Pioneers and Their Contributions to Software Engineering

sd&m Conference on Software Pioneers

Bonn, June 28/29, 2001

Original Historic Contributions

Springer

Friedrich L. Bauer
From the stack principle to Algol

Rudolf Bayer
B-trees and relational data base systems

Barry Boehm
Software economics

Fred Brooks
The /360 architecture and its operating system

Peter Chen
Entity/relationship modelling

Ole-Johan Dahl
The root of object-oriented programming: Simula 67

Tom DeMarco
Structured analysis

Edsger W. Dijkstra
From "goto considered harmful" to structured programming

Michael Fagan
Reviews and inspections

Erich Gamma
Design patterns

John Guttag
Algebraic specification of abstract data types

C.A.R. Hoare
Software fundamentals

Michael Jackson
Data structures form algorithms

Alan Kay
Mice and windows

David L. Parnas
On the criteria to be used in decomposing systems into modules

Niklaus Wirth
Teaching programming principles: Pascal

Pioneers and Their Contributions
to Software Engineering

المنارة للاستشارات

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

المنارة للاستشارات

Manfred Broy • Ernst Denert (Eds.)

Pioneers and Their Contributions to Software Engineering

sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions

Springer

المنارة للاستشارات

Editors
Manfred Broy
Institut für Informatik
Technische Universität München
80290 München, Germany
broy@informatik.tu-muenchen.de

Ernst Denert
sd&m AG
software design & management
Postfach 83 08 51
81708 München, Germany

Sonderausgabe
Buch nicht im Handel erhältlich.

ISBN 978-3-540-42290-7 ISBN 978-3-642-48354-7 (eBook)
DOI 10.1007/978-3-642-48354-7

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2001

The use of general descriptive names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by the authors
Printed on acid-free paper SPIN 10844024 - 06/3142SR - 5 4 3 2 1 0

المنارة للاستشارات

Table of Contents

Friedrich L. Bauer

K. Samelson, F.L. Bauer
Sequentielle Formelübersetzung 3

Friedrich L. Bauer
Verfahren zur automatischen Verarbeitung
von kodierten Daten und Rechenmaschinen
zur Ausübung des Verfahrens 31

Rudolf Bayer

R. Bayer, E. McCreight
Organization and Maintenance of Large Ordered Indexes 43

E.F. Codd
A Relational Model of Data for Large Shared Data Banks 63

Barry Boehm

Software Engineering Economics 101

Fred Brooks

G.H. Mealy, B.I. Witt, W.A. Clark
The Functional Structure of OS/360 153

Peter Chen

The Entity Relationship Model – Toward a Unified View of Data 207

Ole-Johan Dahl

Ole-Johan Dahl, Kristen Nygaard
Class and Subclass Declarations 237

Tom DeMarco

Structure Analysis and System Specification 257

Edsger Dijkstra

Solution of a Problem in Concurrent Programming Control 291

Go To Statement Considered Harmful 297

Michael Fagan

Design and Code Inspections to Reduce Errors in Program Development 303

Advances in Software Inspections 337

المنارة للاستشارات

Erich Gamma
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns: Abstraction and Reuse of Object-Oriented Design 363

John Guttag
Abstract Data Types and the Development of Data Structures 391

C.A.R. Hoare
An Axiomatic Basis for Computer Programming 421

Proof of Correctness of Data Representations 441

Michael Jackson
Constructive Methods of Program Design 455

David L. Parnas
On the Criteria to Be Used in Decomposing Systems into Modules 481

On a 'Buzzword': Hierarchical Structure 501

Niklaus Wirth
The Programming Language Pascal 517

Program Development by Stepwise Refinement 547

المنارة للاستشارات

Friedrich L. Bauer

K. Samelson and Friedrich L. Bauer
Sequentielle Formelübersetzung

Elektronische Rechenanlagen 1, 1959
pp. 176–182

المنارة للاستشارات

Sequentielle Formelübersetzung

Sequential Formula Translation

von K. SAMELSON und F. L. BAUER

Universität Mainz

Elektronische Rechenanlagen 1 (1959), H. 4, S. 176—182
Manuskripteingang: 9. 9. 1959

Die Syntax einer Formelsprache wie ALGOL läßt sich als Folge von Zuständen beschreiben, die durch ein Keller genanntes Element angezeigt werden. Die Übergänge werden gesteuert durch zulässige Zustand-Zeichen-Paare, die sich in Form einer Übergangsmatrix darstellen lassen. Diese Beschreibung liefert gleichzeitig eine äußerst einfache Vorschrift zur Übersetzung der Anweisungen der Formelsprache in Maschinenprogramme. Lediglich Optimisierungsprozesse wie die rekursive Adressenfortschaltung entziehen sich der sequentiellen Behandlung.

The syntax of an algorithmic language such as ALGOL is conveniently described as a sequence of states indicated by an element called cellar. Transitions are controlled by admissable state-symbol pairings which may be represented by a transition matrix. This description at the same time furnishes an extremely simple rule for translating statements of the algorithmic language into machine programs. Sequential treatment, however, is not feasible in the case of optimizing processes such as recursive address calculation.

Verwendete Zeichen

Es gelten alle Bezeichnungen von [12], Elektronische Rechenanlagen 1 (1959), 72. Darüber hinaus oder abweichend sind verwendet:

Symbol $\chi \sim$ *I*, *N*, *'go to'* etc.
Zeichen $\alpha \sim + - \times / ( )$
Ergibtzeichen $\Rightarrow$

المنارة للاستشارات

Adresse von z $\rangle z \langle$
Inhalt einer Speicherzelle $\langle \varphi \rangle$
mit der Adresse φ
AC Inhalt des Akkumulators
æ Ende des Ausdrucks
η Inhalt der Zahlkelleradresse
φ Adresse
Φ Adressenkeller
$\varnothing$ Leersymbol beim Keller
h Zählerstand des Zahlkellers
H Zahlkeller
K Befehlsfolge
Π Programm
s Nummer des Kellersymbols
σ Kellersymbol
Σ Symbolkeller

1. Einleitung, Grund und Entwicklung der Formelübersetzung

Die schnelle Entwicklung des Baues programmgesteuerter Rechenanlagen in den letzten zehn Jahren hat dazu geführt, daß heute eine beträchtliche Anzahl verschiedener Automatentypen hergestellt wird. Alle diese Maschinentypen haben jedoch, trotz großer Unterschiede in Konstruktion und Befehlscode, zwei Charakteristika gemeinsam, die nach allgemeiner (möglicherweise nicht vorurteilsfreier) Ansicht technisch bedingt sind, nämlich

1. den in eine eindimensionale Folge von Worten fester Zeichenlänge zerlegten Speicher (Arbeitsspeicher),
2. Das entsprechend in eine Folge fester unabhängiger Elemente (der Befehle) zerlegte Programm, das von der Steuerung Befehl für Befehl abgearbeitet wird. Dies bedeutet, daß die einem ins Steuerwerk gelangenden Maschinenbefehl zukommende Operation unabhängig ist von der Befehlsvorgeschichte.

Diese beiden Merkmale stellen sich dem Benutzer der Rechenanlage, also dem Programmhersteller, als Hindernisse entgegen, insofern sie verantwortlich sind für die bekannte Unbequemlichkeit und Irrtumsanfälligkeit des Programmierens in Maschinencode. Denn sie erfordern das

المنارة للاستشارات

Operieren mit Adressen und bedingen darüber hinaus eine völlige Atomisierung des Programms. Es ist wichtig festzustellen, daß dieser Zwang unnatürlich ist: ein Problem irgendwelcher Art, das von einer Rechenanlage behandelt werden soll, entsteht in der gedanklichen Konzeption zunächst meist als Ablaufschema für gewisse größere Operationseinheiten, die durch ihren Zweck umrissen und mehr oder weniger vage durch die dem Problemkreis eigentümlichen Bezeichnungen angegeben werden. Die Ausgestaltung des Problems führt zu einer operativen Fixierung, die in möglichst rationeller Form unter Benutzung gebräuchlicher Notation geschieht, vornehmlich unter Heranziehung mathematischer Formeln und verbaler Erläuterungen. Eine Atomisierung in kleinste Einzeloperationen ist unökonomisch hinsichtlich der darauf zu verwendenden Zeit und des erforderlichen Platzes, vor allem führt sie zur Unübersichtlichkeit. Die Hinzunahme der Adressen als völlig künstlicher Elemente wiegt noch schwerer, sie erfordert umfangreiche Buchführung und überdies in rekursiven Prozessen Adressenberechnungen, die sich der eigentlichen Aufgabe überlagern. Die Verhältnisse werden geradezu paradox bei gewissen Grundaufgaben der Numerischen Mathematik: ein generell brauchbares Programm zur Lösung eines linearen Gleichungssystems enthält etwa hundert einzelne Befehle, unter denen ein einziger Additions- und ein einziger Multiplikationsbefehl der eigentlichen Aufgabe dienen. Insbesondere die mit der Einführung der Adressen verbundenen Arbeitsgänge sind weitgehend routinemäßiger Natur, und man hat daher schon frühzeitig versucht, sie wenigstens teilweise dem Rechenautomaten selbst zuzuschieben, der dabei als reiner Codeumsetzer arbeitet [1], [8], [9].

Gewisse Erleichterungen verschaffte man sich ferner durch den Gebrauch vorgefertigter Bibliotheksprogramme für standardisierte Operationseinheiten, die, mit Codeworten bezeichnet, ebenfalls vom Rechenautomaten direkt aufgerufen, d. h. in den Ablauf eingeordnet werden. Derart aufgebaute Programmierungssysteme waren ab 1954 in allgemeinem Gebrauch, wobei das Programm, das die Routinearbeiten der Programmierung („automatische Programmierung") erledigte, als Compiler bezeichnet wurde [10].

المنارة للاستشارات

Daß man sich bei numerischen Aufgaben eine effektive Lösung des Problems der Programmierung erst erhoffen kann, wenn man bei der automatischen Programmfertigung von den in konventioneller Schreibweise geschriebenen Formeln ausgeht und alle weiteren Phasen dem Automaten überläßt, hat schon 1951 *Rutishauser* [4] erkannt. Sein Verwirklichungsvorschlag [5] sowie die daran anknüpfende Arbeit von *Böhm* [14] blieb jedoch unbeachtet, und erst 1955 wurden mit PACT [3] und FORTRAN [2] die ersten Programmierungssysteme mit Formelübersetzungscharakter aufgebaut, ohne daß jedoch etwas über die dabei verwendeten Methoden publiziert worden wäre. Etwa gleichzeitig begannen in Kenntnis der Rutishauserschen Ergebnisse ähnliche Überlegungen am Rechenzentrum der TH München, wobei die Entwicklung solcher Übersetzungsmethoden im Vordergrund stand, die auch für Anlagen von wesentlich geringerem Umfang und Leistungsfähigkeit als etwa der IBM 704 anwendbar sein sollten. Zu diesem Zwecke wurde, auf unabhängigen Vorarbeiten basierend [6], [13], eine sequentielle Übersetzungstechnik entwickelt. Die Arbeiten wurden seit 1957 im Rahmen der heutigen Arbeitsgruppe Zürich—München—Mainz—Darmstadt (ZMMD) fortgesetzt.

Inzwischen hatte sich jedoch eine prinzipielle Verschiebung der Standpunkte angebahnt: die herkömmlichen Programmierungssysteme waren noch vom Maschinencode als dem Ziel der Übersetzung her aufgebaut, und die Übersetzung selbst war schrittweise aus einer Übertragung der bisher von Menschen geleisteten Routinearbeit auf die Rechenanlage entstanden, wobei die Sprache des jeweiligen Programmierungssystems von der Struktur der Rechenanlage her immer weniger bestimmt war. Mit der Beherrschung der Technik des Übersetzungsvorganges gewann man nun auch Freiheit in der Wahl der Programmierungssprache, und die Aufstellung einer möglichst bequem handzuhabenden, übersichtlichen, selbstverständlichen Sprache trat als Aufgabe hervor, die gelöst werden mußte, bevor die Übersetzer selbst programmiert werden konnten. Insbesondere entstand die verlockende Möglichkeit, für verschiedene Rechenanlagen, zunächst innerhalb der ZMMD-Gruppe, dieselbe Programmierungssprache zu verwenden.

المنارة للاستشارات

Die Entwicklung führte 1958 zum Vorschlag einer algorithmischen Formelsprache (ALGOL) durch ein gemeinsames ACM-GAMM-Kommittee [11], [12]. In der Zwischenzeit wurde, nunmehr auf der Basis von ALGOL, die Struktur des Formelübersetzers der ZMMD-Gruppe einheitlich festgelegt und mit der Codierung für die Rechenanlagen der beteiligten Institute (ERMETH, PERM, Z 22, SIEMENS) sowie für die Rechenanlagen einiger befreundeter Institute in Deutschland, USA, Österreich und Dänemark nach diesem ALCOR (ALGOL Converter) genannten System begonnen.

Da somit dieses Projekt seiner Vollendung entgegengeht, erscheint es an der Zeit, einen Überblick über die ihm zugrunde liegenden Prinzipien der sequentiellen Übersetzung zu geben, die sowohl von dem ursprünglichen Rutishauserschen Vorschlag [5] als auch von den kürzlich veröffentlichten Methoden des FORTRAN-Systems [7] wesentlich abweichen[1]). Ausführliche Strukturpläne, die das ganze Formelübersetzungsprogramm in detaillierter Form ohne Bezugnahme auf eine spezielle Maschine beschreiben, wurden im Institut für Angewandte Mathematik der Universität Mainz in reproduktionsfähige Form gebracht; sie bilden die Grundlage der oben erwähnten Zusammenarbeit der ALCOR-Familie.

2. Sequentielle Übersetzung und das Kellerungsprinzip

Die in einer Formelsprache wie ALGOL niedergeschriebenen Anweisungen sind eine Folge von Symbolen, die sich ihrerseits aus einem oder mehreren Charakteren zusammensetzen. Da der Aufbau von Symbolen aus Charakteren jedoch trivial (es handelt sich stets um lückenlose, eindeutig abgegrenzte Folgen) und bis zu einem gewissen Grade von technischen Gegebenheiten wie dem verwendeten Schreibgerät abhängig ist, werden wir im folgenden den Unterschied zwischen Symbolen und Charakteren unterdrücken und jedes Symbol χ als Einheit betrachten. Dies gilt insbesondere für Identifier I, Zahlen N und verbal definierte Begrenzer wie 'go to', 'if' usw.

---

[1]) Einzelne Züge des Systems finden sich bereits in der erwähnten Arbeit von *Böhm* [14], der jedoch starke Einschränkungen hinsichtlich der zulässigen Notation macht.

المنارة للاستشارات

Die Folge von Symbolen χ des Formelprogramms stellt nun (mit der üblichen Interpretation der Symbole) eine Arbeitsvorschrift dar. Dabei ist es jedoch nicht möglich, die Symbole in der angegebenen Reihenfolge in orthodoxe Maschinenoperationen zu übersetzen. Vielmehr erzwingen bereits bestimmte arithmetische Symbole, die Klammern (), und Vorrangregeln ($\times$ vor $+$) eine von der Symbolanordnung abweichende Reihenfolge der Operationen. So heißt $a \times b + c \times d$: multipliziere a mit b, multipliziere c mit d, addiere die Produkte, während die sequentielle Auswertung ergeben würde: multipliziere a mit b, addiere dazu c und multipliziere das Resultat mit d.
Es ist also bei der Abarbeitung des Formelprogramms ständig notwendig, gelesene Symbole als nicht auswertbar zu übergehen und in einem späteren, von der weiteren Symbolfolge abhängigen Zeitpunkt wiederzufinden und auszuwerten. *Rutishauser* hat mit dem „Klammergebirge" die grundsätzliche Lösung angegeben. Die von ihm vorgeschlagene Ausführung, durch Vorwärts- und Rückwärtslesen die ausführbare Operation einzukreisen, ist aber unbequem und (unnötig) zeitraubend. Daran ändert sich auch nicht viel, wenn man *Rutishausers* Methode dahingehend variiert, daß man bereits lokale Gipfel abarbeitet.
Das Problem, die beim ersten Erscheinen als nicht auswertbar übergangene Information im richtigen Augenblick wieder greifbar zu haben, läßt sich aber mit Hilfe eines als Kellerung bezeichneten Prinzips weitgehend vereinfachen, das immer anwendbar ist, wenn die Struktur der Symbolfolge klammerartigen Charakter hat. Das soll heißen, daß zwei verschiedene Paare A, A' und B, B' zusammengehöriger Elemente sich nur umfassen, aber nicht gegenseitig trennen können, daß also nur Anordnungen $ABB'A'$ und nicht $ABA'B'$ vorkommen.
Das Prinzip besagt: Man setze alle nicht sofort auswertbaren Informationen in der Reihenfolge des Einlaufens in einem besonderen Speicher, dem „Symbolkeller", ab, in dem jeweils nur das zuletzt abgesetzte, im obersten Geschoß befindliche Element interessiert und damit unmittelbar zugänglich zu sein braucht. Jedes neu gelesene Symbol wird mit dem obersten Kellersymbol verglichen. Die beiden Symbole in Konjunktion legen fest, ob das Kellersymbol

المنارة للاستشارات

in eine Operation umgesetzt werden kann, worauf es aus dem Keller entfernt wird. Je nach den Umständen wird der Vergleich mit dem nunmehr obersten Symbol des Kellers wiederholt und schließlich gegebenenfalls ein neues Zustandssymbol im Keller abgesetzt.
In der Sprechweise der Theorie der Automaten kann das Prinzip so formuliert werden: Durch die gesamte Besetzung des Kellers wird ein Zustand (des Übersetzungsvorgangs) definiert, der effektiv in jedem Augenblick nur von dem obersten Kellerzeichen abhängt, und neu gelesene Information plus Zustand bestimmen die Aktionen des Übersetzers, die aus der Abgabe von Zeichen, nämlich von Operationsanweisungen für das erzeugte Programm und der Festlegung eines neuen Zustands bestehen. Das Wesentliche ist aber die durch die Besetzung des Kellers induzierte latente Zustandsstruktur.

3. Auswertung einfacher arithmetischer Ausdrücke

Den wichtigsten Fall der Symbolfolgen mit Klammerstruktur stellen die arithmetischen Ausdrücke dar, deren Behandlung wir daher als Beispiel ausführlich besprechen wollen. Um aber den prinzipiellen Sachverhalt nicht mit relativ unwichtigen Details zu belasten, werden wir einige Vereinfachungen vornehmen.
Diese betreffen einmal die zulässigen Symbole. Wir werden Funktionen $\boldsymbol{I}$ $(\boldsymbol{P}, \ldots, \boldsymbol{P})$ und indizierte Variable $\boldsymbol{I}$ $[\boldsymbol{E}, \ldots, \boldsymbol{E}]$ vorläufig ausschließen und die Additionssymbole $\pm$ nur als zweistellige Operation $(a \pm b)$ und nicht als einstellige $(\pm a)$ zulassen.

Weiter werden wir zur Erläuterung hinsichtlich der Rechengrößen selbst unterstellen, daß dem Rechenwerk der Maschine, für die das Programm hergestellt werden soll, ein Schnellspeicher begrenzter Kapazität zur Verfügung steht, dessen Zugriffszeit vernachlässigbar ist gegenüber der Zugriffszeit des Arbeitsspeichers, so daß für alle Zahlen, die zur Verarbeitung dem Rechenwerk zur Verfügung gestellt werden sollen, ein vorübergehendes Absetzen im Schnellspeicher keine Verzögerung des Ablaufs des Resultatprogramms bedeutet.
Dieser Schnellspeicher habe nun dieselbe Kellerstruktur wie der Symbolkeller, d.h., seine Plätze werden sukzessive

المنارة للاستشارات

belegt, und die jeweils zuletzt abgespeicherte (gekellerte) Zahl ist als erste abrufbar. Der Speicher werde deshalb als Zahlkeller H bezeichnet.

Jeder unter den gemachten Voraussetzungen in einem Ausdruck auftretende Identifier stellt eine Variable dar, d. h. den Decknamen für eine Zahl, und ist somit eine symbolische Adresse, die von dem Übersetzer in irgendeiner Weise auf eine echte Speicheradresse abgebildet wird, wie dies schon von allen mit symbolischen Adressen arbeitenden Compilern getan wird[2]). Zahlen N sind, gegebenenfalls nach Konvertierung, in Zellen abzusetzen und ebenfalls durch Adressen zu ersetzen, so daß wir sie weiterhin außer Betracht lassen können.

Die Auswertung eines arithmetischen Ausdrucks mit Hilfe des Kellerungsprinzips geht nun in folgender Weise vor sich:

a) Jeder auftretende Identifier I veranlaßt die Überführung des Inhalts der entsprechenden Speicherzelle in den jeweils obersten Platz des Zahlkellers H. Das Wort „veranlaßt" bedeutet hier, daß der Übersetzer die entsprechenden Befehle an den bereits aufgebauten Teil des zu erzeugenden Maschinenprogramms anfügt. Ein im Übersetzer enthaltener Zähler h hat den jeweils obersten Platz des Zahlkellers anzuzeigen und muß daher gleichzeitig eine Eins aufzählen. Bezeichnen wir den Speicher für das erzeugte Programm mit Π, die Inhalte der Plätze des Zahlkellers H mit η_h, wobei der Index h die Zählgröße darstellt, und die Maschinenbefehlsfolge $I \Rightarrow \eta_h$, die die Überführung in den Zahlkeller darstellt, mit K_I, so sind die vom Übersetzer auszuführenden Operationen:

$$I\colon h \dotplus 1 \Rightarrow h;\ K_I \Rightarrow \Pi;\ \text{lies } \chi$$

'lies χ' bedeutet hier, daß das nächste Zeichen χ des Ausdrucks zu lesen ist.

b) Alle übrigen Symbole α, das sind $+$, $-$, $\times$, $/$, (,), werden beim Einlaufen mit dem jeweils obersten, als σ_s bezeichneten Symbol des Symbolkellers verglichen, der im Anfangs-

[2]) Die einfachste Möglichkeit wäre etwa, die Zahl der zulässigen Identifier soweit zu beschränken, daß jedem Identifier ein fester oder wenigstens relativ zu dem erzeugten Programm fester Speicherplatz zugewiesen wird.

المنارة للاستشارات

zustand das Leersymbol $\varnothing$ enthält. Jedes aus einem Kellersymbol σ_s und einem Formelzeichen α bestehende Paar veranlaßt eine bestimmte Folge von Operationen des Übersetzers entsprechend der folgenden Liste:

σ_s	α	
$\varnothing$	α	$1 \Rightarrow s$; $\alpha \Rightarrow \sigma_s$; lies χ;
$+ -$	$+ -$	$K_\sigma \Rightarrow \Pi$; $\alpha \Rightarrow \sigma_s$; $h - 1 \Rightarrow h$; lies χ;
$\times /$	$\times /$	
(	$+ - \times /$	$s + 1 \Rightarrow s$; $\alpha \Rightarrow \sigma_s$; lies χ;
$+ -$	$\times /$	
$+ - \times /$	)	
(	)	$s - 1 \Rightarrow s$; lies χ;
$\times /$	$+ -$	$K_\sigma \Rightarrow \Pi$; $s - 1 \Rightarrow s$; $h - 1 \Rightarrow h$; repetiere mit α;
$+ - \times /$	) æ	

Die vom Übersetzer erzeugten und an den Programmspeicher Π abgegebenen Maschinenbefehlsfolgen K_σ haben dabei stets die folgende Dreiadreßform, wobei σ eines der vier Operationssymbole $+ - \times /$ darstellt:

$$K_\sigma \colon \eta_{h-1} \sigma \eta_h \Rightarrow \eta_{h-1} .$$

Es werden also die jeweils beiden obersten Elemente des Zahlkellers η_{h-1} und η_h durch die mit σ bezeichnete Operation verknüpft und das Resultat als nunmehr oberstes Element η_{h-1} an den Zahlkeller zurückgegeben. Mit der Abgabe dieser Befehlsfolge muß daher auch der Zähler h des Zahlkellers um Eins heruntergezählt werden.

'Repetiere mit α' bedeutet, daß im nächsten Schritt mit dem gleichen Symbol α und dem neuen σ_s zu arbeiten ist.

Das Ende eines Ausdrucks muß natürlich erkennbar sein. Es ist hier mit 'æ' angedeutet und wirkt wie eine dem Anfang als öffnender Klammer zugeordnete schließende Klammer.

Die Liste von Zeichenpaaren σ_s, α läßt sich bequem durch eine Matrix darstellen, deren Zeilen den möglichen Keller-

المنارة للاستشارات

$$A: (a \times b + c \times d)/(a - d) + b \times c$$

Σ	χ (α oder I)	Π
leer	(	
(	a	$a \Rightarrow \eta_1$
(	$\times$	
($\times$	b	$b \Rightarrow \eta_2$
($\times$	$+$	$\eta_1 \times \eta_2 \Rightarrow \eta_1$
($+$	c	$c \Rightarrow \eta_2$
($+$	$\times$	
($+ \times$	d	$d \Rightarrow \eta_3$
($+ \times$	)	$\eta_2 \times \eta_3 \Rightarrow \eta_2$
($+$		$\eta_1 + \eta_2 \Rightarrow \eta_1$
(		
leer	/	
/	(	
/(	a	$a \Rightarrow \eta_2$
/(	$-$	
/($-$	d	$d \Rightarrow \eta_3$
/($-$	)	$\eta_2 - \eta_3 \Rightarrow \eta_2$
/(		
/	$+$	$\eta_1/\eta_2 \Rightarrow \eta_1$
$+$	b	$b \Rightarrow \eta_2$
$+$	$\times$	
$+ \times$	c	$c \Rightarrow \eta_3$
$+ \times$	æ	$\eta_2 \times \eta_3 \Rightarrow \eta_2$
$+$	æ	$\eta_1 + \eta_2 \Rightarrow \eta_1$
leer		

symbolen σ_s und deren Spalten den Formelzeichen α zugeordnet sind, so daß jedem Paar ein Matrixelement entspricht[3]). Diese Übergangsmatrix liefert eine vollständige syntaktische und operative Beschreibung aller zulässigen arithmetischen Ausdrücke.

