z^{Q_k(n+1)}] = E[z^{[Q_k(n) + A_{k+1}(n) - 1]^+}], \quad (21)$$

which may be rewritten as

$$\begin{aligned} \Phi_{Q_k(n+1)}(z) &= E[z^{[Q_k(n) + A_{k+1}(n) - 1]^+}] \\ &= \Pr[Q_k(n) + A_{k+1} = 0] + \Pr[Q_k(n) + A_{k+1} > 0] \\ &\quad \cdot z^{-1} E[z^{Q_k(n) + A_{k+1}} | Q_k(n) + A_{k+1} > 0] \\ &= \Pr[Q_k(n) = 0] \Pr[A_{k+1} = 0] \\ &\quad + z^{-1} \sum_{i=1}^{\infty} z^i \Pr[Q_k(n) + A_{k+1} = i] \\ &= \Pr[Q_k(n) = 0] \Pr[A_{k+1} = 0] \\ &\quad + z^{-1} \{ E[z^{Q_k(n) + A_{k+1}}] - \Pr[Q_k(n) = 0] \Pr[A_{k+1} = 0] \} \\ &= \Pr[Q_k(n) = 0] \Pr[A_{k+1} = 0] (1 - z^{-1}) \\ &\quad + z^{-1} \Phi_{Q_k(n)}(z) \Phi_{A_{k+1}}(z), \end{aligned} \quad (22)$$

where we have used the fact that $Q_k(n)$ and A_{k+1} are statistically independent. Taking the limit as $n \rightarrow \infty$ on both sides of (22) (the limits exist for $\sigma_k < 1$) yields

$$\Phi_{Q_k}(z) = \Pr[Q_k = 0] \Pr[A_{k+1} = 0] (1 - z^{-1}) + z^{-1} \Phi_{Q_k}(z) \Phi_{A_{k+1}}(z), \quad (23)$$

where Q_k represents the steady-state number of packets queued from sources 1 to k at the beginning of a slot. Rearranging the terms in (23), we obtain

$$\Phi_{Q_k}(z) = \frac{\Pr[Q_k = 0]\Pr[A_{k+1} = 0](z - 1)}{z - \Phi_{A_{k+1}}(z)}. \quad (24)$$

Taking the limit as $z \rightarrow 1$ on both sides of (24) yields

$$\begin{aligned} \Pr[Q_k = 0]\Pr[A_{k+1} = 0] &= 1 - \left. \frac{\partial}{\partial z} \Phi_{A_{k+1}}(z) \right|_{z=1} \\ &= 1 - \sigma_k. \end{aligned}$$

Hence, using this result and (17), (24) may be rewritten as

$$\Phi_{Q_k}(z) = \frac{(1 - \sigma_k)(z - 1)}{z - \prod_{i=1}^k e^{\lambda_i(\Phi_{m_i}(z) - 1)}}. \quad (25)$$

Now consider the end of the time slot during which a source k message is generated. The number of packets of higher or equal priority that are queued and must be transmitted before the first packet in this source k message is given by

$$Q_k + A_k + B_k,$$

where Q_k is the number of queued packets from sources 1 to k just after the beginning of the slot, A_k is the number of packets from sources 1 to $k - 1$ that arrive during the slot, and the new random variable, B_k , represents the number of packets from source k that arrive during the slot prior to the generation of the source k message in question. The i th packet in this set of $(Q_k + A_k + B_k)$ packets initiates a sub-busy period of duration $X_k(i)$. Hence we may write

$$W_{k1} = U + \sum_{i=0}^{(Q_k + A_k + B_k)} [1 + X_k(i)], \quad (26)$$

where U is a random variable, uniformly distributed over one slot time, that represents the time from when the source k message is generated until the start of the next slot.

From (26) we may write

$$E[e^{\nu W_{k1}} | U = u, Q_k = q_k, A_k = a_k, B_k = b_k] = e^{\nu u} [e^{\nu \Phi_{X_k}}(e^{\nu})]^{(q_k + a_k + b_k)}.$$

Removing the conditioning on the independent random variables Q_k and A_k yields

$$\begin{aligned} E[\alpha^{W_{k1}} | U = u, B_k = b_k] \\ = \alpha^u [\alpha \Phi_{X_k}(\alpha)]^{b_k} \Phi_{Q_k}(\alpha \Phi_{X_k}(\alpha)) \Phi_{A_k}(\alpha \Phi_{X_k}(\alpha)), \quad (27) \end{aligned}$$

where, for simplicity, we have substituted α for e^ν . Now, using the same approach as we did with A_k , we obtain

$$E[z^{B_k} | U = u] = e^{\lambda_k(1-u)[\Phi_{m_k}(z)-1]}.$$

Thus, removing the conditioning on B_k in (27) yields

$$\begin{aligned} E[\alpha^{W_{k1}} | U = u] &= \alpha^u e^{\lambda_k(1-u)[\Phi_{m_k}(\alpha\Phi_{X_k}(\alpha))-1]} \cdot \Phi_{Q_k}(\alpha\Phi_{X_k}(\alpha))\Phi_{A_k}(\alpha\Phi_{X_k}(\alpha)) \\ &= [e^{\nu-\lambda_k[\Phi_{m_k}(\alpha\Phi_{X_k}(\alpha))-1]}]^u \Phi_{Q_k}(\alpha\Phi_{X_k}(\alpha))\Phi_{A_{k+1}}(\alpha\Phi_{X_k}(\alpha)). \end{aligned}$$