Anfangszustand ist stets $s = 0$ ($\sigma_s = \varnothing$) und $h = 0$ (Zahlkeller leer), ein zulässiger Endzustand, der

[3]) Bei *Böhm* [14], der für eine stark eingeschränkte Formelsprache bereits eine matrixartige Übersetzungsvorschrift gibt, fehlt der Symbolkeller. *Böhm* hat jedoch bereits die Auswertung klammerfreier Ausdrücke durch Vergleich aufeinanderfolgender Operationszeichen.

المنارة للاستشارات

einem vollständigen Ausdruck entspricht, ist mit $s = 0$ und $h = 1$ erreicht. Der Wert eines vollständigen Ausdrucks findet sich also stets auf dem ersten Platz des Zahlkellers.

Ein einfaches Beispiel möge den Ablauf erläutern, wobei wir nur den jeweiligen Inhalt des Symbolkellers Σ, das neu einlaufende Zeichen χ und das in Π aufgebaute Programm angeben.

$$A: (a \times b + c \times d)/(a - d) \div b \times c$$

Wie man aus dero bigen Tabelle sieht, ist die Reihenfolge der Operationen im entstehenden Programm durch das Formelprogramm völlig festgelegt, und es wird kein Versuch gemacht, etwa zur Beschleunigung Umstellungen vorzunehmen. Denn die Wahl der Reihenfolge der Operationen muß völlig in der Hand des das Programm entwerfenden Mathematikers liegen. Jede Umstellung kann wegen der Ungültigkeit des assoziativen Gesetzes (wenigstens beim Rechnen mit gleitendem Komma) unerwünschte numerische Konsequenzen haben.

4. Vollständige arithmetische Ausdrücke

Wir haben nun zu diskutieren, wie das oben angegebene Schema zu variieren ist, wenn wir die angegebenen Vereinfachungen fallen lassen. Betrachten wir zunächst die Behandlung der Rechengrößen:

Das angegebene Beispiel zeigt deutlich, daß eine Anzahl unnötiger Umspeicherungen vorgenommen wird. Tatsächlich sind alle Operationen $I \Rightarrow \eta_h$ überflüssig, und Variable dürfen von ihrem Platz nur zur Ausführung von Rechenoperationen ins Rechenwerk abgerufen werden. Wenn wir uns, was weiterhin vorausgesetzt sein soll, auf den Fall der Einadreßmaschine beschränken, so fallen Ergebnisse stets im Akkumulator an. Der Zahlkeller darf nur noch dazu dienen, solche (unbenannte) Zwischenergebnisse aufzunehmen, deren Abspeicherung notwendig ist, um das Rechenwerk für die nachfolgenden Operationen freizumachen. Für diese arbeitet er in der vorher beschriebenen Weise.

المنارة للاستشارات

Im übrigen tritt aber an die Stelle des Zahlkellers ein von dem Übersetzer auszuwertender (Variablen- oder) Adressenkeller Φ, und alle überflüssigen Transportoperationen sind durch Eintragung der entsprechenden Adressen in diesem Keller zu ersetzen, die durch den Übersetzer vorgenommen wird und die Programmerzeugung mitsteuert. Da nun auch der Akkumulator als Zahlspeicher verwendet wird, ist es zweckmäßig, auch ihm eine (identifizierbare) Pseudoadresse zuzuweisen, die in den Adressenkeller eingetragen wird. Notwendige Abspeicherungen von Zwischenresultaten ergeben sich dann daraus, daß eine öffnende Klammer auf ein arithmetisches Operationszeichen im Symbolkeller Σ stößt, dem als oberstes Element des Adressenkellers die Adresse des Akkumulators entspricht. Eine solche Klammer wird impliziert auch durch ein einlaufendes $\times/$, das auf ein $\pm$ in Σ stößt. Da zwischen diesen Symbolen ein Identifier aufgetreten sein muß, ist in diesem Falle auch die zweithöchste Position des Adressenkellers zu kontrollieren.

Ist eine Zwischenspeicherung notwendig, so wird die Abspeicherung des AC in den gerade obersten Platz des Zahlkellers H veranlaßt, die Adresse des AC im Adreßkeller durch die Adresse von η_h ersetzt und angemerkt, daß bei Abruf der Adresse in das erzeugte Programm der Zahlkellerindex um Eins heruntergezählt werden muß.

Die vom Übersetzer in den Programmspeicher abzusetzenden Operationen K_σ erhalten jetzt im allgemeinen die Form

$$K_\sigma\colon \begin{array}{l} \langle\varphi_{f-1}\rangle \Rightarrow \mathrm{AC} \\ \mathrm{AC}\,\sigma\,\langle\varphi_f\rangle \Rightarrow \mathrm{AC}. \end{array}$$

Dabei ist jedoch stets zu prüfen, ob eine der beiden Operandenadressen φ_{f-1}, φ_f den AC darstellt. In diesem Falle fällt für σ gleich $+$ oder $\times$ der erste Befehl aus, der zweite erhält die Adresse φ_f oder φ_{f-1}, die nicht den AC darstellt. Bei σ gleich — oder / fällt der erste Befehl weg, wenn φ_{f-1} den AC darstellt. Im entgegengesetzten Falle aber, also $\varphi_f = \mathrm{AC}$, muß man für $\sigma = -$ setzen:

$$K_-\colon \begin{array}{r} -\mathrm{AC} \Rightarrow \mathrm{AC} \\ \mathrm{AC} + \langle\varphi_{f-1}\rangle \Rightarrow \mathrm{AC}, \end{array}$$

المنارة للاستشارات

während man für σ gleich / sogar zuerst den AC sicherstellen muß.

$$K_/: \begin{array}{r} \mathrm{AC} \Rightarrow \eta_h \\ \langle \varphi_{f-1} \rangle \Rightarrow \mathrm{AC} \\ \mathrm{AC}/\eta_h \Rightarrow \mathrm{AC} \end{array}$$

Die beiden Fälle entsprechen Formeln vom Typ $a-(b+c)$ bzw. $a/(b+c)$, die sich bequemer mit Maschinen behandeln ließen, die „vom Speicher subtrahieren" bzw. „in den Speicher dividieren" können.
Die einstelligen Operationen $+a$, $-b$ schließlich erledigen sich begrifflich am einfachsten durch Hinzunahme eines Leerelementes im Adressenkeller, das anzeigt, daß der entsprechende Linksoperand nicht existiert.
Die Behandlung Boolescher Ausdrücke läuft offensichtlich der Behandlung arithmetischer Ausdrücke parallel.
Die Hinzunahme von Funktionen und indizierten Variablen bedeutet zunächst einmal, daß das Auftreten eines Identifiers unmittelbar von einer öffnenden Klammer festgestellt werden muß, da die Kombination $I($ die Funktionen und die Kombination $I[$ die indizierten Variablen eindeutig kennzeichnet. Weiter, und das ist der wesentliche Punkt, stellen beide Symbole, Funktion und indizierte Variable, einen neuen Typ von Klammer mit besonderen Eigenschaften dar. Wenn wir uns hinsichtlich der indizierten Variablen zunächst auf den Fall beschränken, daß die durch die der Variablen zugehörige Feld-Vereinbarung (array declaration) festgelegte Speicherabbildungsfunktion (vgl. Abschnitt 7) für jedes Auftreten der Variablen vollständig ausgewertet wird, ist die Behandlung weitgehend einheitlich.
Zunächst ist der Reihe nach die Auswertung der auf den einzelnen Argument- bzw. Indexpositionen stehenden Ausdrücke zu veranlassen, wobei das trennende Komma bzw. die abschließende Klammer) oder] die Rolle des Abschlußzeichens übernimmt. Die Werte der Ausdrücke sind abzuspeichern, konsequenterweise als Zwischenergebnisse im Zahlkeller. Anschließend an die Berechnung der Argumente ist ein Sprung mit automatischer Rückkehr zu setzen, der in das durch die Funktions- bzw. Feld-Vereinbarung definierte Programm führt. Dieses endet wie üblich mit der Abgabe des ermittelten Wertes an den Akkumulator. Für

المنارة للاستشارات

indizierte Variable mit laufenden Indizes in Schleifen ist eine solche Behandlung natürlich zeitraubend und ineffektiv; sie muß durch rekursive Auswertung der Speicherabbildungsfunktion ersetzt werden, bei der innerhalb der Schleife nur Additionen auftreten, die z. B. durch Indexregister erledigt werden können. Wir kommen darauf noch zurück.

5. Anweisungen (statements)

Die Auswertung vollständiger Anweisungen verläuft nach den gleichen Prinzipien wie die der Ausdrücke, die ja den wesentlichsten Teil aller Anweisungen darstellen. Es muß nur der Symbolkeller einige weitere Symbole aufnehmen können.

Was die arithmetischen (und Booleschen) Anweisungen anbetrifft, handelt es sich hier im wesentlichen um das :=, das stets als erstes im Symbolkeller abgesetzt wird und damit an Stelle des anfänglichen Leerzustands des Kellers tritt. Als Schlußzeichen im Informationseinlauf fungiert das Anweisungstrennzeichen ; bzw. das 'end' der zusammengesetzten Anweisungen, das jeweils erst die Setzung der letzten arithmetischen Operationen des Ausdrucks auslöst und bei Koinzidenz mit dem := anzeigt, daß dieses in den abschließenden Speicherbefehl umgesetzt werden kann.

Die verbalen Klammern 'begin' und 'end' für zusammengesetzte Anweisungen werden naturgemäß ebenso behandelt wie arithmetische Klammern: 'begin' wird in den Symbolkeller abgesetzt. Ein einlaufendes 'end' dient zunächst als Schlußzeichen für die vorangegangene Anweisung und löst die Veranlassung aller im Keller anstehenden Operationen aus, bis es auf das erste 'begin' stößt, das noch gelöscht wird. Damit ist die Funktion des 'end' beendet. Ist das nächste Zeichen wieder ein 'end', so wiederholt sich der Vorgang, bis als Schlußzeichen das Trennzeichen ; eintrifft, das die Abarbeitung des Kellers bis zum nächsten gekellerten 'begin' auslöst, das nun aber natürlich unangetastet bleibt.

Ähnlich ist die Situation bei der einfachen Sprunganweisung 'go to' L. Der führende Begrenzer wird im Keller abgesetzt, anschließend die Marke L ausgewertet. Das Trennzeichen ; schließt die Auswertung ab und zeigt beim Auf-

المنارة للاستشارات

treffen auf den Begrenzer im Keller, daß die zugehörige Operation „Sprung nach dem durch L bezeichneten Speicherplatz" abgesetzt werden kann.
Die Behandlung der beiden Anweisungen 'if' B und 'for' $V := l$, wo l eine Liste entweder von Ausdrücken E oder von Ausdruck-Tripeln $E_i\ (E_s)\ E_e$ darstellt, ist zunächst ähnlich wie die der Sprunganweisung. Der Begrenzer wird im Keller abgesetzt und die anschließende Zeichenfolge B bzw. $V := l$ ausgewertet. Das abschließende Symbol ; zeigt das Ende der Auswertung an. In beiden Fällen ist jedoch die Funktion des Begrenzers noch nicht abgeschlossen.

Im Falle des 'if' kann zwar die Absetzung des an die Aussage B anschließenden bedingten Sprungbefehls durch das ; veranlaßt werden. Jedoch ist die Sprungadresse noch unbekannt. Sie liegt erst fest, wenn die nächste Anweisung voll ausgewertet ist. Daher muß das 'if' als transformiertes 'if_1' im Keller verbleiben, bis es auf das nächste einlaufende Trennzeichen ; oder 'end' trifft, das das Ende der bedingten Anweisung markiert. Erst damit liegt das Sprungziel im erzeugten Programm fest und kann eingetragen werden, worauf das 'if_1' endgültig gelöscht wird.
Der Fall des 'for' ist wesentlich komplizierter. Besteht die Liste l in 'for' $V := l$; Σ (wo Σ die qualifizierte Anweisung darstellt) aus Ausdrücken E_1 bis E_k, so ist die Anweisung unter Einführung einer zusätzlichen Indexvariablen HI und einer indizierten Variablen $V[HI]$ in die folgenden Anweisungen umzusetzen:

$$V[1] := E_1;\ V[2] := E_2; \cdots;\ V[k] := E_k;$$
$$\text{'for'}\ HI := 1(1)k;$$
$$\text{'begin'}\ V := V[HI]\ ;\ \Sigma\ \text{'end'};$$

Damit ist dieser Fall auf den der Progression zurückgeführt. Ähnlich hätte man vorzugehen, wenn die Elemente der Liste l selbst Progressionen $E_i\ (E_s)\ E_e$ sind.

Einfacher ist in diesem Fall sicher, die Anweisungen 'for' $V := E_{ig}\ (E_{sg})\ E_{eg}$; Σ für jedes Listenelement getrennt aufzuschreiben. In jedem Fall aber genügt die Betrachtung der einfachen Progression: 'for' $V := E_i\ (E_s)\ E_e$; Σ.

Nach Kellerung des 'for' kann der erste Teil der folgenden Symbolkette $V := E_i$ wie eine normale arithmetische An-

المنارة للاستشارات

weisung betrachtet werden, da ja hierdurch der erste Wert von V festgelegt wird. Als Schlußzeichen, das auf das 'for' im Keller trifft, wirkt die öffnende Klammer. Ihr Zusammentreffen mit 'for' besagt, daß sie zu ersetzen ist durch $S:=$, was zusammen mit dem folgenden E_s wieder als arithmetische Anweisung ausgewertet werden kann. Die schließende Klammer wirkt als Schlußzeichen und ist zu ersetzen durch $E:=$, worauf wieder mit der Auswertung von E_e fortgefahren werden kann. S und E sind dabei vom Übersetzer einzuführende Hilfsvariable für Schritt und Endgröße. Eine Vereinfachung ist möglich, wenn E_s oder E_k eine Zahl oder eine einzige Variable ist: In diesem Falle genügt es, wenn der Übersetzer die Hilfsvariablen S bzw. E durch die betreffenden Größenbezeichnungen ersetzt[4]).

Da in der auf das 'for' folgenden Schleife die Abschlußbedingung von dem Vorzeichen des Wertes von E_s abhängt, muß dieses noch vor dem Eintritt in die Schleife getestet werden. Ist E_s eine Zahl, so kann dies der Übersetzer übernehmen. In anderen Fällen muß eine Prüfung der Laufrichtung und eine entsprechende Festlegung der Abschlußbedingung im Programm erzeugt werden, wenn man nicht dem Übersetzer sehr unbequeme dynamische Kontrollen aufbürden will.

Die Funktion des 'for' ist mit der durch das erste Semikolon angezeigten Abarbeitung der Progressionsangaben nicht erledigt. Vielmehr muß noch die Schleifenschließung einschließlich Zählung und Prüfung veranlaßt werden. Daher ist auch das 'for' im Keller durch das erste Semikolon zu transformieren zu 'for_1'. Benützt man als Standardschleife den normalerweise effektivsten Typ mit Prüfung am Schluß und Schließung durch bedingten Sprung, so ist die Absetzung der entsprechenden Operationen bis zum Ende der auf die 'for'-Anweisung folgenden Anweisung zurückzustellen. Da man aber dem Fall der leeren Schleife vom Typ 'for' $V := 1(1)0$ Rechnung tragen muß, ist vor dem Schleifenbeginn noch ein Sprung auf die Ausgangsprüfung der Schleife zu setzen. Diesem muß noch eine

---

[4]) Auf den dubiosen Fall, daß die Anweisung $S := E_s$ in die Schleife selbst aufgenommen werden muß, weil etwa E_s von V abhängt (etwa 'for' $V := 1\ (V)\ N$, was die Folge der ganzen Potenzen von 2 liefert), soll hier nicht weiter eingegangen werden. S ist also für die Schleife fest und in sinnvollen Fällen ungleich Null.

المنارة للاستشارات

Marke folgen, die als Ziel für den Schleifenschließungssprung dient.

Anschließend kann die dem 'for' unterliegende einfache und zusammengesetzte Anweisung abgearbeitet werden. Das abschließende Symbol ; oder 'end' führt beim Auftreffen auf das 'for_1' im Keller zur Absetzung der Schließungsbefehle:

Insgesamt ist also die 'for'-Anweisung

$$\text{'for'}\ V := E_i\ (E_s)\ E_e\ ;\ \Sigma\ ;$$

vom Übersetzer wie die aufgelöste Anweisungsfolge

$$V := E_i; \quad S := E_s; \quad E := E_e;$$

$$\neq := \begin{cases} \geqq & \text{falls } S < 0 \\ \leqq & \text{falls } S > 0 \end{cases};\ \text{'go to'}\ L_p;\ L_B\text{: } \Sigma;$$

$$V := V + S;\ L_p\text{: 'if' } V \neq E;\ \text{'go to'}\ L_B$$

$$V := V - S;$$

zu behandeln, wobei sich bei der zweiten, dritten und vierten Anweisung die diskutierten Vereinfachungen ergeben können.

Die Prozeduranweisung schließlich ist als Aufruf eines Bibliotheksprogramms von Standardform zu behandeln, wobei die Parameter in der im Aufruf angegebenen Reihenfolge abgesetzt werden[5]). Da es sich hierbei um bekannte Techniken handelt, sind weitere Ausführungen unnötig.

Die Anweisung 'return' behandelt einen einfachen Rücksprung auf eine eingebrachte Rückkehradresse. Die Anweisung 'stop' bedeutet (unwiderrufliches) Ende des betreffenden Programmlaufs und soll die Maschine in einen Zustand versetzen, in dem sie weitere Aufträge annimmt.

6. Vereinbarungen (declarations)

Von den Vereinbarungen sollen nur die Funktions-, Prozedur- und Feld-Vereinbarungen kurz behandelt werden[6]). Funktions- und Prozedur-Vereinbarungen sowie Feld-Ver-

---

[5]) Insbesondere darf angenommen werden, daß Ein- und Ausgabetätigkeiten, die weithin von den Maschinencharakteristika abhängig sind, in genereller Form als Prozeduren aufgerufen werden können.

[6]) Typ-Vereinbarungen sind in selbstverständlicher Weise bei der Behandlung arithmetischer (oder Boolescher) Ausdrücke zu berücksichtigen.

المنارة للاستشارات

einbarungen (solange man sich auf jeweils vollständige Auswertung der Speicherabbildungsfunktion, siehe 7., beschränkt) definieren jeweils Unterprogramme. Funktions- und Feld-Vereinbarungen führen zu statischen Programmen, die Prozedur-Vereinbarungen dagegen zu dynamischen[7]). Dem aus der Auswertung der Vereinbarung resultierenden Unterprogramm ist also wie bei allen Unter- bzw. Bibliotheksprogrammen ein Anschlußteil voranzustellen, der die Übernahme der Rückkehradresse und der Programmparameter durchführt, im dynamischen Fall ist ein Adaptieren zur Berechnung des benötigten Hilfsspeichers und zur Adressierung der auf den Hilfsspeicher bezüglichen Befehle (mit Hilfe eines speziellen Parameters, der den Beginn des freien Speichers angibt) hinzuzufügen. Auch hier handelt es sich um bekannte Techniken, auf die nicht näher eingegangen zu werden braucht.

7. Adressenfortschaltung bei indizierten Variablen

Wie bereits erwähnt, fügen sich die indizierten Variablen ohne Schwierigkeiten in den Rahmen der diskutierten Übersetzungstechnik, solange man die Speicherabbildungsfunktion im Programmlauf jeweils in geschlossener Form auswertet. Tatsächlich ist ja etwa die Größe $a\,[i,k]$ mit zugehöriger Feld-Vereinbarung 'array' ($a\,[1,1:n,m]$) als Funktion gegeben durch

$$a\,[i,k] := \langle\, k \times n + i + \,\rangle\, a\,[0,0]\,\langle\,\rangle\,.$$

Hier stellt die Variable $\rangle a \langle$ die Zahl dar, die die Adresse des die Zahl a enthaltenden Speicherplatzes angibt. Die Funktion $\langle E \rangle$ hat als Wert diejenige Zahl, die auf dem durch den (ganzzahligen) Wert von E adressierten Speicherplatz steht[8]).

Für Variable mit Laufindex in inneren Schleifen bedeutet eine solche Auswertung jedoch einen unerträglichen Zeitverlust, weshalb die Adressenberechnung stets rekursiv

---

[7]) Die Prozedurvereinbarung liefert auch die Möglichkeit, in ALGOL dynamische (Bibliotheks-) Programme zu formulieren.

[8]) Beide Elemente sind in ALGOL nicht enthalten, dürften im übrigen nahezu ausreichen, um ALGOL in die oft diskutierte universal computer language UNCOL zu verwandeln.

المنارة للاستشارات

vorgenommen wird, insbesondere in der innersten Rekursionsstufe mit Hilfe von Indexregistern. Prinzipiell ist auch hier klar, was getan werden muß, sogar bei Zulassung allgemeinerer Speicherabbildungen:

$$a[i,k] := \langle\,\rangle\, a[0{,}0] \langle + P(i,k,p_j)\rangle$$

wobei p_j weitere Parameter wie n, m darstellt und P eine beliebige Funktion ist.

Tritt nun in einer Schleife mit der Laufvariablen V die indizierte Variable $a\ [f_1(V), f_2(V)]$ auf, deren Indexpositionen mit Funktionen f_1 und f_2 der Variablen V besetzt sind, so hat man in der Abbildungsfunktion einzutragen

$$\begin{aligned} a[f_1(V), f_2(V)] &:= \langle\,\rangle\, a[0{,}0] \langle + P(f_1(V), f_2(V), p_i)\rangle \\ &= \langle\,\rangle\, a[0{,}0] \langle + Q(V, p_i)\rangle \end{aligned}$$

und die entstehende Funktion $Q\ (V, p_i)$ durch eine Rekursion hinsichtlich V auszudrücken, mit deren Hilfe die Abbildungsfunktion ohne Multiplikationen ausgewertet werden kann. Es muß demnach zur Adressenfortschaltung die sukzessive Bildung des vollständigen Differenzenschemas von $Q\ (V, p_i)$ veranlaßt werden.

In praxi bedeutet das eine ungeheure Komplikation, da der Übersetzer das Schema für die Bildung beliebiger Rekursionen in sich tragen muß. Da auch der allgemeine Fall äußerst selten vorkommt (ein nicht triviales Beispiel ist jedoch die Dreieckspeicherung dreieckiger Matrizen), ist bereits in ALGOL nur rechteckige Speicherung von Feldern, d. h. in den Indizes lineare Abbildungsfunktion, unterstellt. Ferner hat man bisher auch die auf Indexpositionen zulässigen Ausdrücke auf in dem Laufindex lineare Funktionen beschränkt. In ALGOL ist eine solche Beschränkung nicht vorgesehen, dementsprechend haben wir bei der geschlossenen Auswertung beliebige (auch indizierte) Indexausdrücke zugelassen. Bei der Adressenfortschaltung beschränken wir uns jedoch ebenfalls auf den Fall linearer Indexausdrücke. Für rekursive Auswertung der Speicherabbildungsfunktionen kommen also nur indizierte Variable der Form $a\ [i \times c_1 + E_1', i \times c_2 + E_2', \ldots, i \times c_k + E_k']$ in Frage, wobei i der Laufindex der betreffenden Schleife sei, die c_j seien Konstante und die E_j' Ausdrücke, die i nicht

المنارة للاستشارات

enthalten. Aus der oben angegebenen Speicherabbildungsfunktion (Fall $k = 2$) ergibt sich, wenn wir abkürzend $\rangle a\,[i \times c_1 + E_1', \dots, i \times c_k + E_k']\langle$ durch A und $\rangle a\,[0,0]\langle$ durch A *null* ersetzen

(1) $\quad A := (E_i \times c_2 + E_2') \times n \dotplus E_i \times c_1 + E_1' + A\ \mathit{null}$

(2) $\quad A := (c_2 \times n + c_1) \times S \dotplus A$

als Rekursion für die Adressen der durch die Werte $i := E_i(S)E_e$ ausgewählten Komponenten $a\,[i \times c_1 + E_1', \dots, i \times c_k + E_k']$ des Feldes $a\,[,]$. Der Wert für n ist dabei der zugehörigen Feld-Vereinbarung zu entnehmen, der Wert von A *null* ist vom Übersetzer aus der Speicherverteilung, die nach Abschluß der eigentlichen Übersetzung des Formelprogramms an Hand der Feld-Vereinbarungen in üblicher Compilertechnik auszuführen ist, zu berechnen und dem erzeugten Programm als Konstante einzuverleiben.

Das ursprüngliche Formelprogramm laute etwa

'for' $i := E_i(S)E$;
$\quad \dots a\,[c_1 \times i + E_1', c_2 \times i + E_2'] \dots$;

Es ist vom Übersetzer zu behandeln wie: ($S > 0$ vorangesetzt)

$i := E_i$;
$A := (i \times c_2 + E_2') \times n + i \times c_1 + E_1' + A\ \mathit{null}$;
delta $A := (c_2 \times n + c_1) \times S$;
'go to' L_p;
L_B: $\dots \langle A \rangle \dots$;
$A := A +$ *delta* A;
$i := i + S$;
L_p: 'if' $i \leqq E$; 'go to' L_B

Das Symbol $\langle A \rangle$ ist dabei zu interpretieren als:
man rechne mit dem Inhalt der durch die Zahl A als Adresse bezeichneten Zelle, d. h. als übliche indirekte Adresse, wenn man nicht diese Zahl A als Adresse in dem Rechenbefehl substituiert.

المنارة للاستشارات

Stehen Indexregister für die Adressenfortschaltung zur Verfügung, so ist die Sequenz abzuändern. Wir können formal auch ein Indexregister mit einer Variablen *IRK* (Indexregister K) bezeichnen. Ist eine Variable mit *IRK* indiziert (*a* [*IRK*]), so bedeutet dies, daß der Übersetzer demjenige erzeugten Befehl, der die dieser Variablen entsprechende Adresse enthält, das Merkmal für Adressenmodifikation durch Addition des Indexregisters K anfügen muß.

Mit diesen Abkürzungen ist der oben angegebene Formelausschnitt vom Übersetzer zu behandeln wie

$$
\begin{aligned}
& i := E_i; \quad \mathit{delta}\ A := (c_2 \times n + c_1) \times S; \\
& A := E_2' \times n + E_1' + A\ \mathit{null};\ IRK := (c_2 \times n + c_1) \times i; \\
& \text{'go to' } L_p; \\
& L_B: \ldots \langle A \rangle\ [IRK] \ldots; \\
& IRK := IRK + \mathit{delta}\ A; \\
& i := i + S; \\
& L_p: \text{'if' } i \leqq E; \text{'go to' } L_B;
\end{aligned}
$$

Das allgemeine Schema zeigt bereits, daß bei der Adressenfortschaltung das Kellerungsprinzip durchbrochen wird. Denn erst das Erscheinen der indizierten Variablen im Inneren der Schleife zeigt dem Übersetzer, daß er in den bereits erzeugten Teil des Programms noch Befehlsserien einzuschieben hat. Er muß also zwei Programmteile, die Befehle der Schleife selbst und den für die Fortschaltungen notwendigen Vorbereitungsteil, gleichzeitig nebeneinander aufbauen und kann sie erst nach Abschluß der Schleife aneinanderfügen, wobei die Reihenfolge Geschmackssache ist, solange dem für das erzeugte Programm erforderlichen zeitlichen Ablauf Rechnung getragen wird.

Abgesehen davon liegt hier nun ein Fall vor, in dem die Symbolfolge nicht mehr sequentiell mit einfacher Kellerung abgearbeitet werden kann: Die auf den Indexpositionen stehenden Ausdrücke müssen in mehrere parallele Züge auseinandergefahren werden, da neben der vollständigen Auswertung zum Aufbau der Speicherabbildungsfunktion, die den Anfangswert von *A* bzw. *A* und *IRK* festlegt, noch zur Festlegung des Programms für die Berechnung von *delta A* vom Übersetzer die Koeffizienten der Laufvariablen

المنارة للاستشارات

i auf allen Indexpositionen zu sammeln und mit den richtigen Faktoren, die aus der Feld-Vereinbarung stammen, zu versehen sind.

Mit der Fortschaltung verknüpft sind eine Reihe von notwendigen Kontrollen und Vergleichen, die für den Übersetzer stets das Anlegen und Durchsehen von speziellen Listen bedeuten. Zunächst ist, als Voraussetzung für die Fortschaltung, vom Übersetzer die Linearität der Indexausdrücke festzustellen. Vor allem muß sichergestellt sein, daß nicht etwa die Koeffizienten eines in der Laufvariablen formal linearen Indexausdrucks Variable enthalten, die in der Schleife umgerechnet werden und damit von den Werten der Laufvariablen abhängen. Das bedeutet aber, daß alle in der betreffenden Schleife als Rechenergebnisse links von dem Symbol := in arithmetischen Anweisungen auftretenden Variablen vom Übersetzer notiert werden müssen, um gegebenenfalls mit einer in einem Indexausdruck auftretenden Variablen, die nicht als Laufvariable eines 'for'-Symbols definiert ist, verglichen werden zu können. Diese Kontrolle ist unumgänglich, da nichtlineare Indizes zugelassen sind, aber nicht fortgeschaltet werden können.

Zur richtigen Behandlung der Indexausdrücke muß der Übersetzer in jedem Augenblick die Laufvariable der gerade abgearbeiteten Schleife greifbar haben. Dies wird erreicht mit Hilfe eines neuen Kellers, des Schleifenkellers, in dem jedes Laufvariablensymbol beim Auftreten nach dem zugehörigen 'for' abgesetzt wird und aus dem es erst beim Schleifenende, das durch das Zusammentreffen des Endzeichens ; oder end mit dem transformierten 'for_1' angezeigt wird, wieder entfernt wird.

Um das erzeugte Programm so kurz und damit so effektiv wie möglich zu machen, muß der Übersetzer weiterhin eine Reihe von Identitätsprüfungen vornehmen. Zunächst ist, beim Auftreten mehrerer indizierter Variablen in einer Schleife, die mögliche Identität der zugehörigen Fortschaltgrößen *delta A* festzustellen. Zwar führen solche Identitäten, solange ohne Indexregister gearbeitet wird, nur zur Verkürzung des Vorbereitungsteils der Schleife und zur Einsparung von Hilfsspeicherzellen. Beim Einsatz von Indexregistern aber gewinnt man mehr. Denn da mit den

المنارة للاستشارات

delta A auch die Anfangseinstellungen der *IRK* identisch sind, kann der Übersetzer alle identischen Fortschaltungen mit einem Indexregister ausführen lassen, und man spart sowohl Fortschaltungsrechnungen in der Schleife als auch Indexregister ein. In den meisten vorkommenden Fällen erscheint die Laufvariable i einer Schleife nur in Indexausdrücken. Außerdem (und eben aus diesem Grunde) verfügen die meisten Maschinen mit Indexregistern über einen speziellen bedingten Sprung, der vom Inhalt eines Indexregisters und evtl. einer anderen Speicherzelle abhängig ist. Um dies auszunützen, muß der Übersetzer kontrollieren, ob die Laufvariablen außerhalb von Indexausdrücken vorkommt. Ist dies nicht der Fall, dann kann die laufende Berechnung der Laufvariablen ganz entfallen und durch die Indexregisterfortschaltung ersetzt werden. Entsprechend ist die Schlußbedingung auf das Indexregister umzustellen, weshalb im Vorbereitungsteil der Endwert E von i mit dem Koeffizienten von S in *delta A* zu multiplizieren ist.

Unser obiges Beispiel hat der Übersetzer dann zu behandeln wie:

$$
\begin{array}{ll}
\textit{delta } A & := c_2 \times n + c_1; \\
IRK & := (E_i) \times \textit{delta } A; \\
E & := E \times \textit{delta } A; \\
\textit{delta } A & := S \times \textit{delta } A; \\
A & := E_2' \times n + E_1' + A \textit{ null}; \\
\multicolumn{2}{l}{\text{'go to' } L_p;} \\
\multicolumn{2}{l}{L_B: \ldots \langle A \rangle [IRK] \ldots \langle A' \rangle [IRK] \ldots;} \\
IRK & := IRK + \textit{delta } A; \\
\multicolumn{2}{l}{L_p: \text{'if' } IRK \leqq E; \text{'go to' } L_B;}
\end{array}
$$

Die Möglichkeiten zur Vereinfachung von Schleifen sind damit natürlich noch nicht erschöpft. Jedoch ist auf das Wesentliche hingewiesen und die Diskussion soll damit abgeschlossen werden.

Wie bereits erwähnt, ist die Adressenfortschaltung von besonderer Bedeutung in den innersten Schleifen eines Programms, und an diesen Stellen sollten die Indexregister in erster Linie zur Fortschaltung eingesetzt werden. Die

المنارة للاستشارات

Tatsache, daß eine Schleife innerste Schleife ist, ist jedoch erst am Schleifenende festzustellen. Der Übersetzer muß daher die endgültige Absetzung der entsprechenden Befehle bis zum Erscheinen des die Schleife abschließenden Trennzeichens verschieben. Weiter ist im Schleifenkeller eine Anmerkung „nicht innerste Schleife“ bei jedem gekellerten Laufindex, für den dies zutrifft, notwendig.

Schlußbemerkung

Die vorangegangene Darstellung zeigt, daß die Umsetzung des Formelprogramms in Maschinenoperationen durch den Übersetzer mit Ausnahme der Behandlung der Adressenfortschaltung zügig ohne Abspeicherung des Formelprogramms, also als reiner Eingabeprozeß, durchgeführt werden kann. Denn die Fernzusammenhänge im Programm beschränken sich auf Adressen, die aus während des Einleseprozesses anzulegenden Listen entnommen werden können. Dementsprechend ist es auch möglich, die Umsetzungsmethode zur sofortigen interpretativen Ausführung der im Formelprogramm angegebenen Operationen, auch mit Hilfe verdrahteter Schaltungen, anzuwenden. Eine Adressenfortschaltung macht allerdings hierbei außerordentliche Schwierigkeiten.

Literatur

[1] *H. Goldstine* and *J. von Neumann:* Planning and Coding for an Electronic Computing Instrument. Institute for Advanced Study, Princeton, N. J., 1947/48.

[2] International Business Machines Corp., FORTRAN Manual.

[3] *W. S. Melahn:* A description of a cooperative venture in the production of an automatic coding system, Journ. Assoc. Comp. Mach. 3 (1956), S. 266—271.

[4] *H. Rutishauser:* Über automatische Rechenplanfertigung bei programmgesteuerten Rechenanlagen, Z. Angew. Math. Mech. 31 (1951), 255.

[5] *H. Ruthisauser:* Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen, Mitt. Inst. f. Angew. Math. der ETH Zürich, Nr. 3, (1952).

المنارة للاستشارات

[6] *K. Samelson:* Probleme der Programmierungstechnik. Intern. Kolloquium über Probleme der Rechentechnik, Dresden 1955, S. 61—68.

[7] *P. B. Sheridan:* The arithmetic translator-compiler of the IBM Fortran automatic coding system, Com. ACM Bd. 2, (1959) 2, S. 9—21.

[8] *M. V. Wilkes; D. J. Wheeler; S. Gill:* The preparation of programmes for an electronic digital computer (Cambridge, Mass., 1951).

[9] *M. V. Wilkes:* The Use of a Floating Address System for Orders in an Automatic Digital Camputer. Proc. Cambridge Philos. Soc. 49, (1953) 84—89.

[10] *Charles W. Adams* and *J. H. Laning jr.:* The M. I. T. System of Automatic Coding; Comprehensive, Summer Session and Algebraic, in: Symposium on Automatic Programming for Digital Computers, US Department of Commerce.

[11] ACM Committee on Programming Languages and GAMM Committee on Programming: Report on the Algorithmic Language ALGOL, edited by *A. J. Perlis* and *K. Samelson.* Num. Math. 1 (1959), S. 41—60.

[12] *H. Zemanek:* Die algorithmische Formelsprache ALGOL. Elektron. Rechenanl. 1 (1959), S. 72—79 und S. 140—143.

[13] *F. L. Bauer:* The Formula Controlled Logical Computer Stanislaus, erscheint in Math. Tabl. Aids Comp.

[14] *C. Böhm:* Calculatrices digitales. Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme (Dissertation, Zürich 1952) Annali dei Matematica pura ed applicata. Ser. 4, 37 (1954), S. 5—47.

المنارة للاستشارات

Friedrich L. Bauer

Ausschnitt aus der Patentschrift:
Verfahren zur automatischen Verarbeitung
von kodierten Daten und Rechenmaschinen
zur Ausübung des Verfahrens

Auslegeschrift 1 094 019
Anmeldetag: 30. März 1957

المنارة للاستشارات

BUNDESREPUBLIK DEUTSCHLAND

KL. 42 m 14
INTERNAT. KL. G 06 f

AUSLEGESCHRIFT 1 094 019

B 44122 IX/42m

ANMELDETAG: 30. MÄRZ 1957

BEKANNTMACHUNG DER ANMELDUNG UND AUSGABE DER AUSLEGESCHRIFT: 1. DEZEMBER 1960

Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens

Anmelder:

Dr. Friedrich Ludwig Bauer,
München, Pörtschacherstr. 40,
und Dr. Klaus Samelson,
München, Hiltenspergerstr. 19

Dr. Friedrich Ludwig Bauer und Dr. Klaus Samelson,
München,
sind als Erfinder genannt worden

1

Die Erfindung betrifft ein Betriebsverfahren für automatische mechanische, elektrische oder elektronische Rechenmaschinen und bezieht sich insbesondere auch auf den technischen und logischen Aufbau der Rechenmaschine sowie der damit in Verbindung stehenden Eingabe- und Ausgabevorrichtungen.

Die bekannten Rechenautomaten und Datenverarbeitungsanlagen erfordern im Einzelfall Anweisungen über die Art und den Ablauf der numerischen oder sonstigen informationsverarbeitenden Prozesse. Die Schreibweise, in der diese Anweisungen fixiert werden, wurde zu Beginn der Entwicklung so gewählt, daß sie gewisse als elementar erachtete technische Funktionen der Anlage beschrieb. Die so geschriebenen Anweisungen werden üblicherweise »Programm« genannt. Das Programm für einen Rechenprozeß etwa und die mathematische Formel, mit der der Mathematiker diesen Prozeß gewöhnlich beschreibt, kennzeichnen jeweils genau denselben Vorgang, allerdings in zwei grundverschiedenen Sprachen.

Die Übersetzung von der mathematischen Formelsprache ins Programm wird üblicherweise Programmierung genannt; sie hat sich in praxi als eine zeitraubende und fehleranfällige, im allgemeinen nur lästige Angelegenheit herausgestellt. Für den Mathematiker stellt die Programmierungssprache eine ungewohnte Formulierung dar, die überdies noch von Anlagentyp zu Anlagentyp wechselt. Diese bei den meisten bestehenden Maschinen jeweils verschiedene Art der Programmschreibweise zeigt bereits, wie sehr das Befehlssystem üblicher Maschinen noch von der verwendeten Technik abhängt und wie wenig die auf der ganzen Welt einheitliche mathematische Formelsprache von den Rechenautomatenbauern bisher ernst genommen wurde.

Die Mängel der üblichen Programmierung sind in der Literatur bereits vor einigen Jahren klar erkannt worden. Man ist jedoch den zunächst naheliegenden Weg gegangen, vorhandene Rechenautomaten universeller Art zu gewissen Routinearbeiten der Programmierung, die selbst Datenverarbeitungsaufgaben darstellen, heranzuziehen. Es gibt heute bereits Programme, die unter gewissen Einschränkungen die ganze Übersetzungsarbeit von einer mathematischen Formel bis zum Programm für einen üblichen Rechenautomaten erledigen.

Die Übersetzungsprogramme sind sehr kompliziert aufgebaut und dementsprechend umfangreich. Kleinere Rechenanlagen sind nicht mehr in der Lage, solche Aufgaben durchzuführen. Umfangreiche Formeln zu übersetzen, führt auch bei mittelgroßen Anlagen zu übermäßig hohem Zeitbedarf.

Demgegenüber ist es von Bedeutung, daß durch geschickte Organisation des Zusammenwirkens geeigneter Einzelkomponenten, gestützt auf grundsätzliche Studien über das Wesen von Rechnungsabläufen, unter Verwendung neuartiger Maschinenfunktionen und -steuerungsabläufe sowie Anlagenteile ein Rechenautomat gebaut werden kann, der unmittelbar durch mathematische Formeln in üblicher Schreibweise gesteuert wird, also ein formelgesteuerter Rechenautomat, der in seinem technischen Aufbau und in seiner praktischen Verwendungsmöglichkeit gegenüber den programmgesteuerten Rechenanlagen bisheriger Art einen wesentlichen Fortschrit darstellt.

2

Eine solche Rechenmaschine muß außer den bekannten, mehr oder weniger üblichen Teilen eine Vorrichtung besitzen, die diese mathematischen Formeln in üblicher Schreibweise analysiert und eine entsprechende Folge von Steuerbefehlen löst. Dabei ergeben sich im einzelnen auch neuartige Lösungen für die Erledigung gewisser Rechenabläufe in Anpassung an diese besondere Art der Verarbeitung der mathematischen Formeln.

Die Erfindung beruht im wesentlichen auf dem Gedanken, den Komponenten einer Rechenmaschine einen Analysator beizuordnen, dem die mathematischen Formeln in üblicher Schreibweise zugeführt werden. Gemäß der Erfindung werden die den einzelnen Zeichen entsprechenden Signale in der Reihenfolge der Aufschreibung dem Analysator zugeführt und in diesem entsprechend der Reihenfolge des Eingangs geprüft, ob die Operationen sofort ausführbar sind oder ob der Eingang weiterer Signale abgewartet werden muß; in diesem letzteren Falle werden die noch nicht verarbeitbaren Zeichen in einen Speicher (Keller) eingeführt; beim Eintreffen neuer Zeichen im Analysator, die die

009 650/226

المنارة للاستشارات

Ausführung einer Operation mit gespeicherten Zeichen ermöglichen, werden diese gespeicherten Zeichen in der durch die Art der Einführung festgelegten, umgekehrten Reihenfolge entnommen und verarbeitet.

Der Analysator im engeren Sinne enthält einen Formelentschlüsseler, der als Bestandteil einen Vorentschlüsseler, einen Formelumsetzer, einen Ziffernumsetzer und ein Ausgabesteuerwerk aufweist, sowie eine Hilfssteuereinrichtung zur Verarbeitung von Indizes; der Analysator im weiteren Sinne umfaßt auch einen Größenvorspeicher und einen Kennzeichenentschlüsseler.

Bei der Ausführung des Verfahrens kann auch eine weitergehende Zurückstellung der zugeführten Signale erfolgen; sie ist aber nicht notwendig und kann nur durch andere, außerhalb der Erfindung liegende Vorteile gerechtfertigt werden.

Gemäß einer weiteren Ausführungsform der Erfindung werden diejenigen Formelzeichen, welche Ziffernsymbole, also Zahlen, darstellen, von solchen Formelzeichen, welche Operationssymbole darstellen, getrennt und, sofern sie zurückgestellt werden müssen, speicherfähigen Vorrichtungen, vorzugsweise zwei verschiedenen »Kellern«, nämlich dem Zahlkeller und dem Operationskeller, zugeführt und von diesen Vorrichtungen her dem Steuerwerk zugänglich gemacht.

Dabei ist es zweckmäßig, die in dem Zahlkeller bzw. dem Operationskeller neu eintreffenden Zeichen jeweils an die Spitze der entsprechenden Sequenz zu setzen und die Entnahme eines Zeichens automatisch durch Wegnahme von der Spitze der entsprechenden Sequenz vorzunehmen.

Ein Ausführungsbeispiel für das Verfahren in seiner einfachsten Form wird als Stufe I im folgenden näher beschrieben. Es ist zur Verarbeitung einfachster Formelausdrücke geeignet. Die Formelsymbole der Arithmetik $+ - \sqrt{\ } \times : ( )$ werden dabei vorzugsweise in Form von Kodezeichen zur Auslösung von Steuerungsabläufen, d. h. von Maschinenfunktionen, benutzt. Der Zeitpunkt der endgültigen Auslösung des Steuerungsablaufes durch ein Kodezeichen hängt unter Umständen, z. B. bei der Klammer, davon ab, daß ein oder mehrere nachfolgende Kodezeichen eintreffen. Aus diesem Grund werden die Kodezeichen zunächst in dem Operationskeller zurückgestellt und erst dann, wenn der Ausführungszeitpunkt eintritt, dem Operationskeller wieder entnommen, wobei die Darbietung bei der obenerwähnten sequentiellen Aufreihung automatisch die richtige ist. In ähnlicher Weise wird durch den Zahlkeller dafür gesorgt, daß die vorzunehmenden Rechenoperationen automatisch mit den jeweils dafür in Frage kommenden Zahlen vorgenommen werden, sobald alle für die Ausführung der Operation erforderlichen Zahlenwerte vorhanden sind. Die Zahlen, die in einer Rechnung Verwendung finden sollen, können durch Ziffernsymbole und entsprechende Kodezeichen, z. B. im Dezimalsystem, dargestellt werden.

Solche Formelzeichen, welche ein Resultat verlangen, insbesondere das Gleichheitszeichen, werden einer besonderen Vorrichtung, nämlich der Ausgabesteuerung, zugeführt.

Im folgenden wird auf ein weiteres Ausführungsbeispiel des Verfahrens eingegangen, die als Stufe II bezeichnet wird.

In vielen Fällen ist es erwünscht, solche Teilergebnisse, die sich wiederholen, nur einmal zu berechnen und sie in der mathematischen Schreibweise durch besondere Symbole, z. B. Buchstaben, zu bezeichnen. Gemäß der weiteren Erfindung werden zur formelartigen Benennung von Zahlen oder Zahlsätzen, z. B. Ausgangsdaten und Teilergebnissen, besondere Zeichen als algebraische Größen-

symbole und zugehörige Kodezeichen benutzt, derart, daß jedes Zeichen beim erstmaligen Einlaufen in das Steuerwerk eine Reservierung von an sich beliebigen Plätzen für eine Zahl oder einen Zahlsatz in einem an sich bekannten Speicher veranlaßt, wobei eine Zuordnung zwischen diesem Platz bzw. diesen Plätzen und dem betreffenden Größensymbol bis auf Widerruf festgehalten wird. Das Größensymbol wird in Formeln vom Steuerwerk stellvertretend für die auf dem zugeordneten Platz bzw. Plätzen gespeicherte Zahl bzw. Zahlsatz behandelt.

Die Belegung eines mit einem Größensymbol bezeichneten Platzes im Zahlspeicher durch eine Zahl erfolgt durch das Ergibtsymbol ⇒ und Angabe der Größe. Es kann vorteilhaft sein, die obenerwähnte Platzreservierung erst zusammen mit der eben besprochenen Platzbelegung durchzuführen. Eine Zurückstellung des Rechenvorganges erfolgt, wenn innerhalb einer Formel ein Größensymbol ins Steuerwerk gelangt, für das noch keine Platzbelegung im Zahlspeicher vorgenommen war, wobei von außen neue Information so lange verlangt wird, bis der zu reservierende Platz nunmehr durch eine Zahl besetzt worden ist.

Durch die Einführung der Buchstaben als Zeichen ist bereits hier die Möglichkeit gegeben, die Ausgangsgrößen einer Rechnung von vornherein mit Buchstaben zu bezeichnen, so daß die mathematischen Formeln ganz oder teilweise mit algebraischen Zeichen geschrieben werden können.

Eine weitere Ausgestaltung der Rechenanlage ist durch Maßnahmen gegeben, die im folgenden beschrieben und als Stufe III bezeichnet werden. Während die bisher beschriebene Ausführung bereits eine Rechenanlage mit direkter Formelsteuerung und Niederschrift des gesamten Ablaufs, d. h. der Eingabe und Resultate, in mathematischer Schreibweise ermöglicht, ist es häufig erwünscht, die Möglichkeit der Wiederholung von Formeln ausnutzen zu können. Zu diesem Zweck wird die gesamte einlaufende, nach wie vor der direkten Formelsteuerung dienende Information gleichzeitig nebenher in einem Formelspeicher gespeichert. Zur Erschließung weiterer Möglichkeiten werden zur Numerierung von Formelgruppen besondere Zeichen als Kennzeichnungssymbole benutzt, derart, daß jedes Kennzeichnungssymbol beim erstmaligen Einlaufen in das Steuerwerk bewirkt, daß die Zuordnung zwischen dem Platz, den der Anfang der Formelgruppe in einem Formelspeicher einnimmt, und dem Kennzeichnungssymbol bis auf Widerruf festgehalten wird.

Es ist insbesondere möglich, die Anfänge von Formelgruppen zu kennzeichnen, wobei das Kennzeichnungssymbol etwa bestehen kann aus Ziffern mit Beifügung eines speziellen Zeichens, für das hinfort * benutzt wird. Eine laufende Durchnumerierung soll nicht erforderlich sein. Dabei ist es lediglich notwendig, vor dem Formelspeicher einen Vorspeicher anzuordnen, worin unter der Nummer jedes Kennzeichnungssymbols als Eingang derjenige Platz des dahinterliegenden Hauptspeichers, der mit dem betreffenden Kennzeichnungssymbol gekennzeichnet ist, festgehalten ist.

Das Verfahren kann weiterhin so ausgebildet werden, daß bereits die Angabe eines Kennzeichnungssymbols in Verbindung mit einem speziellen Zeichen, z. B. einem →, als Sprungsymbol genügt, um zu bewirken, daß die Rechnung wiederholt wird, allgemeiner, daß sie mit dem Beginn der unter dem betreffenden Kennzeichen im Formelspeicher notierten Formelgruppe fortgesetzt wird, wobei der Übergang in bekannter Weise von Bedingungen abhängig sein kann. Eine Zurückstellung des Rechenvorganges erfolgt, wenn ein Sprungsymbol auf eine noch nicht im Formelspeicher notierte Formelgruppe führt, wobei ebenfalls von außen neue Information verlangt

المنارة للاستشارات

wird. In diesem wie auch in dem obenerwähnten Fall, daß ein Größensymbol in einer Formel erscheint, das noch keine Belegung im Speicher hat, wird die verlangte Information im Formelspeicher lediglich notiert unter Festhaltung der durch Kennzeichnungssymbole bezeichneten Plätze der Anfänge einzelner Formelgruppen im Formelspeicher. Dieser Vorgang bricht automatisch ab, wenn das Kennzeichnungssymbol der aufgerufenen Formelgruppe ausgewertet wird bzw. wenn die aufgerufene Größe mit einer Zahl belegt worden ist, wobei der Rechenvorgang an der Unterbrechungsstelle wieder einsetzt, insbesondere im letzterwähnten Fall der Sprung ausgeführt wird. Bei Ausnutzung dieses Verfahrens erfolgt eine Zurückstellung des Rechenvorganges, auch wenn das zeitlich zuletzt im Formelspeicher notierte Formelzeichen abgearbeitet ist, ohne daß es einen Sprung auf ein schon vorhandenes Kennzeichnungssymbol bewirkt. Die Meldung, daß der Formelspeicher abgearbeitet ist, bewirkt dann, daß das Steuerwerk von außen neue Information verlangt, die im Formelspeicher notiert und gleichzeitig ausgeführt wird.

An Stelle des bisher verfolgten Prinzips der baldmöglichen Ausführung aller Verarbeitungsvorgänge von Formelsymbolen kann auch wahlweise eine weitere ganze oder teilweise Zurückstellung bis zu einem geeigneten späteren Zeitpunkt vorgenommen werden.