Now, removing the conditioning on U , we obtain

$$G_{W_{k1}}(\nu) = G_U\{\nu - \lambda_k[\Phi_{m_k}(\alpha\Phi_{X_k}(\alpha)) - 1]\} \Phi_{Q_k}(\alpha\Phi_{X_k}(\alpha))\Phi_{A_{k+1}}(\alpha\Phi_{X_k}(\alpha)),$$

where

$$G_U(\nu) = \int_0^1 e^{\nu u} du = \frac{1}{\nu} [e^\nu - 1]. \quad (28)$$

Finally using (19), we obtain

$$\begin{aligned} G_{W_{kj}}(\nu) &= G_U\{\nu - \lambda_k[\Phi_{m_k}(\alpha\Phi_{X_k}(\alpha)) - 1]\} \\ &\quad \cdot \Phi_{Q_k}(\alpha\Phi_{X_k}(\alpha))\Phi_{A_{k+1}}(\alpha\Phi_{X_k}(\alpha)) \cdot [\alpha\Phi_{X_k}(\alpha)]^{j-1}, \end{aligned}$$

where $\alpha = e^\nu$, $G_U(\nu)$ is given by (28), $\Phi_{Q_k}(z)$ is given by (25), $\Phi_{A_{k+1}}(z)$ is given by (17), and $\Phi_{X_k}(z)$ is given by (18).

The delay in transmitting a source k message of length m , $D_k(m)$, is given by

$$D_k(m) = W_{km} + 1.$$

Hence, the MGF for $D_k(m)$ is given by

$$G_{D_k(m)}(\nu) = e^\nu G_{W_{km}}(\nu).$$

It follows that the MGF for D_k , the delay in transmitting a randomly selected source k message, is given by

$$G_{D_k}(\nu) = G_{W_{k1}}(\nu)\Phi_{m_k}(e^\nu\Phi_{X_k}(e^\nu))/\Phi_{X_k}(e^\nu).$$

AUTHORS

Robert R. Boorstyn, B.E.E., 1958, City College of New York; M.S. (Electrical Engineering), and Ph.D. (Electrical Engineering), 1963 and 1966, respectively, Polytechnic Institute of Brooklyn; Sperry Gyroscope Company, 1958-1961; Polytechnic Institute of New York, 1961-; on leave at Bell Laboratories, 1977-1978; consultant at Bell Laboratories, 1981-1982. Mr. Boorstyn is engaged in research on computer communication networks, specializing in packet radio networks, routing, network design, and analysis. He is currently a Professor of Electrical Engineering and Computer Science at Polytechnic Institute of New York. He has been Editor for Computer Communications of the *IEEE Transactions on Communications*, Chairman of the Computer Com-

munications Committee of the IEEE Communications Society, and Secretary of the IEEE Information Theory Group. He is an Associate Editor of *Networks Journal*.

Michael G. Hluchyj, B.S. (Electrical Engineering), 1976, University of Massachusetts at Amherst; S.M., E.E., and Ph.D. (Electrical Engineering), Massachusetts Institute of Technology, 1978, 1978, and 1981, respectively; Bell Laboratories, 1981-1982; AT&T Information Systems, 1983—. Mr. Hluchyj's work at Bell Laboratories and AT&T Information Systems has centered around the architectural design and performance analysis of local area data communication networks.

Chan David Tsao, B.S. (Control Engineering), 1973, National Chiao Tung University (Taiwan); M.S. (Systems Engineering), 1978, Florida Institute of Technology; M.S. and Ph.D. (Electrical Engineering), Polytechnic Institute of New York, 1979 and 1982, respectively; Bell Laboratories, 1981-1982; AT&T Information Systems, 1983—. At Bell Laboratories, Mr. Tsao worked on performance evaluation and system design for local area networks. His current research interest is architectures and applications for local area networks.

THE BELL SYSTEM TECHNICAL JOURNAL is abstracted or indexed by *Abstract Journal in Earthquake Engineering*, *Applied Mechanics Review*, *Applied Science & Technology Index*, *Chemical Abstracts*, *Computer Abstracts*, *Current Contents/Engineering, Technology & Applied Sciences*, *Current Index to Statistics*, *Current Papers in Electrical & Electronic Engineering*, *Current Papers on Computers & Control*, *Electronics & Communications Abstracts Journal*, *The Engineering Index*, *International Aerospace Abstracts*, *Journal of Current Laser Abstracts*, *Language and Language Behavior Abstracts*, *Mathematical Reviews*, *Science Abstracts (Series A, Physics Abstracts; Series B, Electrical and Electronic Abstracts; and Series C, Computer & Control Abstracts)*, *Science Citation Index*, *Sociological Abstracts*, *Social Welfare*, *Social Planning and Social Development*, and *Solid State Abstracts Journal*. Reproductions of the Journal by years are available in microform from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106.