Ein besonderes Zeichen kann als Symbol »Nicht notieren« interpretiert werden, so daß anschließend die Notierung der von außen einlaufenden Information bis auf Widerruf, z. B. durch ein auflösendes Symbol oder das nächste einlaufende Kennzeichnungssymbol für Formelgruppen, unterdrückt wird.

Die Maschine zur Ausführung des Verfahrens enthält in ihrer einfachsten Ausführungsform einen Vorentschlüßler, dem sämtliche Formelzeichen in der Reihenfolge der üblichen Schreibweise zugeführt werden und der mehrere Ausgänge aufweist, die zu einem Ziffernumsetzer, zu einem Operationsumsetzer und zu einem Ausgabesteuerwerk sowie zu einem Steuerwerk für »bedeutungslose Zeichen« führen. Die Umsetzer der Steuerwerke können mit dem Schreibwerk in Verbindung stehen.

Der Zahlkeller und der Ziffernumsetzer sind derart verbunden, daß der Zahlkeller die Zahlen in der Reihenfolge des Eintreffens von dem Umsetzer abnehmen kann und daß ferner die jeweils erste in der Sequenz stehende Zahl beim Eintreffen eines entsprechenden Befehles über das Ausgabesteuerwerk als Ergebnis dem Schreibwerk zugeführt wird.

Der Operationsumsetzer steht mit dem Operationskeller in Verbindung, so daß er in diesen die Operationssymbole in der Reihenfolge ihres Eintreffens einspeisen kann, wobei das jeweils zuletzt eingespeiste der von unten nachrückenden, früher eingespeisten Symbole bzw. das neu ankommende Symbol an das Rechenwerk abgegeben werden kann, um solche Operationen auszuführen, für die die zugehörigen Operanden an der Spitze der im Zahlkeller befindlichen Sequenz vorliegen.

Es ist zweckmäßig, daß solche Zeichen, die eine Formel abschließen, insbesondere das Gleichheitszeichen oder das Ergibtzeichen, eine Prüfung auf »sinnvolle Formel« veranlassen.

Der Operationskeller und der Zahlkeller können Einrichtungen aufweisen, die die zugeführten Zeichen dadurch sequentiell speichern, daß sie die bereitsgespeicherten Zeichen in der Reihenfolge des Eintreffens nach unten weiterschieben und eine Abgabe nur des jeweils zuletzt gespeicherten oder des obersten der von unten nachzuschiebenden Zeichen gestatten.

Bei einer anderen Ausführungsform weist der Operationskeller und/oder der Zahlkeller Einrichtungen auf, die die zugeführten Zeichen dadurch sequentiell speichern, daß jedes eintreffende Zeichen auf den Platz vor dem zuletzt eingetroffenen gesetzt wird, daß dieser Platz festgehalten wird und daß ferner die Abnahme von dem zuletzt festgehaltenen Platz erfolgt.

Das Rechenwerk verarbeitet die im obersten oder in den beiden obersten Geschossen des Zahlkellers befindlichen Zahlen entsprechend den von der Operationssteuerung erhaltenen Anweisungen und gibt das Ergebnis wieder an das oberste Geschoß des Zahlkellers ab.

Das Ausgabesteuerwerk ist vorzugsweise mit dem Zahlkeller derart verbunden, daß beim Eintreffen eines Gleichheitszeichens und gegebenenfalls nachfolgender Zeichen »Ziffer verlangt« die Verbindung des Zahlkellers mit dem Schreibwerk hergestellt und die im obersten Geschoß des Zahlkellers befindliche Zahl ganz oder teilweise an das Schreibwerk abgegeben wird.

Mit der angegebenen Maschine kann auch mit Indexgrößen gerechnet werden. Die Rechnung mit Indexgrößen erfolgt dabei ganz analog wie das Rechnen mit den übrigen Rechengrößen. Zur Bezeichnung von indizierten Größen können besondere Zeichen als Indexsymbole verwendet werden, die Beginn und Ende der Indizes und die Abtrennung der einzelnen Indexstellen angeben, wobei diese Zeichen besonderen Vorrichtungen zugeführt werden, die intermediär eine Unterbrechung der laufenden Rechnung, die Auswertung der auf den Indexstellen befindlichen Ausdrücke nach dem obengenannten Verfahren und die Ansteuerung der durch die Indexauswertung festgestellten Einzelkomponente der induzierten Größe bewirken. Die Durchführung von derartigen Rechnungen wird weiter unten beispielsweise näher beschrieben, wobei diese Verfahren als Stufe IV bezeichnet sind.

Weitere Merkmale und Vorteile des Erfindungsgegenstandes gehen aus der folgenden Beschreibung von Ausführungsbeispielen hervor, die an Hand der Zeichnungen beschrieben werden.

الاستشارات المنارة

Patentansprüche:

1. Verfahren zur automatischen Verarbeitung von kodierten Daten, z. B. arithmetischen Formeln in üblicher Schreibweise, die als kodierte Zeichen Klammern, Operationssymbole, Zahlen und Variable gemischt, Dezimalkommas, Indizes, Entscheidungssymbole sowie Formelnummern enthalten, in einer datenverarbeitenden Maschine mit einer Eingabe- und einer Ausgabevorrichtung, **dadurch gekennzeichnet**, daß die den einzelnen Zeichen entsprechenden Signale in der Reihenfolge der Aufschreibung einem Analysator (4, 28) zugeführt und in diesem entsprechend der Reihenfolge des Eingangs geprüft werden, ob die Operationen sofort ausführbar sind oder ob der Eingang weiterer Signale abgewartet werden muß, daß in diesem letzteren Fall die noch nicht verarbeitbaren Zeichen in einen Speicher (Keller) eingeführt werden und daß beim Eintreffen neuer Zeichen im Analysator (4, 28), die die Ausführung einer Operation mit gespeicherten Zeichen ermöglichen, diese gespeicherten Zeichen in der durch die Art der Einführung festgelegten umgekehrten Reihenfolge entnommen und verarbeitet werden.

2. Verfahren nach Anspruch 1, dadurch gekennzeichnet, daß im Analysator die Formelzeichen nach Ziffernsymbolen (Zahlen) und Operationssymbolen getrennt sind und, sofern sie zurückgestellt werden müssen, zwei verschiedenen speicherfähigen Vorrichtungen (11, 12), vorzugsweise zwei verschiedenen »Kellern«, nämlich dem Zahlkeller (11) und dem Operationskeller (12), zugeführt werden.

3. Verfahren nach Ansprüchen 1 und 2, dadurch gekennzeichnet, daß die in dem Zahlkeller (11) bzw. dem Operationskeller (12) neu eintreffenden Zeichen sich jeweils an die Spitze der entsprechenden Sequenz setzen und die Entnahme eines Zeichens automatisch durch Wegnahme von der Spitze der entsprechenden Sequenz erfolgt.

4. Verfahren nach Ansprüchen 1 und 2, dadurch gekennzeichnet, daß solche Formelzeichen, welche ein Resultat verlangen, insbesondere das Gleichheitszeichen, einer besonderen Vorrichtung, nämlich der Ausgabesteuerung (6), zugeführt werden, daß ferner dadurch automatisch das Endergebnis der Formelauswertung zur Ausgabe gebracht wird.

5. Verfahren nach Ansprüchen 1 bis 4, dadurch gekennzeichnet, daß zur formelartigen Benennung von Zahlen oder Zahlsätzen, z. B. Ausgangsdaten und Teilergebnissen, besondere Zeichen, z. B. Buchstaben, als algebraische Größensymbole und zugehörige Kodezeichen benutzt werden, derart, daß jedes Zeichen beim erstmaligen Einlaufen in den Analysator (4, 28) eine Reservierung von an sich beliebigen Plätzen für eine Zahl oder einen Zahlsatz in einem an sich bekannten Speicher veranlaßt, wobei eine Zuordnung zwischen diesem Platz bzw. diesen Plätzen und dem betreffenden Größensymbol im Größenvorspeicher (24) bis auf Widerruf festgehalten wird.

6. Verfahren nach Anspruch 5, dadurch gekennzeichnet, daß das Größensymbol in Formeln vom Analysator (4, 28) stellvertretend für die auf dem zugeordneten Platz bzw. Plätzen gespeicherte Zahl bzw. Zahlsätze behandelt wird.

7. Verfahren nach Ansprüchen 1 bis 6, dadurch gekennzeichnet, daß zur Bezeichnung von indizierten Größen besondere Zeichen als Indexsymbole verwendet werden, die Beginn und Ende der Indizes und die Abtrennung der einzelnen Indexstellen angeben, wobei diese Zeichen im Analysator einer Hilfssteuereinrichtung (28) zugeführt werden, die intermediär eine Unterbrechung der laufenden Rechnung, die Auswertung der auf den Indexstellen befindlichen Ausdrücke nach dem obengenannten Verfahren und die Ansteuerung der durch die Indexauswertung festgestellten Einzelkomponente der indizierten Größe bewirken.

8. Verfahren nach Ansprüchen 1, 5 und 6, dadurch gekennzeichnet, daß eine Zurückstellung des Rechenvorganges erfolgt, wenn innerhalb einer Formel ein Größensymbol in den Analysator (4, 28) gelangt, für das noch keine Platzreservierung im Zahlspeicher (22) vorgenommen war, wobei von außen neue Information so lange verlangt wird, bis der nunmehr zu reservierende Platz durch eine Zahl besetzt worden ist.

9. Verfahren nach Ansprüchen 1 bis 4, dadurch gekennzeichnet, daß zur Numerierung von Formelgruppen besondere Zeichen als Kennzeichnungssymbole und zugehörige Kodezeichen benutzt werden.

المنارة للاستشارات

derart, daß jedes Kennzeichensymbol beim erstmaligen Einlaufen in den Analysator bewirkt, daß die Zuordnung zwischen dem Platz, den der Anfang der Formelgruppe in einem Formelspeicher (23) einnimmt, und dem Kennzeichnungssymbol bis auf Widerruf im Kennzeichenentschlüßler (25) festgehalten wird.

10. Verfahren nach Anspruch 9, dadurch gekennzeichnet, daß bereits die Angabe eines Kennzeichnungssymbols in Verbindung mit einem speziellen Sprungsymbol genügt, um zu bewirken, daß die Rechnung mit dem Beginn der unter dem betreffenden Kennzeichen im Formelspeicher (23) notierten Formelgruppe fortgesetzt wird, wobei der Sprung in bekannter Weise von Bedingungen abhängig sein kann.

11. Verfahren nach Ansprüchen 1, 9 und 10, dadurch gekennzeichnet, daß eine Zurückstellung des Rechenvorgangs erfolgt, wenn das zeitlich zuletzt im Formelspeicher (23) notierte Formelzeichen abgearbeitet ist, ohne daß es einen Sprung auf ein schon vorhandenes Kennzeichnungssymbol bewirkt, wobei von außen neue Information verlangt wird.

12. Verfahren nach Ansprüchen 1 und 9 bis 11, dadurch gekennzeichnet, daß ein Sprungsymbol, das auf eine noch nicht im Formelspeicher (23) notierte Formelgruppe führt, bzw. die Meldung, daß der Formelspeicher abgearbeitet ist bzw. daß ein Größensymbol in einer Formel erscheint, das noch keine Belegung im Zahlspeicher hat, bewirkt, daß der Analysator den Rechenvorgang abbricht und von außen neue Information verlangt; diese Information wird mindestens so lange im Formelspeicher oder, soweit es sich um eine Voreinstellungspeicherung handelt, im Zahlspeicher lediglich notiert, bis alle zur Fortsetzung der Rechnung notwendigen Angaben zur Verfügung stehen, worauf der Rechenvorgang automatisch an der Unterbrechungsstelle wieder einsetzt.

13. Verfahren nach Ansprüchen 1, 9 und 10, dadurch gekennzeichnet, daß eine Zurückstellung des Rechenvorganges erfolgt, wenn ein Sprungsymbol auf eine noch nicht im Formelspeicher (23) notierte Formelgruppe führt, wobei von außen eine neue Information verlangt wird, die im Formelspeicher lediglich notiert wird unter Festhaltung der durch Kennzeichnungssymbole bezeichneten Plätze der Anfänge einzelner Formelgruppen im Formelspeicher, und daß dieser Vorgang automatisch abbricht, wenn das Kennzeichnungssymbol der aufgerufenen Formelgruppe ausgewertet wird, wobei der Rechenvorgang wieder einsetzt.

14. Verfahren nach Ansprüchen 1 bis 13, dadurch gekennzeichnet, daß wahlweise an Stelle des bisher befolgten Prinzips der baldmöglichsten Ausführung aller Verarbeitungsvorgänge von Formelsymbolen eine weitere ganze oder teilweise Zurückstellung bis zu einem geeigneten späteren Zeitpunkt vorgenommen wird.

15. Verfahren nach Ansprüchen 1 und 9 bis 12, dadurch gekennzeichnet, daß ein besonderes Zeichen als Symbol »nicht notieren« interpretiert wird, derart, daß anschließend die Notierung der von außen einlaufenden Information bis auf Widerruf, z. B. durch ein auslösendes Symbol oder das nächste, in den Analysator (4) einlaufende Kennzeichnungssymbol für Formelgruppen, unterdrückt wird.

16. Automatische Rechenmaschine zur Ausführung des Verfahrens nach Ansprüchen 1 bis 15, dadurch gekennzeichnet, daß der Analysator einen Vorentschlüßler (5) enthält, dem sämtliche Formelzeichen in der Reihenfolge der üblichen Schreibweise zugeführt werden, und der mehrere Ausgänge aufweist, die zu einem Ziffernumsetzer (7), zu einem Operationsumsetzer (8) und zu einem Ausgabesteuerwerk (6) sowie zu einem Steuerwerk (9) für »bedeutungslose Zeichen« führen.

17. Rechenmaschine nach Anspruch 16, dadurch gekennzeichnet, daß die Umsetzer der Steuerwerke mit dem Schreibwerk (2) in Verbindung stehen.

18. Rechenmaschine nach Ansprüchen 16 und 17, dadurch gekennzeichnet, daß der Zahlkeller (11) und der Ziffernumsetzer (7) derart verbunden sind, daß der Zahlkeller die Zahlen in der Reihenfolge des Eintreffens von dem Umsetzer abnehmen kann und daß ferner die jeweils erste in der Sequenz stehende Zahl beim Eintreffen eines entsprechenden Befehls über das Ausgangssteuerwerk als Ergebnis dem Schreibwerk (2) zugeführt wird.

19. Rechenmaschine nach Ansprüchen 16 bis 18, dadurch gekennzeichnet, daß der Operationsumsetzer (8) mit dem Operationskeller (12) in Verbindung steht, so daß er in diesen die Operationssymbole in der Reihenfolge ihres Eintreffens einspeisen kann, wobei nach der den Ablauf der direkten Formelauswertung wiedergebenden Vorschrift entweder das jeweils zuletzt eingespeiste unter gleichzeitigem Nachrücken der früher eingespeisten Symbole von unten her oder das neu ankommende Symbol an das Rechenwerk (10) abgegeben werden kann, um solche Operationen auszuführen, für die die zugehörigen Operanden an der Spitze der im Zahlkeller befindlichen Sequenz vorliegen.

20. Rechenmaschine nach Ansprüchen 16 bis 19, dadurch gekennzeichnet, daß Einrichtungen vorgesehen sind, die beim Eintreffen solcher Zeichen, die eine Formel abschließen, insbesondere des Gleichheitszeichens oder des Ergibtzeichens, eine Prüfung auf »sinnvolle Formel« veranlassen.

21. Rechenmaschine nach Ansprüchen 16 bis 20, dadurch gekennzeichnet, daß der Operationskeller (12) und der Zahlkeller (11) Einrichtungen aufweisen, die die zugeführten Zeichen dadurch sequentiell speichern, daß sie die bereits gespeicherten Zeichen in der Reihenfolge des Eintreffens nach unten weiterschieben und eine Abgabe nur des jeweils zuletzt gespeicherten oder des obersten der von unten nachzuschiebenden Zeichen gestatten.

22. Rechenmaschine nach Ansprüchen 16 bis 20, dadurch gekennzeichnet, daß der Operationskeller (12) und der Zahlkeller (11) Einrichtungen aufweisen, die die zugeführten Zeichen dadurch sequentiell speichern, daß jedes eintreffende Zeichen auf den Platz vor dem zuletzt eingetroffenen gesetzt wird, daß dieser Platz festgehalten wird und daß ferner die Abnahme von dem zuletzt festgehaltenen Platz erfolgt.

23. Rechenmaschine nach Ansprüchen 16 bis 22, dadurch gekennzeichnet, daß das Rechenwerk (10) die im obersten oder in den beiden obersten Geschossen des Zahlkellers befindlichen Zahlen entsprechend den von der Operationssteuerung erhaltenen Befehlen verarbeitet und das Ergebnis wieder an das oberste Geschoß des Zahlkellers abgibt.

24. Rechenmaschine nach Ansprüchen 16 bis 23, dadurch gekennzeichnet, daß das Ausgabesteuerwerk (16) mit dem Zahlkeller (11) derart verbunden ist, daß beim Eintreffen eines Gleichheitszeichens und gegebenenfalls nachfolgender Zeichen »Ziffer verlangt« die Verbindung des Zahlkellers mit dem Schreibwerk hergestellt und die im obersten Geschoß des Zahlkellers befindliche Zahl ganz oder teilweise an das Schreibwerk abgegeben wird.

المنارة للاستشارات

25. Rechenmaschine nach Anspruch 24, dadurch gekennzeichnet, daß in dem Tastenfeld (1) eine Ergebnistaste vorgesehen ist, die bewirkt, daß die einzelnen Stellen des Ergebnisses jeweils beim Anschlag der Taste geschrieben werden, so daß jede gewünschte Anzahl von Ergebnisstellen geschrieben werden kann.

26. Rechenmaschine nach Ansprüchen 16 bis 25, dadurch gekennzeichnet, daß an den Vorentschlüßler (5) ein im Steuerwerk befindlicher Größenentschlüßler angeschlossen ist, der beim erstmaligen Eintreffen eines Größensymbols, vorzugsweise wenn es unmittelbar auf ein »Ergibtzeichen« folgt, diesem Größensymbol die Nummer eines freien Platzes im Zahlspeicher derart zuordnet, daß fernerhin dasselbe Größensymbol beim Einlaufen in den Größenspeicher unmittelbar die Ansteuerung des zugehörigen Speicherplatzes zur Aufnahme von Zahlen aus dem Zahlkeller bzw. dem Rechenwerk oder zur Abgabe von Zahlen in den Zahlkeller bzw. das Rechenwerk bewirkt.

27. Rechenmaschine nach Ansprüchen 16 bis 26, dadurch gekennzeichnet, daß an den Vorentschlüßler ein im Steuerwerk befindlicher Kennzeichenentschlüßler (25) angeschlossen ist, der beim erstmaligen Eintreffen eines Kennzeichnungssymbols für Formelgruppen diesem die Nummer desjenigen Platzes im Formelspeicher zuweist, auf den das erste Symbol der nachfolgenden Formelgruppe trifft, derart, daß fernerhin dasselbe Kennzeichnungssymbol in Verbindung mit einem Sprungsymbol unmittelbar die Ansteuerung des festgehaltenen Platzes des Anfangs der Formelgruppe im Formelspeicher bewirkt, von wo aus die Formelentschlüßlung fortgesetzt wird.

28. Rechenmaschine nach Ansprüchen 21 und 22, dadurch gekennzeichnet, daß der Zahlkeller (11) mit dem Rechenwerk (10) derart vereinigt ist, daß die üblicherweise als Multiplikanden-Divisor-Register, Akkumulator und Multiplikatorregister bezeichneten speicherfähigen Einrichtungen des Rechenwerkes ganz oder teilweise in den obersten Plätzen des Zahlkellers liegen.

29. Rechenmaschine nach Ansprüchen 16 bis 28, dadurch gekennzeichnet, daß die Plätze des Zahlkellers auch im Falle des Vorkommens von Zahlen wechselnder Länge voll ausgenutzt werden, wobei die einzelnen Zahlen gegebenenfalls durch Markierungs- oder Schlußzeichen voneinander getrennt sein können.

30. Rechenmaschine nach Anspruch 28, dadurch gekennzeichnet, daß der Zahlkeller nach oben als Appendix fortgesetzt und andererseits mit einem ringförmigen Speicher über Verschiebeeinrichtungen verbunden ist, derart, daß die Durchführung der Rechenoperationen auf synchrone Verschiebungen im Appendix und im ringförmigen Speicher unter gleichzeitiger stellenweiser Addition und Verschiebung in den Zahlkeller hinein zurückgeführt werden kann.

31. Rechenmaschine nach Anspruch 28, dadurch gekennzeichnet, daß der Zahlkeller als ringförmiger Speicher derart ausgebildet ist, daß er durch Verschiebeeinrichtungen in Verbindung mit einem weiteren ringförmigen Speicher steht, so daß die Rechenoperationen auf synchrone Verschiebungen in den beiden ringförmigen Speichern unter gleichzeitig stellenweiser Addition in den Zahlkeller hinein zurückgeführt werden können.

32. Rechenmaschine zur Ausführung des Verfahrens nach Ansprüchen 1 bis 15, dadurch gekennzeichnet, daß die Rechenregister als ansteuerbare, aber nicht notwendig verschiebbare Speicher ausgebildet sind, derart, daß über parallel ausgebildete Sucheinrichtungen die Operanden abgegriffen und dadurch die Durchführung der Rechenoperationen auf sukzessive stellenweise Additionen zurückgeführt wird, wobei das Ergebnis in einem der beiden Operandenplätze wieder aufgebaut werden kann.

33. Rechenmaschine nach Anspruch 32, dadurch gekennzeichnet, daß der Zahlkeller und gegebenenfalls das Multiplikanden-Divisor-Register als ansteuerbare, aber nicht notwendig verschiebbare Speicher ausgebildet sind und daß zur Durchführung der arithmetischen Operationen der Zahlkeller mitbenutzt wird.

34. Rechenmaschine nach Ansprüchen 32 und 33, dadurch gekennzeichnet, daß der Zahlkeller (11) ganz oder teilweise in Plätze des an sich vorhandenen Zahlspeichers (22), vorzugsweise in die jeweils freien Plätze, gelegt wird, wobei der jeweilige Stand des freien Speichers und der jeweilige Stand der Spitze der Zahlkellersequenz durch besondere Zählregister festgehalten werden kann.

35. Rechenmaschine zur Ausführung des Verfahrens nach Ansprüchen 1 bis 15, dadurch gekennzeichnet, daß an Stelle des Zahlkellers ein Platznummernkeller tritt, in dem anstatt der in den Zahlkeller einzufahrenden Zahlen deren Platznummern im Hauptzahlspeicher festgehalten und bei der Formelauswertung stellvertretend für die durch sie anzusteuernden Zahlen behandelt werden.

36. Rechenmaschine nach Ansprüchen 32, 33 und 34, dadurch gekennzeichnet, daß der Operand oder die beiden Operanden einer arithmetischen Operation mittels zählfähiger Register, die von den Inhalten des Platznummernspeichers her eingestellt werden, angesteuert werden und daß die Speicherplätze des Resultats von dem Register, in dem der jeweilige Stand der Spitze der Zahlkellersequenz festgehalten wird, her angesteuert werden, wobei der Stand des Registers »freier Speicher« zur Sinnvolltestung herangezogen werden kann.

37. Rechenmaschine nach Ansprüchen 35 und 36, dadurch gekennzeichnet, daß zur Kennzeichnung von Zahlen wechselnder Länge die Platznummern des Zahlenanfanges und die Stellenzahl im Größenvorspeicher (24) festgehalten werden.

38. Rechenmaschine nach Ansprüchen 29 oder 37 und 38, dadurch gekennzeichnet, daß mit Zahlen wechselnder und im Prinzip beliebiger Länge gearbeitet wird.

39. Rechenmaschine zur Ausführung des Verfahrens nach Ansprüchen 1 bis 6, dadurch gekennzeichnet, daß zur Auffindung der einzelnen Komponenten von Zahlsätzen, die indizierten Größen entsprechen, die Platznummer des Anfangs des Zahlsatzes und die Kenngrößen für den Indexlauf sowie gegebenenfalls die Zahllänge im Größenvorspeicher (24) festgehalten werden.

40. Rechenmaschine nach Anspruch 39, dadurch gekennzeichnet, daß eine Zuweisung von Speicherplätzen zu Größensymbolen, insbesondere indizierten Größensymbolen, beim ersten Auftreten einer expliziten Speicherungsvorschrift vorgenommen wird, wobei einer zählfähigen Vorrichtung der Stand des »freien Speichers« entnommen und im Vorspeicher unter dem Eingang des Größensymbols gespeichert wird und wobei ferner die Kenngrößen für den Indexlauf, gegebenenfalls einschließlich der Zahllängen, der Speicherungsvorschrift entnommen werden.

41. Rechenmaschine nach Ansprüchen 39 und 40, dadurch gekennzeichnet, daß die Indexsymbole einer Hilfssteuereinrichtung zugeführt werden, die den Übergang zur Auswertung der Formelausdrücke auf

المنارة للاستشارات

den einzelnen Indexstellen veranlaßt und für die Einschiebung der speziellen Index-Auswertungsoperationen, die mit den Kenngrößen des Indexlaufes bzw. der Zahllänge und der Platznummer des Anfangs durchzuführen sind, sorgt und mit der so errechneten 5 Platznummer der betreffenden Komponente der indizierten Größe diese im Zahlspeicher aufsucht und sie der weiteren Formelauswertung zur Verfügung stellt.

In Betracht gezogene Druckschriften:
Deutsche Patentschrift Nr. 922 085;
Hollerith-Nachr., 74, 1937, S. 1022.

Hierzu 2 Blatt Zeichnungen

المنارة للاستشارات

ZEICHNUNGEN BLATT 1

AUSGABETAG: 1. DEZEMBER 1960

DAS 1094019
KL. 42 m 14
INTERNAT. KL. G 06 f

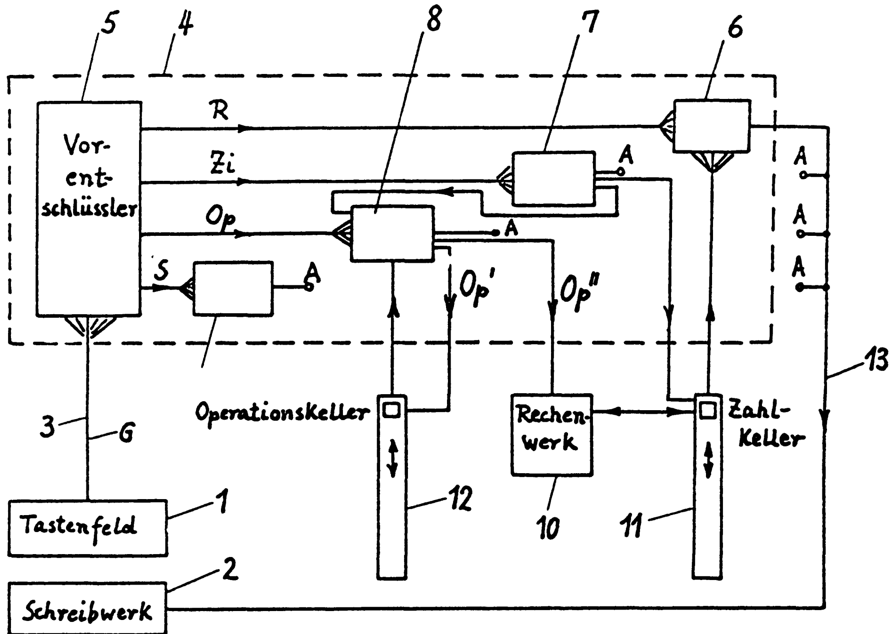


Fig. 1

E \ D	⊠	(	B R	× :	+ −
V	nf	nf	f	f	f
(	K	K	ke	ke	Ke
B,R	Kn	Kn	K	K	K
× :	K	K	K	a	K
+ −	K	K	r	r	a
)		c	r	r	r
⇒	s,e		r	r	r

Op", Op'

Fig. 2

المنارة للاستشارات

ZEICHNUNGEN BLATT 1 AUSGABETAG: 1. DEZEMBER 1960

DAS 1094019
KL. 42 m 14
INTERNAT. KL. G 06 f

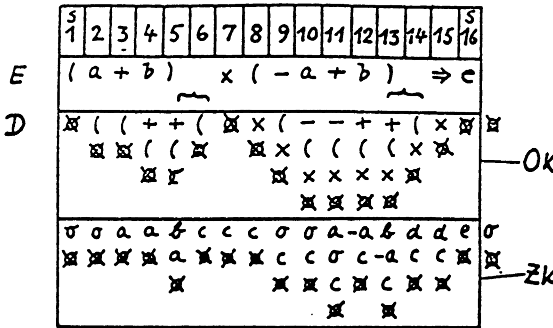


Fig. 3

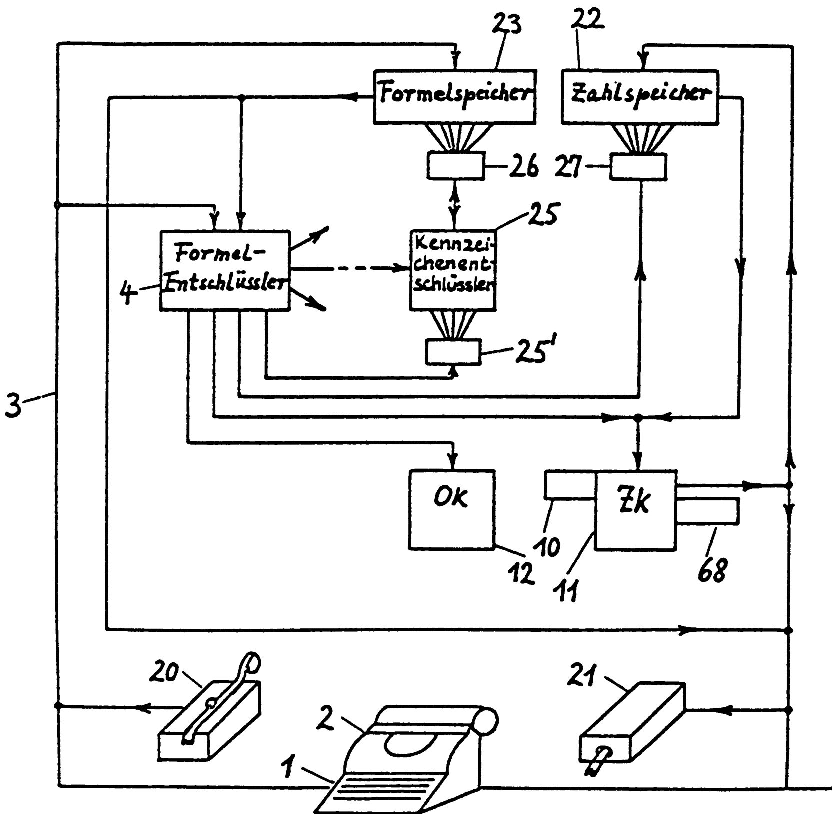


Fig. 4

المنارة للاستشارات

ZEICHNUNGEN BLATT 2 AUSGABETAG: 1. DEZEMBER 1960 DAS 1094019
KL. 42 m 14
INTERNAT. KL. G 06 f

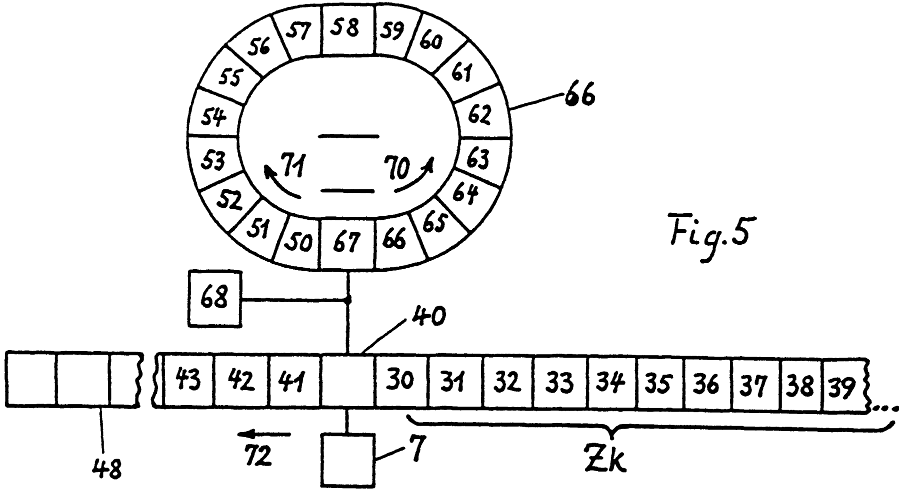


Fig. 5

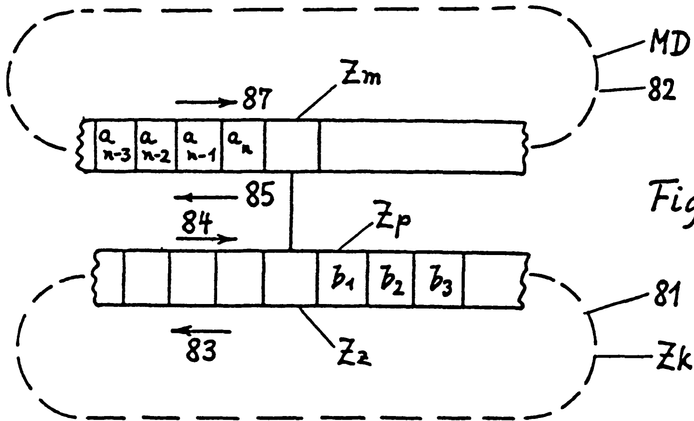


Fig. 6

009 650/226

المنارة للاستشارات

Rudolf Bayer

R. Bayer, E. McCreight
Organization and Maintenance of Large Ordered Indexes

Acta Informatica, Vol. 1, Fasc. 3, 1972
pp. 173–189

المنارة للاستشارات

Acta Informatica 1, 173–189 (1972)
© by Springer-Verlag 1972

Organization and Maintenance of Large Ordered Indexes

R. Bayer and E. McCreight

Received September 29, 1971

Summary. Organization and maintenance of an index for a dynamic random access file is considered. It is assumed that the index must be kept on some pseudo random access backup store like a disc or a drum. The index organization described allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ where I is the size of the index and k is a device dependent natural number such that the performance of the scheme becomes near optimal. Storage utilization is at least 50% but generally much higher. The pages of the index are organized in a special data-structure, so-called B-trees. The scheme is analyzed, performance bounds are obtained, and a near optimal k is computed. Experiments have been performed with indexes up to 100000 keys. An index of size 15000 (100000) can be maintained with an average of 9 (at least 4) transactions per second on an IBM 360/44 with a 2311 disc.

1. Introduction

In this paper we consider the problem of organizing and maintaining an index for a dynamically changing random access file. By an *index* we mean a collection of index elements which are pairs (x, α) of fixed size physically adjacent data items, namely a key x and some associated information α. The key x identifies a unique element in the index, the associated information is typically a pointer to a record or a collection of records in a random access file. For this paper the associated information is of no further interest.

We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are *pseudo random access devices* which have a rather long access or wait time—as opposed to a true random access device like core store—and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head discs, drums, and data cells.

Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert keys—more accurately index elements—economically. The index organization described in this paper always allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ or better, where I is the size of the index, and k is a device dependent natural number which describes the page size such that the performance of the maintenance and retrieval scheme becomes near optimal.

In more illustrative terms theoretical analysis and actual experiments show that it is possible to maintain an index of size 15000 with an average of 9 retrievals, insertions, and deletions per second in real time on an IBM 360/44 with a 2311 disc as backup store. According to our theoretical analysis, it should be possible to maintain an index of size 1500000 with at least two transactions per second on such a configuration in real time.

المنارة للاستشارات

The index is organized in pages of fixed size capable of holding up to $2k$ keys, but pages need only be partially filled. Pages are the blocks of information transferred between main store and backup store.

The pages themselves are the nodes of a rather specialized tree, a so-called B-tree, described in the next section. In this paper these trees grow and contract in only one way, namely nodes split off a brother, or two brothers are merged or "catenated" into a single node. The splitting and catenation processes are initiated at the leaves only and propagate toward the root. If the root node splits, a new root must be introduced, and this is the only way in which the height of the tree can increase. The opposite process occurs if the tree contracts.

There are, of course, many competitive schemes, e.g., hash-coding, for organizing an index. For a large class of applications the scheme presented in this paper offers significant advantages over others:

i) Storage utilization is at least 50% at any time and should be considerably better in the average.

ii) Storage is requested and released as the file grows and contracts. There is no congestion problem or degradation of performance if the storage occupancy is very high.

iii) The natural order of the keys is maintained and allows processing based on that order like: find predecessors and successors; search the file sequentially to answer queries; skip, delete, retrieve a number of records starting from a given key.

iv) If retrievals, insertions, and deletions come in batches, very efficient essentially sequential processing of the index is possible by presorting the transactions on their keys and by using a simple prepaging algorithm.

Several other schemes try to solve the same or very similar problems. AVL-trees described in [1] and [2] guarantee performance in time $\log_2 I$, but they are suitable only for a one-level store. The schemes described in [3] and [4] do not have logarithmic performance. The solution presented in this paper is new and is related to those described in [1–4] only in the sense that the problem to be solved is similar and that it uses a data organization involving tree structures.

2. *B*-Trees

Def. 2.1. Let $h \geqq 0$ be an integer, k a natural number. A directed tree T is in the class $\tau(k, h)$ of *B-trees* if T is either empty ($h=0$) or has the following properties:

i) Each path from the root to any leaf has the same length h, also called the *height* of T, i.e., h = number of nodes in path.

ii) Each node except the root and the leaves has at least $k+1$ sons. The root is a leaf or has at least two sons.

iii) Each node has at most $2k+1$ sons.

Number of Nodes in B-Trees. Let $N_{\min}$ and $N_{\max}$ be the minimal and maximal number of nodes in a B-tree $T \in \tau(k, h)$. Then

$$N_{\min} = 1 + 2\left((k+1)^0 + (k+1)^1 + \cdots + (k+1)^{h-2}\right) = 1 + \frac{2}{k}\left((k+1)^{h-1} - 1\right)$$

المنارة للاستشارات

for $h \geqq 2$. This also holds for $h=1$. Similarly one obtains

$$N_{\max} = \sum_{i=0}^{h-1} (2k+1)^i = \frac{1}{2k}\left((2k+1)^h - 1\right); \quad h \geqq 1.$$

Upper and lower bounds for the number $N(T)$ of nodes of $T \in \tau(k, h)$ are given by:

$$N(T) = 0 \quad \text{if } T \in \tau(k, 0); \tag{2.1}$$

$$1 + \frac{2}{k}\left((k+1)^{h-1} - 1\right) \leqq N(T) \leqq \frac{1}{2k}\left((2k+1)^h - 1\right) \quad \text{otherwise.}$$

Note that the classes $\tau(k, h)$ need not be disjoint.

3. The Data Structure and Retrieval Algorithm

To repeat, the pages on which the index is stored are the nodes of a B-tree $T \in \tau(k, h)$ and can hold up to $2k$ keys. In addition the data structure for the index has the following properties:

i) Each page holds between k and $2k$ keys (index elements) except the root page which may hold between 1 and $2k$ keys.

ii) Let the number of keys on a page P, which is not a leaf, be l. Then P has $l+1$ sons.

iii) Within each page P the keys are sequential in increasing order: $x_1, x_2, \ldots, x_l$; $k \leqq l \leqq 2k$ except for the root page for which $1 \leqq l \leqq 2k$. Furthermore, P contains $l+1$ pointers $p_0, p_1, \ldots, p_l$ to the sons of P. On leaf pages these pointers are undefined. Logically a page is then organized as shown in Fig. 1.

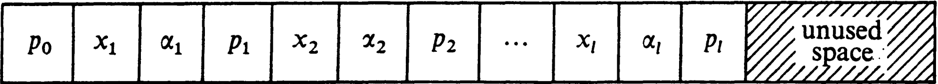


Fig. 1. Organization of a page

The α_i are the associated information in the index element (x_i, α_i). The triple (x_i, α_i, p_i) or—omitting α_i—the pair (x_i, p_i) is also called an *entry*.

iv) Let $P(p_i)$ be the page to which p_i points, let $K(p_i)$ be the set of keys on the pages of that maximal subtree of which $P(p_i)$ is the root. Then for the B-trees considered here the following conditions shall always hold:

$$(\forall y \in K(p_0))(y < x_1), \tag{3.1}$$

$$(\forall y \in K(p_i))(x_i < y < x_{i+1}); \quad i = 1, 2, \ldots, l-1, \tag{3.2}$$

$$(\forall y \in K(p_l))(x_l < y). \tag{3.3}$$

Fig. 2 is an example of a B-tree in $\tau(2, 3)$ satisfying all the above conditions. In the figure the α_i are not shown and the page pointers are represented graphically. The boxes represent pages and the numbers outside are page numbers to be used later.

المنارة للاستشارات

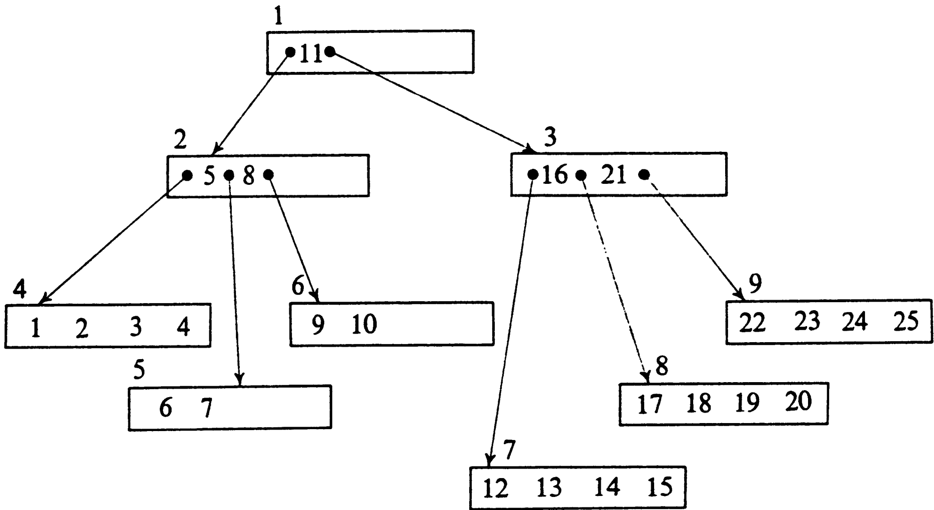


Fig. 2. A data structure in $\tau(2, 3)$ for an index

Retrieval Algorithm. The flowchart in Fig. 3 is an algorithm for retrieving a key y. Let p, r, s be pointer variables which can also assume the value "undefined" denoted as u. r points to the root and is u if the tree is empty, s does not serve any purpose for retrieval, but will be used in the insertion algorithm. Let $P(p)$ be the page to which p is pointing, then $x_1, \ldots, x_l$ are the keys in $P(p)$ and $p_0, \ldots, p_l$ the page pointers in $P(p)$.

The retrieval algorithm is simple logically, but to program it for a computer one would use an efficient technique, e.g., a binary search, to scan a page.

Cost of Retrieval. Let h be the height of the page tree. Then at most h pages must be scanned and therefore fetched from backup store to retrieve a key y. We will now derive bounds for h for a given index of size I. The minimum and maximum number $I_{\min}$ and $I_{\max}$ of keys in a B-tree of pages in $\tau(k, h)$ are:

$$I_{\min} = 1 + k\left(2\,\frac{(k+1)^{h-1}-1}{k}\right) = 2(k+1)^{h-1} - 1$$

$$I_{\max} = 2k\left(\frac{(2k+1)^h - 1}{2k}\right) = (2k+1)^h - 1.$$

This is immediate from (2.1) for $h \geqq 1$. Thus we have as sharp bounds for the height h:

$$\log_{2k+1}(I+1) \leqq h \leqq 1 + \log_{k+1}\left(\frac{I+1}{2}\right) \quad \text{for} \quad I \geqq 1, \qquad h = 0 \quad \text{for} \quad I = 0. \tag{3.1}$$

4. Key Insertion

The algorithm in Fig. 4 inserts a single key y into an index described in Section 3. The variable s is a page pointer set by the retrieval algorithm pointing to the last page that was scanned or having the value u if the page tree is empty.

المنارة للاستشارات

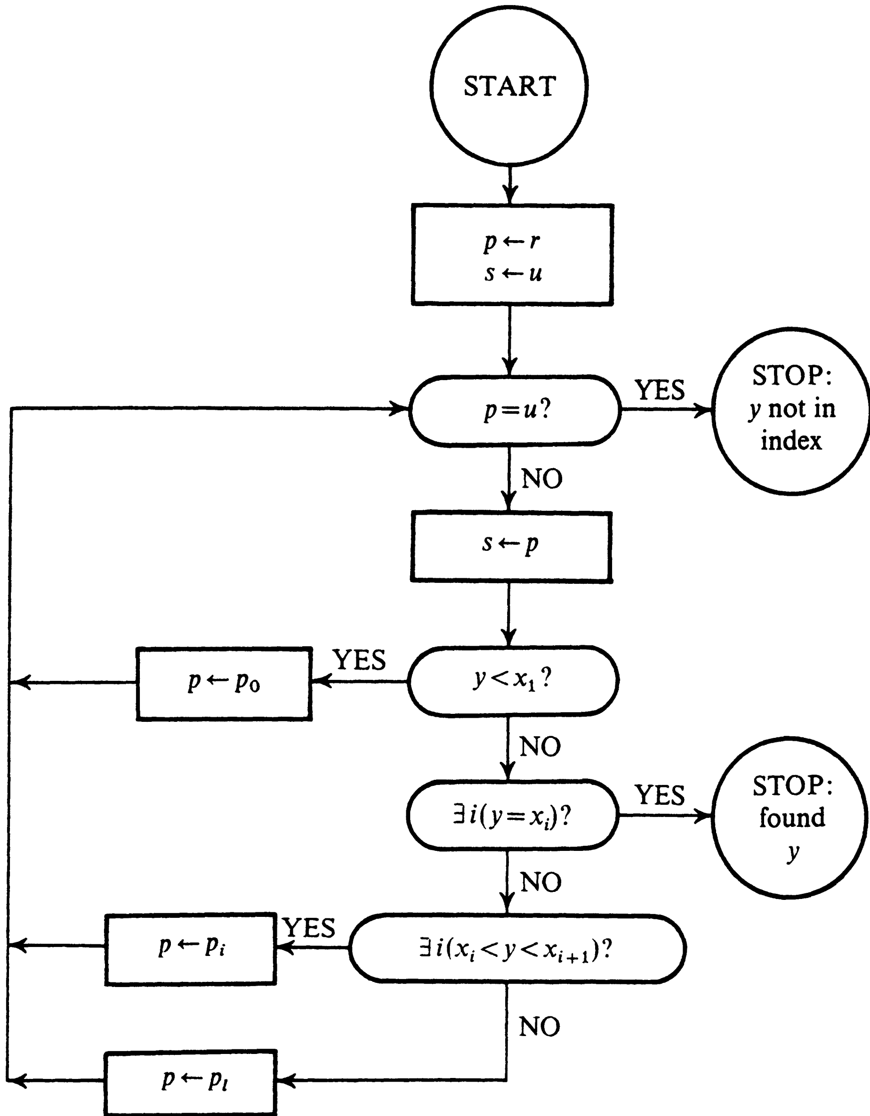


Fig. 3. Retrieval algorithm

Splitting a Page. If a page P in which an entry should be inserted is already full, it will be split into two pages. Logically first insert the entry into the sequence of entries in P—which is assumed to be in main store—resulting in a sequence

$$p_0, (x_1, p_1), (x_2, p_2), \ldots, (x_{2k+1}, p_{2k+1}).$$

Now put the subsequence $p_0, (x_1, p_1), \ldots, (x_k, p_k)$ into P and introduce a new page P' to contain the subsequence

$$p_{k+1}, (x_{k+2}, p_{k+2}), (x_{k+3}, p_{k+3}), \ldots, (x_{2k+1}, p_{2k+1}).$$

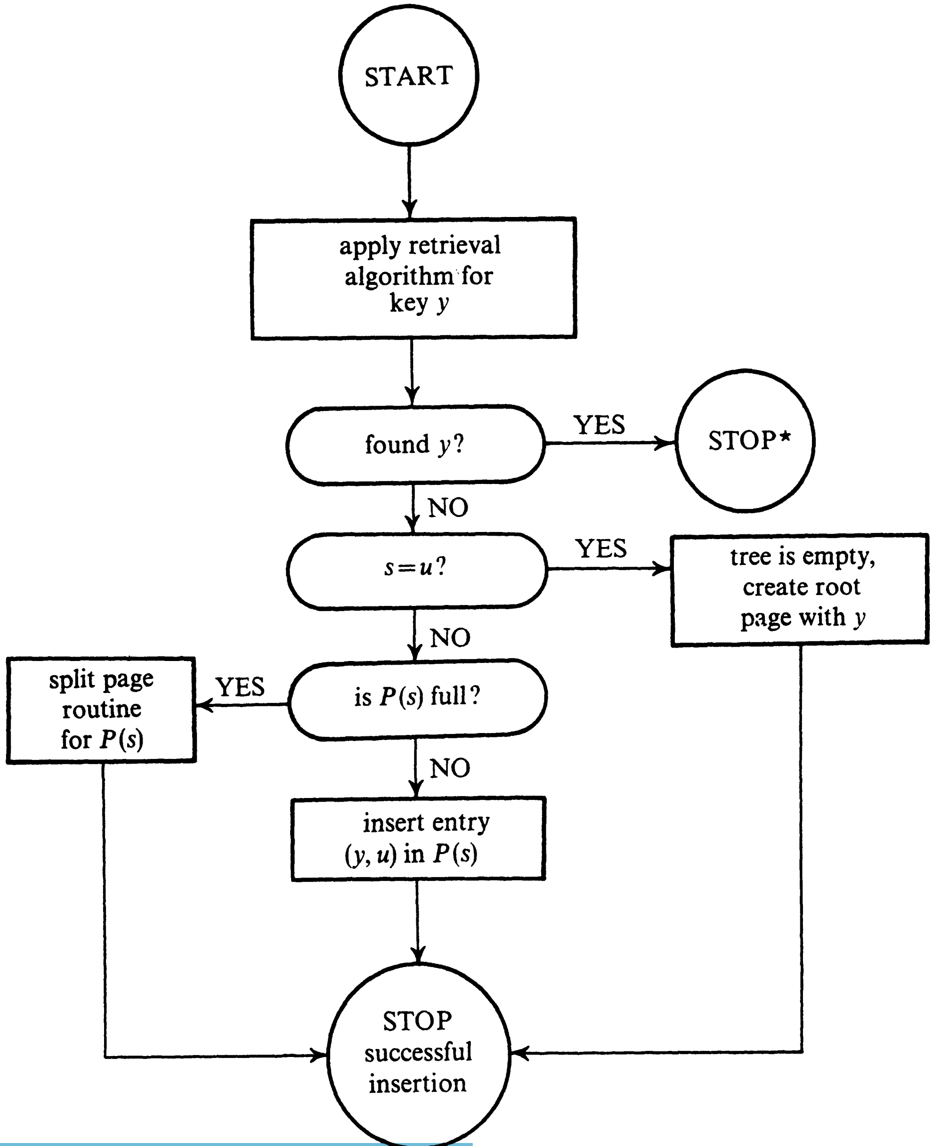
Let Q be the father page of P. Insert the entry (x_{k+1}, p'), where p' points to P', into Q. Thus P' becomes a brother of P.

المنارة للاستشارات

Inserting (x_{k+1}, p') into Q may, of course, cause Q to split too, and so on, possibly up to the root. If the splitting page P is the root, then we introduce a new root page Q containing p, (x_{k+1}, p') where p points to P and p' to P'.

Note that this insertion process maps B-trees with parameter k into B-trees with parameter k, and preserves properties (3.1), (3.2), and (3.3).

To illustrate the insertion process, insertion of key 9 into the tree in Fig. 5 with parameter $k=2$ results in the tree in Fig. 2.



* Key y is already in index, take appropriate action.

Fig. 4. Insertion algorithm

المنارة للاستشارات

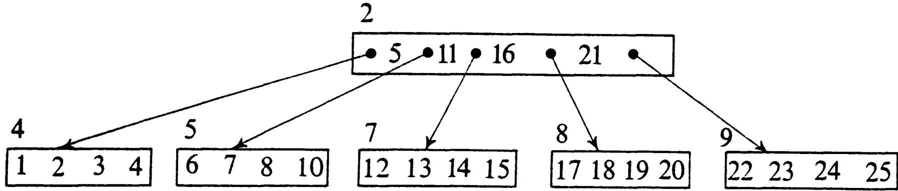


Fig. 5. Index structure in $\tau(2, 2)$

5. Cost of Retrievals and Insertions

To analyze the cost of maintaining an index and retrieving keys we need to know how many pages must be fetched from the backup store into main store and how many pages must be written onto the backup store. For our analysis we make the following assumption: Any page, whose content is examined or modified during a single retrieval, insertion, or deletion of a key, is fetched or paged out respectively exactly once. It will become clear during the course of this paper that a paging area to hold $h+1$ pages in main store is sufficient to do this.

Any more powerful paging scheme, like e.g., keeping the root page permanently locked in main store, will, of course, decrease the number of pages which must be fetched or paged out. We will not, however, analyze such schemes, although we have used them in our experiments.

Denote by $f_{\min}$ ($f_{\max}$) the minimal (maximal) number of pages fetched, and by $w_{\min}$ ($w_{\max}$) the minimal (maximal) number of pages written.

Cost of Retrieval. From the retrieval algorithm it is clear that for retrieving a single key we get

$$f_{\min}=1; \quad f_{\max}=h; \quad w_{\min}=w_{\max}=0.$$

Cost of Insertion. For inserting a single key the least work is required if no page splitting occurs, then

$$f_{\min}=h; \quad w_{\min}=1.$$

Most work is required if all pages in the retrieval path including the root page split into two. Since the retrieval path contains h pages and we have to write a new root page, we get:

$$f_{\max}=h; \quad w_{\max}=2h+1.$$

Note that h always denotes the height of the old tree. Although this worst bound is sharp, it is not a good measure for the amount of work which must generally be done for inserting one key.

If we consider an index in which keys are only retrieved or inserted, but no keys are deleted, then we can derive a bound for the average amount of work to be done for building an index of I keys as follows:

Each page split causes one (or two if the root page splits) new pages to be created. Thus the number of page splits occurring in building an index of I items is bounded by $n(I)-1$, where $n(I)$ is the number of pages in the tree. Since

المنارة للاستشارات

each page has at least k keys, except the root page which may have only 1, we get: $n(I) \leqq \frac{I-1}{k} + 1$. Each single page split causes at most 2 additional pages to be written. Thus the average number of pages written per single key insertion due to page splitting is bounded by

$$(n(I)-1) \cdot \frac{2}{I} < \frac{2}{k}.$$

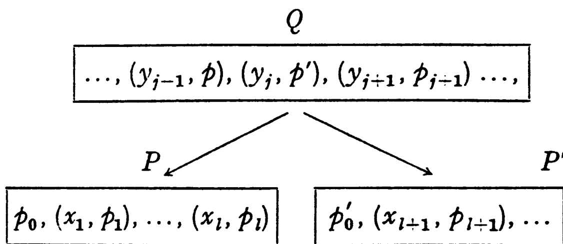
A page split does not require any additional page retrievals. Thus in the average for an index without deletions we get for a single insertion:

$$f_a = h; \quad w_a < 1 + \frac{2}{k}.$$

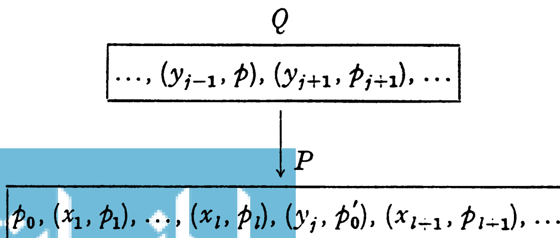
6. Deletion Process

In a dynamically changing index it must be necessary to delete keys. The algorithm of Fig. 6 deletes one key y from an index and maintains our data structure properly. It first locates the key, say y_i. To maintain the data structure properly, y_i is deleted if it is on a leaf, otherwise it must be replaced by the smallest key in the subtree whose root is $P(p_i)$. This smallest key is found by going from $P(p_i)$ along the p_0 pointers to the leaf page, say L, and taking the first key in L. Then this key, say x_1, is deleted from L. As a consequence L may contain fewer than k keys and a catenation or underflow between L and an adjacent brother is performed.

Catenation. Two pages P and P' are called *adjacent brothers* if they have the same father Q and are pointed to by adjacent pointers in Q. P and P' can be catenated, if together they have fewer than $2k$ keys, as follows: The three pages of the form



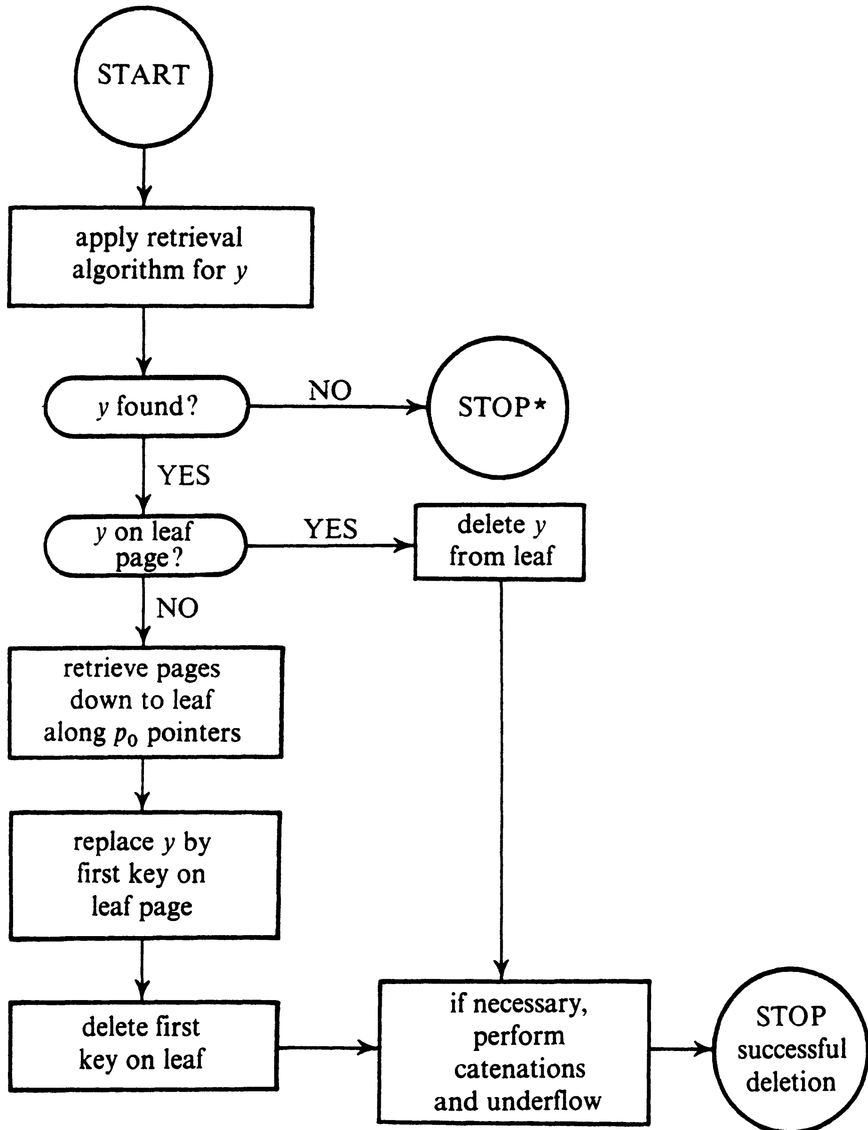
can be replaced by two pages of the form:



المنارة للاستشارات

As a consequence of deleting the entry (y_j, p') from Q it is now possible that Q contains fewer than k keys and special action must be taken for Q. This process may propagate up to the root of the tree.

Underflow. If the sum of the number of keys in P and P' is greater than $2k$, then the keys in P and P' can be equally distributed, the process being called an underflow, as follows:



* The key to be deleted is not in index, take appropriate action.

Fig. 6. Deletion algorithm

المنارة للاستشارات

Perform the catenation between P and P' resulting in too large a P. This is possible since P is in main store. Now split P "in the middle" as described in Section 4 with some obvious minor modifications.

Note that underflows do not propagate. Q is modified, but the number of keys in it is not changed.

To illustrate the deletion process consider the index in Fig. 2. Deleting key 9 results in the index in Fig. 5.

7. Cost of Deletions

For a successful deletion, i.e., if the key y to be deleted is in the index, the least amount of work is required if no catenations or underflows are performed and y is in a leaf. This requires:

$$f_{\min} = h; \qquad w_{\min} = 1.$$

If y is not in a leaf and no catenations or underflows occur, then

$$f = h; \qquad w = 2.$$

A maximal amount of work must be done if all but the first two pages in the retrieval path are catenated, the son of the root in the retrieval path has an underflow, and the root is modified. This requires:

$$f_{\max} = 2h - 1; \qquad w_{\max} = h + 1.$$

As in the case of the insertion process the bounds obtained are sharp, but very far apart and assumed rarely except in pathological examples. To obtain a more useful measure for the average amount of work necessary to delete a key, let us consider a "pure deletion process" during which all keys in an index I are deleted, but no keys are inserted.

Disregarding for the moment catenations and underflows we may get $f_1 = h$ and $w_1 = 2$ for each deletion at worst. But this is the best bound obtainable if one considers an example in which keys are always deleted from the root page.

Each deletion causes at most one underflow, requiring $f_2 = 1$ additional fetches and $w_2 = 2$ additional writes.

The total number of possible catenations is bounded by $n(I) - 1$, which is at most $\frac{I-1}{k}$. Each catenation causes 1 additional fetch and 2 additional writes, which results in an average

$$f_3 = \frac{1}{I}\left(\frac{I-1}{k}\right) < \frac{1}{k}$$

$$w_3 = \frac{2}{I}\left(\frac{I-1}{k}\right) < \frac{2}{k}.$$

Thus in the average we get:

$$f_a \leqq f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$

$$w_a \leqq w_1 + w_2 + w_3 < 2 + 2 + \frac{2}{k} = 4 + \frac{2}{k}.$$

المنارة للاستشارات

8. Page Overflow and Storage Utilization

In the scheme described so far utilization of back-up store may be as low as 50% in extreme cases—disregarding the root page—if all pages contain only k keys. This could be improved by avoiding certain page splits.

An *overflow* between two adjacent brother pages P and P' can be performed as follows: Assume that a key must be inserted in P and P is already full, but P' is not full. Then the key is inserted into the key-sequence in P and an underflow as described in Section 6 between the resulting sequence and P' is performed. This avoids the need to split P into two pages. Thus a page will be split only if both adjacent brothers are full, otherwise an overflow occurs.

In an index without deletions overflows will increase the storage utilization in the worst cases to about 66%. If both insertions and deletions occur, then the storage utilization may of course again be as low as 50%. For most practical applications, however, storage utilization should be improved appreciably with overflows.

One could, of course, consider a larger neighborhood of pages than just the adjacent brothers as candidates for overflows, underflows, and catenations and increase the minimal storage occupancy accordingly.

Bounds for the cost of insertions for a scheme with overflows are easily derived as:

$$f_{\min} = h; \qquad w_{\min} = 1;$$

$$f_{\max} = 3h - 2; \quad w_{\max} = 2h + 1.$$

For a pure insertion process one obtains as bounds for the average cost:

$$f_a < h + 2 + \frac{2}{k}; \qquad w_a < 3 + \frac{2}{k}.$$

It is easy to construct examples in which each insertion causes an overflow, thus these bounds cannot be improved very much without special assumptions about the insertion process.

9. Maintenance Cost for Index with Insertions and Deletions

The main purpose of this paper is to develop a data structure which allows economical maintenance of an index in which retrievals, insertions, and deletions must be done in any order. We will now derive bounds on the processing cost in such an environment.

The derivation of bounds for retrieval cost did not make any assumptions about the order of insertions or deletions, so they are still valid. Also, the minimal and maximal bounds for the cost of insertions and deletions were derived without any such assumptions and are still valid. The bounds derived for the average cost, however, are no longer valid if insertions and deletions are mixed.

The following example shows that the upper bounds for the average cost cannot be improved appreciably over the upper bounds of the cost derived for a single retrieval or deletion.

المنارة للاستشارات

Example. Consider the trees T_2 in Fig. 2 and T_5 in Fig. 5. Deleting key 9 from T_2 leads to T_5, and inserting key 9 in T_5 leads back to T_2. Consider a sequence of alternating deletions and insertions of key 9 being applied starting with T_2.

Case 1. No page overflows, but only page splits occur:

i) Each deletion of key 9 from T_2 requires:
3 retrievals to locate key 9, namely pages 1, 2, 6.
1 retrieval of brother 5 of page 6 to find out that pages 5 and 6 can be catenated.
2 pages, namely 5 and 2 are modified and must be written. Pages 6 and 3 are deleted from the tree T_2.
Thus $f=5$ and $w=2$. But $f=5=2h-1=f_{max}$ and $w=2=h-1=w_{max}-2$.

ii) Each insertion of key 9 into T_5 requires:
2 retrievals to locate slot for 9 in page 5.
5 pages must be written, namely 1, 2, 3, 5, 6.
Thus

$$f=2=h=f_{max}$$

$$w=5=2h+1=w_{max}.$$

Case 2. Consider a scheme with page overflows.

i) Deletion of key 9 leads to the same results as in Case 1.

ii) Insertion of key 9 requires:
2 retrievals to locate slot for 9 on page 5.
2 retrievals of brothers 4 and 7 of 5 to find out that 5 must be split.
5 pages must be written as in Case 1.
Thus:

$$f=4=3h-2=f_{max}$$

$$w=5=2h+1=w_{max}.$$

Analogous examples can be constructed for arbitrary h and k.

From the analysis it is clear that the performance of our scheme depends on the actual sequence of insertions and deletions. The interference between insertions and deletions may degrade the performance of the scheme as opposed to doing insertions or deletions only. But even in the worst cases this interference degrades the performance at most by a factor of 3.

It is an open question how important this interference is in any actual applications and how relevant our worst case analysis is. Although the derivable cost bounds are worse, the scheme with overflows performed better in our experiments than the scheme without overflows.

10. Choice of k

The performance of our scheme depends on the parameter k. Thus care should be taken in choosing k to make the performance as good as possible.

To obtain a very rough approximation to the performance of the scheme we make the following assumptions:

المنارة للاستشارات

	Retrieval	Insertion in index without deletions and without overflows	Deletion in index without insertions, with or without overflows	Insertion in index without deletions, but with overflow	Insertion in index with deletions, without overflow	Deletion in index with insertions, with or without overflows	Insertion in index with deletion, with overflow
min	$f=1$ $w=0$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$
Average as derived in paper	$f\leqq h$ $w=0$	$f=h$ $w<1+\frac{2}{k}$	$f<h+1+\frac{1}{k}$ $w<4+\frac{2}{k}$	$f\leqq h+2+\frac{2}{k}$ $w\leqq 3+\frac{2}{k}$	$f=h$ $w\leqq 2h+1$	$f\leqq 2h-1$ $h-1\leqq u \leqq h+1$	$f\leqq 3h-2$ $w\leqq 2h+1$
max	$f=h$ $w=0$	$f=h$ $w=2h+1$	$f=2h-1$ $w=h+1$	$f=3h-2$ $w=2h+1$	$f=h$ $w=2h+1$	$f=2h-1$ $w=h+1$	$f=3h-2$ $w=2h+1$

f = number of pages fetched
w = number of pages written
I = size of index set
h = height of B-tree
k = parameter of B-tree of pages
u = best upper bound obtainable for w

Fig. 7. Table of costs for a single retrieval, insertion, or deletion of a key

i) The time spent for each page which is written or fetched can be expressed in the form:

$$\alpha+\beta(2k+1)+\gamma \ln(\nu k+1)$$

α fixed time spent per page, e.g., average disc seek time plus fixed CPU overhead, etc.

β transfer time per page entry.

γ constant for the logarithmic part of the time, e.g., for a binary search.

ν factor for average page occupancy, $1 \leqq \nu \leqq 2$.

We assume that modifying a page does not require moving keys within a page, but that the necessary channel subcommands are generated to write a page by concatenating several pieces of information in main store. This is the reason for our assumption that fetching and writing a page takes the same time.

i) The average number of pages fetched and written per single transaction in an environment of mixed retrievals, insertions, and deletions is approximately proportional—see Fig. 7—to h, say δh. The total time T spent per transaction can then be approximated by:

$$T \approx \delta h\big(\alpha+\beta(2k+1)+\gamma \ln(\nu k+1)\big).$$

Approximating h itself by: $h \approx \log_{\nu k+1}(I+1)$ where I is the size of the index, we get: $T \approx T_a = \delta \log_{\nu k+1}(I+1)\,\big(\alpha+\beta(2k+1)+\gamma \ln(\nu k+1)\big)$.

Now one easily obtains the minimum of T_a if k is chosen such that:

$$\frac{\alpha}{\beta}=2\left(\frac{(\nu k+1)}{\nu}\ln(\nu k+1)\right)-(2k+1)=f(k,\nu).$$

المنارة للاستشارات

Neglecting CPU time, k is a number which is characteristic for the device used as backup store. To obtain a near optimal page size for our test examples we assumed $\alpha = 50$ ms and $\beta = 90$ µs. According to the table in Fig. 8 an acceptable choice should be $64 < k < 128$. For reasons of programming convenience we chose $k = 60$ resulting in a page size of 120 entries.

k	$f(k, 1)$	$f(k, 1.5)$	$f(k, 2)$
2.00000E+00	1.59167E+00	2.39356E+00	3.04718E+00
4.00000E+00	7.09437E+00	9.16182E+00	1.07750E+01
8.00000E+00	2.25500E+01	2.74591E+01	3.11646E+01
1.60000E+01	6.33292E+01	7.42958E+01	8.23847E+01
3.20000E+01	1.65769E+02	1.89265E+02	2.06334E+02
6.40000E+01	4.13670E+02	4.62662E+02	4.97915E+02
1.28000E+02	9.96831E+02	1.09726E+03	1.16911E+03
2.56000E+02	2.33922E+03	2.54299E+03	2.68826E+03
5.12000E+02	5.37752E+03	5.78842E+03	6.08075E+03
1.02400E+03	1.21625E+04	1.29881E+04	1.35748E+04
2.04800E+03	2.71506E+04	2.88062E+04	2.99818E+04
4.09600E+03	5.99647E+04	6.32806E+04	6.56343E+04
8.19200E+03	1.31269E+05	1.37906E+05	1.42617E+05
1.63840E+04	2.85235E+05	2.98514E+05	3.07938E+05
3.27680E+04	6.15877E+05	6.42442E+05	6.61292E+05
6.55360E+04	1.32258E+06	1.37572E+06	1.41342E+06

Fig. 8. The function $f(k, v)$ for optimal choice of k

The size of the index which can be stored for $k = 60$ in a page tree of a certain height can be seen from Fig. 9.

Height of page tree	Minimum index size	Maximum index size
1	1	120
2	121	14640
3	7441	1771560
4	453961	214358880

Fig. 9. Height of page tree and index size

11. Experimental Results

The algorithms presented here were programmed and their performance measured during various experiments. The programs were run on an IBM 360/44 computer with a 2311 disc unit as a backup store. For the index element size chosen (14 8-bit characters) and index size generally used (about 10000 index elements), the average access mechanism delay for this unit is about 50 ms, after which information transfer takes place at the rate of about 90 µs per index element. From these two parameters, our analysis predicts an optimal page size ($2k$) on the order of 120 index elements.

المنارة للاستشارات

The programming included a simple demand paging scheme to take advantage of available core storage (about 1250 index elements' worth) and thus to attempt to reduce the number of physical disc operations. In the following section by *virtual disc read* we mean a request to the paging scheme that a certain disc page be available in core; a virtual disc read will result in a physical disc read only of there is no copy of the requested disc page already in the paging area of core storage. A *virtual disc write* is defined analogously.

At the time of this writing ten experiments had been performed. These experiments were intended to give us an idea of what kind of performance to expect, what kind of storage utilization to expect, and so forth. For us the specification of an experiment consists of choosing

1) whether or not to permit overflows on insertion,
2) a number of index elements per page, and
3) a sequence of transactions to be made against an initially empty index.

At several points during the performance of an experiment certain performance variables are recorded. From these the performance of the algorithms according to various performance measures can be deduced; to wit

1) % storage utilization
2) average number of virtual disc reads/transaction
3) average number of physical disc reads/transaction
4) average number of virtual disc writes/insertion or deletion
5) average number of physical disc writes/insertion or deletion
6) average number of transactions/second.

We now summarize the experiments. Each experiment was divided into several phases, and at the end of each of these the performance variables were measured. Phases are denoted by numbers within parentheses.

$E1$: 25 elements/page, overflow permitted.
- (1) 10000 insertions sequential by key,
- (2) 50 insertions, 50 retrievals, and 100 deletions uniformly random in the key space.

$E2$: 120 elements/page; otherwise identical to $E1$.

$E3$: 250 elements/page; otherwise identical to $E1$.

$E4$: 120 elements/page, overflow permitted.
- (1) 10000 insertions sequential by key,
- (2) 1000 retrievals uniformly random in key space,
- (3) 10000 sequential deletions.

$E5$: 120 elements/page, overflow *not* permitted.
- (1) 5000 insertions uniformly random in key space,
- (2) 1000 retrievals uniformly random in key space,
- (3) 5000 deletions uniformly random in key space.

$E6$: Overflow permitted; otherwise identical to $E5$.

$E7$: 120 elements/page, overflow permitted.
- (1) 5000 insertions sequential by key,
- (2) 6000 each insertions, retrievals, and deletions uniformly random in key space.

المنارة للاستشارات

$E8$: 120 elements/page, overflow permitted.

(1) 15000 insertions uniformly random in key space,

(2) 100 each insertions, deletions, and retrievals uniformly random in key space.

$E9$: 250 elements/page; otherwise identical to $E8$.

$E10$: 120 elements/page, overflow permitted.

(1) 100000 insertions sequential by key,

(2) 1000 each insertions, deletions, and retrievals uniformly random in key space,

(3) 100 group retrievals uniformly random in key space, where a group is a sequence of 100 consecutive keys (statistics on the basis of 10000 transactions),

(4) 10000 insertions sequential by key, to merge uniformly with the elements inserted in phase (1).

	% Storage used	VR/T*	PR/T	VW/I or D	PW/I or D	T/sec
$E1$ (1)	99.8	2.2	0	2.3	0.04	66.1
$E1$ (2)	91.5	4.4	1.62	2.7	1.5	6.6
$E2$ (1)	99.2	1.0	0	1.0	0.008	94.5
$E2$ (2)	87.3	2.5	1.15	1.3	1.1	6.7
$E3$ (1)	97.6	1.0	0	1.0	0.004	100.0
$E3$ (2)	84.7	2.4	1.08	1.3	1.1	5.2
$E4$ (1)	99.2	1.0	0	1.0	0.008	94.5
$E4$ (2)	99.2	2.0	—	—	—	19.5
$E4$ (3)	—	2.0	0.01	2.0	0	74.1
$E5$ (1)	67.1	1.0	0.55	1.0	0.56	17.0
$E5$ (2)	67.1	2.0	0.83	—	—	18.2
$E5$ (3)	—	4.0	0.68	2.2	0.65	12.4
$E6$ (1)	86.7	1.1	0.55	1.1	0.54	17.1
$E6$ (2)	86.7	2.0	0.79	—	—	24.3
$E6$ (3)	—	4.0	0.65	2.2	0.62	13.4
$E7$ (1)	96.9	1.0	0	1.0	0.008	111.9
$E7$ (2)	76.8	2.3	0.83	1.3	0.88	13.1
$E8$ (1)	84.5	1.3	0.87	1.3	0.85	10.1
$E8$ (2)	83.9	3.7	1.00	3.0	1.00	9.5
$E9$ (1)	86.4	1.1	0.84	1.0	0.82	8.5
$E9$ (2)	85.2	2.3	0.94	1.1	0.96	8.2
$E10$ (1)	99.8	1.9	0	1.9	0.008	91.7
$E10$ (2)	82.1	4.1	1.94	1.8	1.54	4.2
$E10$ (3)	82.1	4.0	0.03	—	—	75.7
$E10$ (4)	83.8	2.2	0.10	2.2	0.11	38.0

* This statistic is unnecessarily large for deletions, due to the way deletions were programmed. To find the *necessary* number of virtual reads, for sequential deletions subtract one from the number shown, and for random deletions subtract one and multiply the result by about 0.5.

المنارة للاستشارات

References

1. Adelson-Velskii, G. M., Landis, E. M.: An information organization algorithm. DANSSSR, **146**, 263–266 (1962).
2. Foster, C. C.: Information storage and retrieval using AVL trees. Proc. ACM 20th Nat'l. Conf. 192–205 (1965).
3. Landauer, W. I.: The balanced tree and its utilization in information retrieval. IEEE Trans. on Electronic Computers, Vol. EC-12, No. 6, December 1963.
4. Sussenguth, E. H., Jr.: The use of tree structures for processing files. Comm. ACM, **6**, No. 5, May 1963.

Prof. Dr. R. Bayer
Dept. of Computer Science
Purdue University
Lafayette, Ind. 47907
U.S.A.

Dr. E M. McCreight
Palo Alto Research Center
3180 Porter Drive
Palo Alto, Calif. 94304
U.S.A.

للاستشارات

E.F. Codd

A Relational Model of Data for Large Shared Data Banks

Communications of the ACM, Vol. 13 (6), 1970
pp. 377–387

المنارة للاستشارات

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

للاستشارات

1. Relational Model and Normal Form

1.1. Introduction

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted

المنارة للاستشارات

data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. Data Dependencies in Present Systems

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

المنارة للاستشارات

stored ordering. Those application programs which take advantage of the stored ordering of a file are likely to fail to operate correctly if for some reason it becomes necessary to replace that ordering by a different one. Similar remarks hold for a stored ordering implemented by means of pointers.

It is unnecessary to single out any system as an example, because all the well-known information systems that are marketed today fail to make a clear distinction between order of presentation on the one hand and stored ordering on the other. Significant implementation problems must be solved to provide this kind of independence.

1.2.2. *Indexing Dependence.* In the context of formatted data, an index is usually thought of as a purely performance-oriented component of the data representation. It tends to improve response to queries and updates and, at the same time, slow down response to insertions and deletions. From an informational standpoint, an index is a redundant component of the data representation. If a system uses indices at all and if it is to perform well in an environment with changing patterns of activity on the data bank, an ability to create and destroy indices from time to time will probably be necessary. The question then arises: Can application programs and terminal activities remain invariant as indices come and go?

Present formatted data systems take widely different approaches to indexing. TDMS [7] unconditionally provides indexing on all attributes. The presently released version of IMS [5] provides the user with a choice for each file: a choice between no indexing at all (the hierarchic sequential organization) or indexing on the primary key only (the hierarchic indexed sequential organization). In neither case is the user's application logic dependent on the existence of the unconditionally provided indices. IDS [8], however, permits the file designers to select attributes

المنارة للاستشارات

to be indexed and to incorporate indices into the file structure by means of additional chains. Application programs taking advantage of the performance benefit of these indexing chains must refer to those chains by name. Such programs do not operate correctly if these chains are later removed.

1.2.3. *Access Path Dependence.* Many of the existing formatted data systems provide users with tree-structured files or slightly more general network models of the data. Application programs developed to work with these systems tend to be logically impaired if the trees or networks are changed in structure. A simple example follows.

Suppose the data bank contains information about parts and projects. For each part, the part number, part name, part description, quantity-on-hand, and quantity-on-order are recorded. For each project, the project number, project name, project description are recorded. Whenever a project makes use of a certain part, the quantity of that part committed to the given project is also recorded. Suppose that the system requires the user or file designer to declare or define the data in terms of tree structures. Then, any one of the hierarchical structures may be adopted for the information mentioned above (see Structures 1–5).

Structure 1. Projects Subordinate to Parts

File	*Segment*	*Fields*
F	PART	part #
		part name
		part description
		quantity-on-hand
		quantity-on-order
	PROJECT	project #
		project name
		project description
		quantity committed

المنارة للاستشارات

Structure 2. Parts Subordinate to Projects

File	*Segment*	*Fields*
F	PROJECT	project #
		project name
		project description
	PART	part #
		part name
		part description
		quantity-on-hand
		quantity-on-order
		quantity committed

Structure 3. Parts and Projects as Peers Commitment Relationship Subordinate to Projects

File	*Segment*	*Fields*
F	PART	part #
		part name
		part description
		quantity-on-hand
		quantity-on-order
G	PROJECT	project #
		project name
		project description
	PART	part #
		quantity committed

Structure 4. Parts and Projects as Peers Commitment Relationship Subordinate to Parts

File	*Segment*	*Fields*
F	PART	part #
		part description
		quantity-on-hand
		quantity-on-order
	PROJECT	project #
		quantity committed
G	PROJECT	project #
		project name
		project description

المنارة للاستشارات

Structure 5. Parts, Projects, and Commitment Relationship as Peers

File	*Segment*	*Fields*
F	PART	part # part name part description quantity-on-hand quantity-on-order
G	PROJECT	project # project name project description
H	COMMIT	part # project # quantity committed

Now, consider the problem of printing out the part number, part name, and quantity committed for every part used in the project whose project name is "alpha." The following observations may be made regardless of which available tree-oriented information system is selected to tackle this problem. If a program *P* is developed for this problem assuming one of the five structures above—that is, *P* makes no test to determine which structure is in effect—then *P* will fail on at least three of the remaining structures. More specifically, if *P* succeeds with structure 5, it will fail with all the others; if *P* succeeds with structure 3 or 4, it will fail with at least 1, 2, and 5; if *P* succeeds with 1 or 2, it will fail with at least 3, 4, and 5. The reason is simple in each case. In the absence of a test to determine which structure is in effect, *P* fails because an attempt is made to exceute a reference to a nonexistent file (available systems treat this as an error) or no attempt is made to execute a reference to a file containing needed information. The reader who is not convinced should develop sample programs for this simple problem.

المنارة للاستشارات

Since, in general, it is not practical to develop application programs which test for all tree structurings permitted by the system, these programs fail when a change in structure becomes necessary.

Systems which provide users with a network model of the data run into similar difficulties. In both the tree and network cases, the user (or his program) is required to exploit a collection of user access paths to the data. It does not matter whether these paths are in close correspondence with pointer-defined paths in the stored representation—in IDS the correspondence is extremely simple, in TDMS it is just the opposite. The consequence, regardless of the stored representation, is that terminal activities and programs become dependent on the continued existence of the user access paths.

One solution to this is to adopt the policy that once a user access path is defined it will not be made obsolete until all application programs using that path have become obsolete. Such a policy is not practical, because the number of access paths in the total model for the community of users of a data bank would eventually become excessively large.

1.3. A Relational View of Data

The term *relation* is used here in its accepted mathematical sense. Given sets $S_1, S_2, \cdots, S_n$ (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on.[1] We shall refer to S_j as the jth *domain* of R. As defined above, R is said to have *degree* n. Relations of degree 1 are often called *unary*, degree 2 *binary*, degree 3 *ternary*, and degree n *n-ary*.

[1] More concisely, R is a subset of the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$.

المنارة للاستشارات

For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded. An array which represents an n-ary relation R has the following properties:

(1) Each row represents an n-tuple of R.
(2) The ordering of rows is immaterial.
(3) All rows are distinct.
(4) The ordering of columns is significant—it corresponds to the ordering $S_1, S_2, \cdots, S_n$ of the domains on which R is defined (see, however, remarks below on domain-ordered and domain-unordered relations).
(5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

The example in Figure 1 illustrates a relation of degree 4, called *supply*, which reflects the shipments-in-progress of parts from specified suppliers to specified projects in specified quantities.

supply	(*supplier*	*part*	*project*	*quantity*)
	1	2	5	17
	1	3	5	23
	2	3	7	9
	2	7	5	4
	4	1	1	12

FIG. 1. A relation of degree 4

One might ask: If the columns are labeled by the name of corresponding domains, why should the ordering of columns matter? As the example in Figure 2 shows, two columns may have identical headings (indicating identical domains) but possess distinct meanings with respect to the relation. The relation depicted is called *component*. It is a

المنارة للاستشارات

component	(*part*	*part*	*quantity*)
	1	5	9
	2	5	7
	3	5	2
	2	6	12
	3	6	3
	4	7	1
	6	7	1

FIG. 2. A relation with two identical domains

ternary relation, whose first two domains are called *part* and third domain is called *quantity*. The meaning of *component* (x, y, z) is that part x is an immediate component (or subassembly) of part y, and z units of part x are needed to assemble one unit of part y. It is a relation which plays a critical role in the parts explosion problem.

It is a remarkable fact that several existing information systems (chiefly those based on tree-structured files) fail to provide data representations for relations which have two or more identical domains. The present version of IMS/360 [5] is an example of such a system.

The totality of data in a data bank may be viewed as a collection of time-varying relations. These relations are of assorted degrees. As time progresses, each n-ary relation may be subject to insertion of additional n-tuples, deletion of existing ones, and alteration of components of any of its existing n-tuples.

In many commercial, governmental, and scientific data banks, however, some of the relations are of quite high degree (a degree of 30 is not at all uncommon). Users should not normally be burdened with remembering the domain ordering of any relation (for example, the ordering *supplier*, then *part*, then *project*, then *quantity* in the relation *supply*). Accordingly, we propose that users deal, not with relations which are domain-ordered, but with *relationships* which are

المنارة للاستشارات

their domain-unordered counterparts.[2] To accomplish this, domains must be uniquely identifiable at least within any given relation, without using position. Thus, where there are two or more identical domains, we require in each case that the domain name be qualified by a distinctive *role name*, which serves to identify the role played by that domain in the given relation. For example, in the relation *component* of Figure 2, the first domain *part* might be qualified by the role name *sub*, and the second by *super*, so that users could deal with the relationship *component* and its domains—*sub.part super.part*, *quantity*—without regard to any ordering between these domains.

To sum up, it is proposed that most users should interact with a relational model of the data consisting of a collection of time-varying relationships (rather than relations). Each user need not know more about any relationship than its name together with the names of its domains (role qualified whenever necessary).[3] Even this information might be offered in menu style by the system (subject to security and privacy constraints) upon request by the user.

There are usually many alternative ways in which a relational model may be established for a data bank. In order to discuss a preferred way (or normal form), we must first introduce a few additional concepts (active domain, primary key, foreign key, nonsimple domain) and establish some links with terminology currently in use in information systems programming. In the remainder of this paper, we shall not bother to distinguish between re-

[2] In mathematical terms, a relationship is an equivalence class of those relations that are equivalent under permutation of domains (see Section 2.1.1).

[3] Naturally, as with any data put into and retrieved from a computer system, the user will normally make far more effective use of the data if he is aware of its meaning.

المنارة للاستشارات

lations and relationships except where it appears advantageous to be explicit.

Consider an example of a data bank which includes relations concerning parts, projects, and suppliers. One relation called *part* is defined on the following domains:

(1) part number
(2) part name
(3) part color
(4) part weight
(5) quantity on hand
(6) quantity on order

and possibly other domains as well. Each of these domains is, in effect, a pool of values, some or all of which may be represented in the data bank at any instant. While it is conceivable that, at some instant, all part colors are present, it is unlikely that all possible part weights, part names, and part numbers are. We shall call the set of values represented at some instant the *active domain* at that instant.

Normally, one domain (or combination of domains) of a given relation has values which uniquely identify each element (n-tuple) of that relation. Such a domain (or combination) is called a *primary key*. In the example above, part number would be a primary key, while part color would not be. A primary key is *nonredundant* if it is either a simple domain (not a combination) or a combination such that none of the participating simple domains is superfluous in uniquely identifying each element. A relation may possess more than one nonredundant primary key. This would be the case in the example if different parts were always given distinct names. Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called *the* primary key of that relation.

المنارة للاستشارات

A common requirement is for elements of a relation to cross-reference other elements of the same relation or elements of a different relation. Keys provide a user-oriented means (but not the only means) of expressing such cross-references. We shall call a domain (or domain combination) of relation R a *foreign key* if it is not the primary key of R but its elements are values of the primary key of some relation S (the possibility that S and R are identical is not excluded). In the relation *supply* of Figure 1, the combination of *supplier*, *part*, *project* is the primary key, while each of these three domains taken separately is a foreign key.

In previous work there has been a strong tendency to treat the data in a data bank as consisting of two parts, one part consisting of entity descriptions (for example, descriptions of suppliers) and the other part consisting of relations between the various entities or types of entities (for example, the *supply* relation). This distinction is difficult to maintain when one may have foreign keys in any relation whatsoever. In the user's relational model there appears to be no advantage to making such a distinction (there may be some advantage, however, when one applies relational concepts to machine representations of the user's set of relationships).

So far, we have discussed examples of relations which are defined on simple domains—domains whose elements are atomic (nondecomposable) values. Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on. For example, one of the domains on which the relation *employee* is defined might be *salary history*. An element of the salary history domain is a binary relation defined on the domain *date* and the domain *salary*. The *salary history* domain is the set of all such binary relations. At any instant of time there are as many instances of the *salary history* relation

المنارة للاستشارات

in the data bank as there are employees. In contrast, there is only one instance of the *employee* relation.

The terms attribute and repeating group in present data base terminology are roughly analogous to simple domain and nonsimple domain, respectively. Much of the confusion in present terminology is due to failure to distinguish between type and instance (as in "record") and between components of a user model of the data on the one hand and their machine representation counterparts on the other hand (again, we cite "record" as an example).

1.4. Normal Form

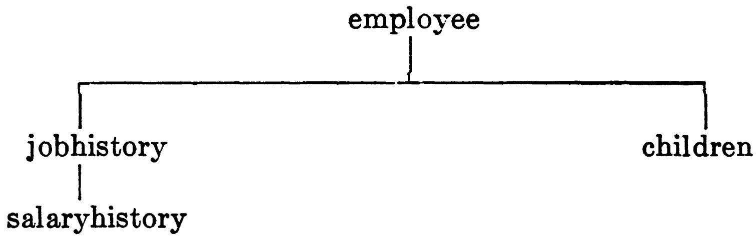
A relation whose domains are all simple can be represented in storage by a two-dimensional column-homogeneous array of the kind discussed above. Some more complicated data structure is necessary for a relation with one or more nonsimple domains. For this reason (and others to be cited below) the possibility of eliminating nonsimple domains appears worth investigating.[4] There is, in fact, a very simple elimination procedure, which we shall call normalization.

Consider, for example, the collection of relations exhibited in Figure 3(a). *Job history* and *children* are nonsimple domains of the relation *employee*. *Salary history* is a nonsimple domain of the relation *job history*. The tree in Figure 3(a) shows just these interrelationships of the nonsimple domains.

Normalization proceeds as follows. Starting with the relation at the top of the tree, take its primary key and expand each of the immediately subordinate relations by inserting this primary key domain or domain combination. The primary key of each expanded relation consists of the primary key before expansion augmented by the primary

[4] M. E. Sanko of IBM, San Jose, independently recognized the desirability of eliminating nonsimple domains.

المنارة للاستشارات



employee (*man#*, name, birthdate, jobhistory, children)
jobhistory (*jobdate*, title, salaryhistory)
salaryhistory (*salarydate*, salary)
children (*childname*, birthyear)

FIG. 3(a). Unnormalized set

employee′ (*man#*, name, birthdate)
jobhistory′ (*man#*, *jobdate*, title)
salaryhistory′ (*man#*, *jobdate*, *salarydate*, salary)
children′ (*man#*, *childname*, birthyear)

FIG. 3(b). Normalized set

key copied down from the parent relation. Now, strike out from the parent relation all nonsimple domains, remove the top node of the tree, and repeat the same sequence of operations on each remaining subtree.

The result of normalizing the collection of relations in Figure 3(a) is the collection in Figure 3(b). The primary key of each relation is italicized to show how such keys are expanded by the normalization.

If normalization as described above is to be applicable, the unnormalized collection of relations must satisfy the following conditions:

(1) The graph of interrelationships of the nonsimple domains is a collection of trees.

(2) No primary key has a component domain which is nonsimple.

المنارة للاستشارات

The writer knows of no application which would require any relaxation of these conditions. Further operations of a normalizing kind are possible. These are not discussed in this paper.

The simplicity of the array representation which becomes feasible when all relations are cast in normal form is not only an advantage for storage purposes but also for communication of bulk data between systems which use widely different representations of the data. The communication form would be a suitably compressed version of the array representation and would have the following advantages:

(1) It would be devoid of pointers (address-valued or displacement-valued).

(2) It would avoid all dependence on hash addressing schemes.

(3) It would contain no indices or ordering lists.

If the user's relational model is set up in normal form, names of items of data in the data bank can take a simpler form than would otherwise be the case. A general name would take a form such as

$$R(g).r.d$$

where R is a relational name; g is a generation identifier (optional); r is a role name (optional); d is a domain name. Since g is needed only when several generations of a given relation exist, or are anticipated to exist, and r is needed only when the relation R has two or more domains named d, the simple form $R.d$ will often be adequate.

1.5. Some Linguistic Aspects

The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus. A first-order predicate calculus suffices if the collection of relations is in normal form. Such a language would provide a yard-

المنارة للاستشارات

stick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented). While it is not the purpose of this paper to describe such a language in detail, its salient features would be as follows.

Let us denote the data sublanguage by R and the host language by H. R permits the declaration of relations and their domains. Each declaration of a relation identifies the primary key for that relation. Declared relations are added to the system catalog for use by any members of the user community who have appropriate authorization. H permits supporting declarations which indicate, perhaps less permanently, how these relations are represented in storage. R permits the specification for retrieval of any subset of data from the data bank. Action on such a retrieval request is subject to security constraints.

The universality of the data sublanguage lies in its descriptive ability (not its computing ability). In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval. Thus, the class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus. It is well known that to preserve this descriptive power it is unnecessary to express (in whatever syntax is chosen) every formula of the selected predicate calculus. For example, just those in prenex normal form are adequate [9].

Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in H and invoked in R.

المنارة للاستشارات

A set so specified may be fetched for query purposes only, or it may be held for possible changes. Insertions take the form of adding new elements to declared relations without regard to any ordering that may be present in their machine representation. Deletions which are effective for the community (as opposed to the individual user or subcommunities) take the form of removing elements from declared relations. Some deletions and updates may be triggered by others, if deletion and update dependencies between specified relations are declared in R.

One important effect that the view adopted toward data has on the language used to retrieve it is in the naming of data elements and sets. Some aspects of this have been discussed in the previous section. With the usual network view, users will often be burdened with coining and using more relation names than are absolutely necessary, since names are associated with paths (or path types) rather than with relations.

Once a user is aware that a certain relation is stored, he will expect to be able to exploit[5] it using any combination of its arguments as "knowns" and the remaining arguments as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems) which we shall call (logically) *symmetric exploitation* of relations. Naturally, symmetry in performance is not to be expected.

To support symmetric exploitation of a single binary relation, two directed paths are needed. For a relation of degree n, the number of paths to be named and controlled is n factorial.

Again, if a relational view is adopted in which every n-ary relation ($n > 2$) has to be expressed by the user as a nested expression involving only binary relations (see

[5] Exploiting a relation includes query, update, and delete.

المنارة للاستشارات

Feldman's LEAP System [10], for example) then $2n - 1$ names have to be coined instead of only $n + 1$ with direct n-ary notation as described in Section 1.2. For example, the 4-ary relation *supply* of Figure 1, which entails 5 names in n-ary notation, would be represented in the form

$$P\ (supplier,\ Q\ (part,\ R\ (project,\ quantity)))$$

in nested binary notation and, thus, employ 7 names.

A further disadvantage of this kind of expression is its asymmetry. Although this asymmetry does not prohibit symmetric exploitation, it certainly makes some bases of interrogation very awkward for the user to express (consider, for example, a query for those parts and quantities related to certain given projects via Q and R).

1.6. Expressible, Named, and Stored Relations

Associated with a data bank are two collections of relations: the *named set* and the *expressible set*. The named set is the collection of all those relations that the community of users can identify by means of a simple name (or identifier). A relation R acquires membership in the named set when a suitably authorized user declares R; it loses membership when a suitably authorized user cancels the declaration of R.

The expressible set is the total collection of relations that can be designated by expressions in the data language. Such expressions are constructed from simple names of relations in the named set; names of generations, roles and domains; logical connectives; the quantifiers of the predicate calculus;[6] and certain constant relation symbols such as =, >.

[6] Because each relation in a practical data bank is a finite set at every instant of time, the existential and universal quantifiers can be expressed in terms of a function that counts the number of elements in any finite set.

المنارة للاستشارات

The named set is a subset of the expressible set—usually a very small subset.

Since some relations in the named set may be time-independent combinations of others in that set, it is useful to consider associating with the named set a collection of statements that define these time-independent constraints. We shall postpone further discussion of this until we have introduced several operations on relations (see Section 2).

One of the major problems confronting the designer of a data system which is to support a relational model for its users is that of determining the class of stored representations to be supported. Ideally, the variety of permitted data representations should be just adequate to cover the spectrum of performance requirements of the total collection of installations. Too great a variety leads to unnecessary overhead in storage and continual reinterpretation of descriptions for the structures currently in effect.

For any selected class of stored representations the data system must provide a means of translating user requests expressed in the data language of the relational model into corresponding—and efficient—actions on the current stored representation. For a high level data language this presents a challenging design problem. Nevertheless, it is a problem which must be solved—as more users obtain concurrent access to a large data bank, responsibility for providing efficient response and throughput shifts from the individual user to the data system.

2. Redundancy and Consistency

2.1. Operations on Relations

Since relations are sets, all of the usual set operations are applicable to them. Nevertheless, the result may not be a relation; for example, the union of a binary relation and a ternary relation is not a relation.

المنارة للاستشارات

The operations discussed below are specifically for relations. These operations are introduced because of their key role in deriving relations from other relations. Their principal application is in noninferential information systems—systems which do not provide logical inference services—although their applicability is not necessarily destroyed when such services are added.

Most users would not be directly concerned with these operations. Information systems designers and people concerned with data bank control should, however, be thoroughly familiar with them.

2.1.1. *Permutation.* A binary relation has an array representation with two columns. Interchanging these columns yields the converse relation. More generally, if a permutation is applied to the columns of an n-ary relation, the resulting relation is said to be a *permutation* of the given relation. There are, for example, $4! = 24$ permutations of the relation *supply* in Figure 1, if we include the identity permutation which leaves the ordering of columns unchanged.

Since the user's relational model consists of a collection of relationships (domain-unordered relations), permutation is not relevant to such a model considered in isolation. It is, however, relevant to the consideration of stored representations of the model. In a system which provides symmetric exploitation of relations, the set of queries answerable by a stored relation is identical to the set answerable by any permutation of that relation. Although it is logically unnecessary to store both a relation and some permutation of it, performance considerations could make it advisable.

2.1.2. *Projection.* Suppose now we select certain columns of a relation (striking out the others) and then remove from the resulting array any duplication in the rows. The final array represents a relation which is said to be a *projection* of the given relation.

A selection operator π is used to obtain any desired permutation, projection, or combination of the two operations. Thus, if L is a list of k indices[7] $L = i_1, i_2, \cdots, i_k$ and R is an n-ary relation ($n \geq k$), then $\pi_L(R)$ is the k-ary relation whose jth column is column i_j of R ($j = 1, 2, \cdots, k$) except that duplication in resulting rows is removed. Consider the relation *supply* of Figure 1. A permuted projection of this relation is exhibited in Figure 4. Note that, in this particular case, the projection has fewer n-tuples than the relation from which it is derived.

2.1.3. *Join.* Suppose we are given two binary relations, which have some domain in common. Under what circumstances can we combine these relations to form a ternary relation which preserves all of the information in the given relations?

The example in Figure 5 shows two relations R, S, which are joinable without loss of information, while Figure 6 shows a join of R with S. A binary relation R is *joinable* with a binary relation S if there exists a ternary relation U such that $\pi_{12}(U) = R$ and $\pi_{23}(U) = S$. Any such ternary relation is called a *join* of R with S. If R, S are binary relations such that $\pi_2(R) = \pi_1(S)$, then R is joinable with S. One join that always exists in such a case is the *natural join* of R with S defined by

$$R*S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}$$

where $R(a, b)$ has the value *true* if (a, b) is a member of R and similarly for $S(b, c)$. It is immediate that

$$\pi_{12}(R*S) = R$$

and

$$\pi_{23}(R*S) = S.$$

[7] When dealing with relationships, we use domain names (role-qualified whenever necessary) instead of domain positions.

المنارة للاستشارات

Π_{31}(*supply*)	(*project*	*supplier*)
	5	1
	5	2
	1	4
	7	2

FIG. 4. A permuted projection of the relation in Figure 1

R	(*supplier*	*part*)
	1	1
	2	1
	2	2

S	(*part*	*project*)
	1	1
	1	2
	2	1

FIG. 5. Two joinable relations

R*S	(*supplier*	*part*	*project*)
	1	1	1
	1	1	2
	2	1	1
	2	1	2
	2	2	1

FIG. 6. The natural join of R with S (from Figure 5)

U	(*supplier*	*part*	*project*)
	1	1	2
	2	1	1
	2	2	1

FIG. 7. Another join of R with S (from Figure 5)

Note that the join shown in Figure 6 is the natural join of *R* with *S* from Figure 5. Another join is shown in Figure 7.

Inspection of these relations reveals an element (element 1) of the domain *part* (the domain on which the join

المنارة للاستشارات

is to be made) with the property that it possesses more than one relative under R and also under S. It is this element which gives rise to the plurality of joins. Such an element in the joining domain is called a *point of ambiguity* with respect to the joining of R with S.

If either $\pi_{21}(R)$ or S is a function,[8] no point of ambiguity can occur in joining R with S. In such a case, the natural join of R with S is the only join of R with S. Note that the reiterated qualification "of R with S" is necessary, because S might be joinable with R (as well as R with S), and this join would be an entirely separate consideration. In Figure 5, none of the relations R, $\pi_{21}(R)$, S, $\pi_{21}(S)$ is a function.

Ambiguity in the joining of R with S can sometimes be resolved by means of other relations. Suppose we are given, or can derive from sources independent of R and S, a relation T on the domains *project* and *supplier* with the following properties:

$$(1)\quad \pi_1(T) = \pi_2(S),$$

$$(2)\quad \pi_2(T) = \pi_1(R),$$

$$(3)\quad T(j, s) \rightarrow \exists p(R(S, p) \wedge S(p, j)),$$

$$(4)\quad R(s, p) \rightarrow \exists j(S(p, j) \wedge T(j, s)),$$

$$(5)\quad S(p, j) \rightarrow \exists s(T(j, s) \wedge R(s, p)),$$

then we may form a three-way join of R, S, T; that is, a ternary relation such that

$$\pi_{12}(U) = R, \qquad \pi_{23}(U) = S, \qquad \pi_{31}(U) = T.$$

Such a join will be called a *cyclic* 3-*join* to distinguish it from a *linear* 3-*join* which would be a quaternary relation V such that

$$\pi_{12}(V) = R, \qquad \pi_{23}(V) = S, \qquad \pi_{34}(V) = T.$$

[8] A function is a binary relation, which is one-one or many-one, but not one-many.

المنارة للاستشارات

R	(*s*	*p*)
	1	a
	2	a
	2	b

S	(*p*	*j*)
	a	d
	a	e
	b	d
	b	e

T	(*j*	*s*)
	d	1
	d	2
	e	2
	e	2

FIG. 8. Binary relations with a plurality of cyclic 3-joins

U	(*s*	*p*	*j*)
	1	a	d
	2	a	e
	2	b	d
	2	b	e

U′	(*s*	*p*	*j*)
	1	a	d
	2	a	d
	2	a	e
	2	b	d
	2	b	e

FIG. 9. Two cyclic 3-joins of the relations in Figure 8

While it is possible for more than one cyclic 3-join to exist (see Figures 8, 9, for an example), the circumstances under which this can occur entail much more severe constraints than those for a plurality of 2-joins. To be specific, the relations R, S, T must possess points of ambiguity with respect to joining R with S (say point x), S with T (say y), and T with R (say z), and, furthermore, y must be a relative of x under S, z a relative of y under T, and x a relative of z under R. Note that in Figure 8 the points $x = a$; $y = d$; $z = 2$ have this property.

The natural linear 3-join of three binary relations R, S, T is given by

$$R{*}S{*}T = \{(a, b, c, d){:}R(a, b) \wedge \mathrm{S}(b, c) \wedge T(c, d)\}$$

where parentheses are not needed on the left-hand side because the natural 2-join ($*$) is associative. To obtain the cyclic counterpart, we introduce the operator γ which produces a relation of degree $n - 1$ from a relation of degree n

المنارة للاستشارات

by tying its ends together. Thus, if R is an n-ary relation $(n \geq 2)$, the *tie* of R is defined by the equation

$$\gamma(R) = \{(a_1, a_2, \cdots, a_{n-1}): R(a_1, a_2, \cdots, a_{n-1}, a_n) \wedge a_1 = a_n\}.$$

We may now represent the natural cyclic 3-join of R, S, T by the expression

$$\gamma(R*S*T).$$

Extension of the notions of linear and cyclic 3-join and their natural counterparts to the joining of n binary relations (where $n \geq 3$) is obvious. A few words may be appropriate, however, regarding the joining of relations which are not necessarily binary. Consider the case of two relations R (degree r), S (degree s) which are to be joined on p of their domains $(p < r, \quad p < s)$. For simplicity, suppose these p domains are the last p of the r domains of R, and the first p of the s domains of S. If this were not so, we could always apply appropriate permutations to make it so. Now, take the Cartesian product of the first r-p domains of R, and call this new domain A. Take the Cartesian product of the last p domains of R, and call this B. Take the Cartesian product of the last s-p domains of S and call this C.

We can treat R as if it were a binary relation on the domains A, B. Similarly, we can treat S as if it were a binary relation on the domains B, C. The notions of linear and cyclic 3-join are now directly applicable. A similar approach can be taken with the linear and cyclic n-joins of n relations of assorted degrees.

2.1.4. *Composition.* The reader is probably familiar with the notion of composition applied to functions. We shall discuss a generalization of that concept and apply it first to binary relations. Our definitions of composition

المنارة للاستشارات

and composability are based very directly on the definitions of join and joinability given above.

Suppose we are given two relations R, S. T is a *composition* of R with S if there exists a join U of R with S such that $T = \pi_{13}(U)$. Thus, two relations are composable if and only if they are joinable. However, the existence of more than one join of R with S does not imply the existence of more than one composition of R with S.

Corresponding to the natural join of R with S is the *natural composition*[9] of R with S defined by

$$R \cdot S = \pi_{13}(R * S).$$

Taking the relations R, S from Figure 5, their natural composition is exhibited in Figure 10 and another composition is exhibited in Figure 11 (derived from the join exhibited in Figure 7).

R·S	(*project*	*supplier*)
	1	1
	1	2
	2	1
	2	2

FIG. 10. The natural composition of R with S (from Figure 5)

T	(*project*	*supplier*)
	1	2
	2	1

FIG. 11. Another composition of R with S (from Figure 5)

When two or more joins exist, the number of distinct compositions may be as few as one or as many as the number of distinct joins. Figure 12 shows an example of two relations which have several joins but only one composition.

[9] Other writers tend to ignore compositions other than the natural one, and accordingly refer to this particular composition as *the* composition—see, for example, Kelley's "General Topology."

المنارة للاستشارات

R	(*supplier*	*part*)	S	(*part*	*project*)
	1	a		a	g
	1	b		b	f
	1	c		c	f
	2	c		c	g
	2	d		d	g
	2	e		e	f

FIG. 12. Many joins, only one composition

Note that the ambiguity of point c is lost in composing R with S, because of unambiguous associations made via the points a, b, d, e.

Extension of composition to pairs of relations which are not necessarily binary (and which may be of different degrees) follows the same pattern as extension of pairwise joining to such relations.

A lack of understanding of relational composition has led several systems designers into what may be called the *connection trap*. This trap may be described in terms of the following example. Suppose each supplier description is linked by pointers to the descriptions of each part supplied by that supplier, and each part description is similarly linked to the descriptions of each project which uses that part. A conclusion is now drawn which is, in general, erroneous: namely that, if all possible paths are followed from a given supplier via the parts he supplies to the projects using those parts, one will obtain a valid set of all projects supplied by that supplier. Such a conclusion is correct only in the very special case that the target relation between projects and suppliers is, in fact, the natural composition of the other two relations—and we must normally add the phrase "for all time," because this is usually implied in claims concerning path-following techniques.

المنارة للاستشارات

2.1.5. *Restriction.* A subset of a relation is a relation. One way in which a relation S may act on a relation R to generate a subset of R is through the operation *restriction* of R by S. This operation is a generalization of the restriction of a function to a subset of its domain, and is defined as follows.

Let L, M be equal-length lists of indices such that $L = i_1, i_2, \cdots, i_k$, $M = j_1, j_2, \cdots, j_k$ where $k \leqq$ degree of R and $k \leqq$ degree of S. Then the L, M restriction of R by S denoted $R_L|_M S$ is the maximal subset R' of R such that

$$\pi_L(R') = \pi_M(S).$$

The operation is defined only if equality is applicable between elements of $\pi_{i_h}(R)$ on the one hand and $\pi_{j_h}(S)$ on the other for all $h = 1, 2, \cdots, k$.

The three relations R, S, R' of Figure 13 satisfy the equation $R' = R_{(2,3)}|_{(1,2)}S$.

R	(*s*	*p*	*j*)	S	(*p*	*j*)	R′	(*s*	*p*	*j*)
	1	a	A		a	A		1	a	A
	2	a	A		c	B		2	a	A
	2	a	B		b	B		2	b	B
	2	b	A							
	2	b	B							

Fig. 13. Example of restriction

We are now in a position to consider various applications of these operations on relations.

2.2. Redundancy

Redundancy in the named set of relations must be distinguished from redundancy in the stored set of representations. We are primarily concerned here with the former. To begin with, we need a precise notion of derivability for relations.

Suppose θ is a collection of operations on relations and each operation has the property that from its operands it

المنارة للاستشارات

yields a unique relation (thus natural join is eligible, but join is not). A relation R is *θ-derivable* from a set S of relations if there exists a sequence of operations from the collection θ which, for all time, yields R from members of S. The phrase "for all time" is present, because we are dealing with time-varying relations, and our interest is in derivability which holds over a significant period of time. For the named set of relationships in noninferential systems, it appears that an adequate collection θ_1 contains the following operations: projection, natural join, tie, and restriction. Permutation is irrelevant and natural composition need not be included, because it is obtainable by taking a natural join and then a projection. For the stored set of representations, an adequate collection θ_2 of operations would include permutation and additional operations concerned with subsetting and merging relations, and ordering and connecting their elements.

2.2.1. *Strong Redundancy.* A set of relations is *strongly redundant* if it contains at least one relation that possesses a projection which is derivable from other projections of relations in the set. The following two examples are intended to explain why strong redundancy is defined this way, and to demonstrate its practical use. In the first example the collection of relations consists of just the following relation:

$$employee\ (serial\#, name, manager\#, managername)$$

with *serial#* as the primary key and *manager#* as a foreign key. Let us denote the active domain by Δ_t, and suppose that

$$\Delta_t(manager\#) \subset \Delta_t(serial\#)$$

and

$$\Delta_t(managername) \subset \Delta_t(name)$$

المنارة للاستشارات

for all time t. In this case the redundancy is obvious: the domain *managername* is unnecessary. To see that it is a strong redundancy as defined above, we observe that

$$\pi_{34}(employee) = \pi_{12}(employee)_1|_1\pi_3(employee).$$

In the second example the collection of relations includes a relation S describing suppliers with primary key $s\#$, a relation D describing departments with primary key $d\#$, a relation J describing projects with primary key $j\#$, and the following relations:

$$P(s\#, d\#, \cdots), \qquad Q(s\#, j\#, \cdots), \qquad R(d\#, j\#, \cdots),$$

where in each case $\cdots$ denotes domains other than $s\#$, $d\#$, $j\#$. Let us suppose the following condition C is known to hold independent of time: supplier s supplies department d (relation P) if and only if supplier s supplies some project j (relation Q) to which d is assigned (relation R). Then, we can write the equation

$$\pi_{12}(P) = \pi_{12}(Q) \cdot \pi_{21}(R)$$

and thereby exhibit a strong redundancy.

An important reason for the existence of strong redundancies in the named set of relationships is user convenience. A particular case of this is the retention of semi-obsolete relationships in the named set so that old programs that refer to them by name can continue to run correctly. Knowledge of the existence of strong redundancies in the named set enables a system or data base administrator greater freedom in the selection of stored representations to cope more efficiently with current traffic. If the strong redundancies in the named set are directly reflected in strong redundancies in the stored set (or if other strong redundancies are introduced into the stored set), then, generally speaking, extra storage space and update time are consumed with a potential drop in query time for some queries and in load on the central processing units.

المنارة للاستشارات

2.2.2. *Weak Redundancy.* A second type of redundancy may exist. In contrast to strong redundancy it is not characterized by an equation. A collection of relations is *weakly redundant* if it contains a relation that has a projection which is not derivable from other members but is at all times a projection of *some* join of other projections of relations in the collection.

We can exhibit a weak redundancy by taking the second example (cited above) for a strong redundancy, and assuming now that condition C does not hold at all times. The relations $\pi_{12}(P)$, $\pi_{12}(Q)$, $\pi_{12}(R)$ are complex[10] relations with the possibility of points of ambiguity occurring from time to time in the potential joining of any two. Under these circumstances, none of them is derivable from the other two. However, constraints do exist between them, since each is a projection of some cyclic join of the three of them. One of the weak redundancies can be characterized by the statement: for all time, $\pi_{12}(P)$ is *some* composition of $\pi_{12}(Q)$ with $\pi_{21}(R)$. The composition in question might be the natural one at some instant and a nonnatural one at another instant.

Generally speaking, weak redundancies are inherent in the logical needs of the community of users. They are not removable by the system or data base administrator. If they appear at all, they appear in both the named set and the stored set of representations.

2.3. Consistency

Whenever the named set of relations is redundant in either sense, we shall associate with that set a collection of statements which define all of the redundancies which hold independent of time between the member relations. If the information system lacks—and it most probably will—de-

[10] A binary relation is complex if neither it nor its converse is a function.

المنارة للاستشارات

tailed semantic information about each named relation, it cannot deduce the redundancies applicable to the named set. It might, over a period of time, make attempts to *in*duce the redundancies, but such attempts would be fallible.

Given a collection C of time-varying relations, an associated set Z of constraint statements and an instantaneous value V for C, we shall call the state (C, Z, V) *consistent* or *inconsistent* according as V does or does not satisfy Z. For example, given stored relations R, S, T together with the constraint statement "$\pi_{12}(T)$ is a composition of $\pi_{12}(R)$ with $\pi_{12}(S)$", we may check from time to time that the values stored for R, S, T satisfy this constraint. An algorithm for making this check would examine the first two columns of each of R, S, T (in whatever way they are represented in the system) and determine whether

(1) $\pi_1(T) = \pi_1(R)$,

(2) $\pi_2(T) = \pi_2(S)$,

(3) for every element pair (a, c) in the relation $\pi_{12}(T)$ there is an element b such that (a, b) is in $\pi_{12}(R)$ and (b, c) is in $\pi_{12}(S)$.

There are practical problems (which we shall not discuss here) in taking an instantaneous snapshot of a collection of relations, some of which may be very large and highly variable.

It is important to note that consistency as defined above is a property of the instantaneous state of a data bank, and is independent of how that state came about. Thus, in particular, there is no distinction made on the basis of whether a user generated an inconsistency due to an act of omission or an act of commission. Examination of a simple example will show the reasonableness of this (possibly unconventional) approach to consistency.

المنارة للاستشارات

Suppose the named set C includes the relations S, J, D, P, Q, R of the example in Section 2.2 and that P, Q, R possess either the strong or weak redundancies described therein (in the particular case now under consideration, it does not matter which kind of redundancy occurs). Further, suppose that at some time t the data bank state is consistent and contains no project j such that supplier 2 supplies project j and j is assigned to department 5. Accordingly, there is no element (2, 5) in $\pi_{12}(P)$. Now, a user introduces the element (2, 5) into $\pi_{12}(P)$ by inserting some appropriate element into P. The data bank state is now inconsistent. The inconsistency could have arisen from an act of omission, if the input (2, 5) is correct, and there does exist a project j such that supplier 2 supplies j and j is assigned to department 5. In this case, it is very likely that the user intends in the near future to insert elements into Q and R which will have the effect of introducing $(2, j)$ into $\pi_{12}(Q)$ and $(5, j)$ in $\pi_{12}(R)$. On the other hand, the input (2, 5) might have been faulty. It could be the case that the user intended to insert some other element into P—an element whose insertion would transform a consistent state into a consistent state. The point is that the system will normally have no way of resolving this question without interrogating its environment (perhaps the user who created the inconsistency).

There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs. Naturally, such checking will slow these operations down. If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch

المنارة للاستشارات

operation once a day or less frequently. Inputs causing the inconsistencies which remain in the data bank state at checking time can be tracked down if the system maintains a journal of all state-changing transactions. This latter approach would certainly be superior if few non-transitory inconsistencies occurred.

2.4. Summary

In Section 1 a relational model of data is proposed as a basis for protecting users of formatted data systems from the potentially disruptive changes in data representation caused by growth in the data bank and changes in traffic. A normal form for the time-varying collection of relationships is introduced.

In Section 2 operations on relations and two types of redundancy are defined and applied to the problem of maintaining the data in a consistent state. This is bound to become a serious practical problem as more and more different types of data are integrated together into common data banks.

Many questions are raised and left unanswered. For example, only a few of the more important properties of the data sublanguage in Section 1.4 are mentioned. Neither the purely linguistic details of such a language nor the implementation problems are discussed. Nevertheless, the material presented should be adequate for experienced systems programmers to visualize several approaches. It is also hoped that this paper can contribute to greater precision in work on formatted data systems.

Acknowledgment. It was C. T. Davies of IBM Poughkeepsie who convinced the author of the need for data independence in future information systems. The author wishes to thank him and also F. P. Palermo, C. P. Wang, E. B. Altman, and M. E. Senko of the IBM San Jose Research Laboratory for helpful discussions.

المنارة للاستشارات

Received September, 1969; revised February, 1970

REFERENCES

1. Childs, D. L. Feasibility of a set-theoretical data structure —a general structure based on a reconstituted definition of relation. Proc. IFIP Cong., 1968, North Holland Pub. Co., Amsterdam, p. 162–172.
2. Levein, R. E., and Maron, M. E. A computer system for inference execution and data retrieval. *Comm. ACM 10*, 11 (Nov. 1967), 715–721.
3. Bachman, C. W. Software for random access processing. *Datamation* (Apr. 1965), 36–41.
4. McGee, W. C. Generalized file processing. In *Annual Review in Automatic Programming 5*, 13, Pergamon Press, New York, 1969, pp. 77–149.
5. Information Management System/360, Application Description Manual H20-0524-1. IBM Corp., White Plains, N. Y., July 1968.
6. GIS (Generalized Information System), Application Description Manual H20-0574. IBM Corp., White Plains, N. Y., 1965.
7. Bleier, R. E. Treating hierarchical data structures in the SDC time-shared data management system (TDMS). Proc. ACM 22nd Nat. Conf., 1967, MDI Publications, Wayne, Pa., pp. 41–49.
8. IDS Reference Manual GE 625/635, GE Inform. Sys. Div., Pheonix, Ariz., CPB 1093B, Feb. 1968.
9. Church, A. *An Introduction to Mathematical Logic I*. Princeton U. Press, Princeton, N.J., 1956.
10. Feldman, J. A., and Rovner, P. D. An Algol-based associative language. Stanford Artificial Intelligence Rep. AI-66, Aug. 1, 1968.

المنارة للاستشارات

Barry Boehm

Software Engineering Economics

IEEE Transactions on Software Engineering, Vol. SE-10 (1), 1984
pp. 4–21

المنارة للاستشارات

Software Engineering Economics

BARRY W. BOEHM

Manuscript received April 26, 1983; revised June 28, 1983.

The author is with the Software Information Systems Division, TRW Defense Systems Group, Redondo Beach, CA 90278.

***Abstract*—This paper summarizes the current state of the art and recent trends in software engineering economics. It provides an overview of economic analysis techniques and their applicability to software engineering and management. It surveys the field of software cost estimation, including the major estimation techniques available, the state of the art in algorithmic cost models, and the outstanding research issues in software cost estimation.**

***Index Terms*—Computer programming costs, cost models, management decision aids, software cost estimation, software economics, software engineering, software management.**

I. INTRODUCTION

Definitions

The dictionary defines "economics" as "a social science concerned chiefly with description and analysis of the production, distribution, and consumption of goods and services." Here is another definition of economics which I think is more helpful in explaining how economics relates to software engineering.

> *Economics* is the study of how people make decisions in resource-limited situations.

This definition of economics fits the major branches of classical economics very well.

Macroeconomics is the study of how people make decisions in resource-limited situations on a national or global scale. It deals with the effects of decisions that national leaders make on such issues as tax rates, interest rates, foreign and trade policy.

المنارة للاستشارات

Microeconomics is the study of how people make decisions in resource-limited situations on a more personal scale. It deals with the decisions that individuals and organizations make on such issues as how much insurance to buy, which word processor to buy, or what prices to charge for their products or services.

Economics and Software Engineering Management

If we look at the discipline of software engineering, we see that the microeconomics branch of economics deals more with the types of decisions we need to make as software engineers or managers.

Clearly, we deal with limited resources. There is never enough time or money to cover all the good features we would like to put into our software products. And even in these days of cheap hardware and virtual memory, our more significant software products must always operate within a world of limited computer power and main memory. If you have been in the software engineering field for any length of time, I am sure you can think of a number of decision situations in which you had to determine some key software product feature as a function of some limiting critical resource.

Throughout the software life cycle,[1] there are many decision situations involving limited resources in which software engineering economics techniques provide useful assistance. To provide a feel for the nature of these economic decision issues, an example is given below for each of the major phases in the software life cycle.

- *Feasibility Phase:* How much should we invest in information system analyses (user questionnaires and in-

[1] Economic principles underlie the overall structure of the software life cycle, and its primary refinements of prototyping, incremental development, and advancemanship. The primary economic driver of the life-cycle structure is the significantly increasing cost of making a software change or fixing a software problem, as a function of the phase in which the change or fix is made. See [11, ch. 4].

المنارة للاستشارات

terviews, current-system analysis, workload characterizations, simulations, scenarios, prototypes) in order that we converge on an appropriate definition and concept of operation for the system we plan to implement?

- *Plans and Requirements Phase:* How rigorously should we specify requirements? How much should we invest in requirements validation activities (automated completeness, consistency, and traceability checks, analytic models, simulations, prototypes) before proceeding to design and develop a software system?
- *Product Design Phase:* Should we organize the software to make it possible to use a complex piece of existing software which generally but not completely meets our requirements?
- *Programming Phase:* Given a choice between three data storage and retrieval schemes which are primarily execution time-efficient, storage-efficient, and easy-to-modify, respectively; which of these should we choose to implement?
- *Integration and Test Phase:* How much testing and formal verification should we perform on a product before releasing it to users?
- *Maintenance Phase:* Given an extensive list of suggested product improvements, which ones should we implement first?
- *Phaseout:* Given an aging, hard-to-modify software product, should we replace it with a new product, restructure it, or leave it alone?

Outline of This Paper

The economics field has evolved a number of techniques (cost-benefit analysis, present value analysis, risk analysis, etc.) for dealing with decision issues such as the ones above. Section

المنارة للاستشارات

II of this paper provides an overview of these techniques and their applicability to software engineering.

One critical problem which underlies all applications of economic techniques to software engineering is the problem of estimating software costs. Section III contains three major sections which summarize this field:

III-A: Major Software Cost Estimation Techniques

III-B: Algorithmic Models for Software Cost Estimation

III-C: Outstanding Research Issues in Software Cost Estimation.

Section IV concludes by summarizing the major benefits of software engineering economics, and commenting on the major challenges awaiting the field.

II. Software Engineering Economics Analysis Techniques

Overview of Relevant Techniques

The microeconomics field provides a number of techniques for dealing with software life-cycle decision issues such as the ones given in the previous section. Fig. 1 presents an overall master key to these techniques and when to use them.[2]

As indicated in Fig. 1, standard optimization techniques can be used when we can find a single quantity such as dollars (or pounds, yen, cruzeiros, etc.) to serve as a "universal solvent" into which all of our decision variables can be converted. Or, if the nondollar objectives can be expressed as constraints (system availability must be at least 98 percent; throughput must be at least 150 transactions per second), then standard constrained optimization techniques can be used. And if cash flows occur at different times, then present-value techniques can be used to normalize them to a common point in time.

[2] The chapter numbers in Fig. 1 refer to the chapters in [11], in which those techniques are discussed in further detail.

المنارة للاستشارات

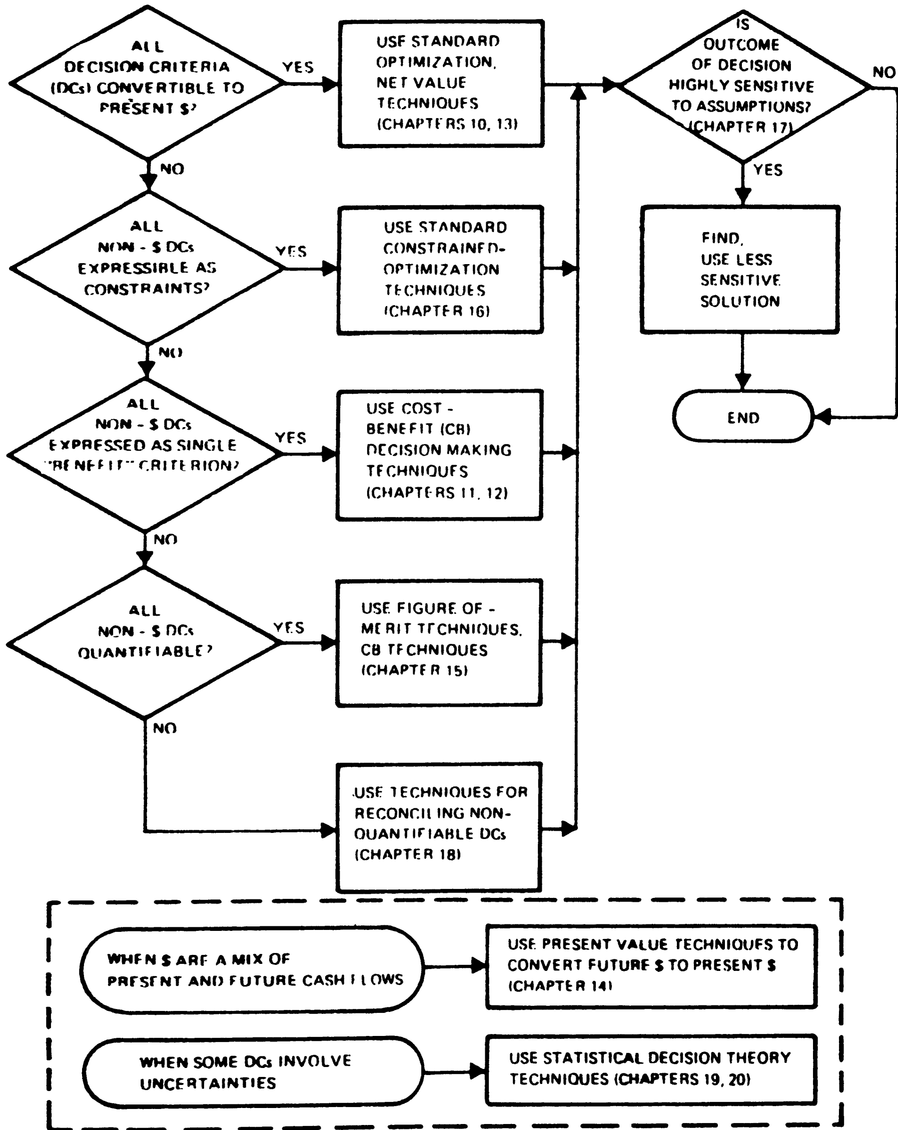


Fig. 1. Master key to software engineering economics decision analysis techniques.

المنارة للاستشارات

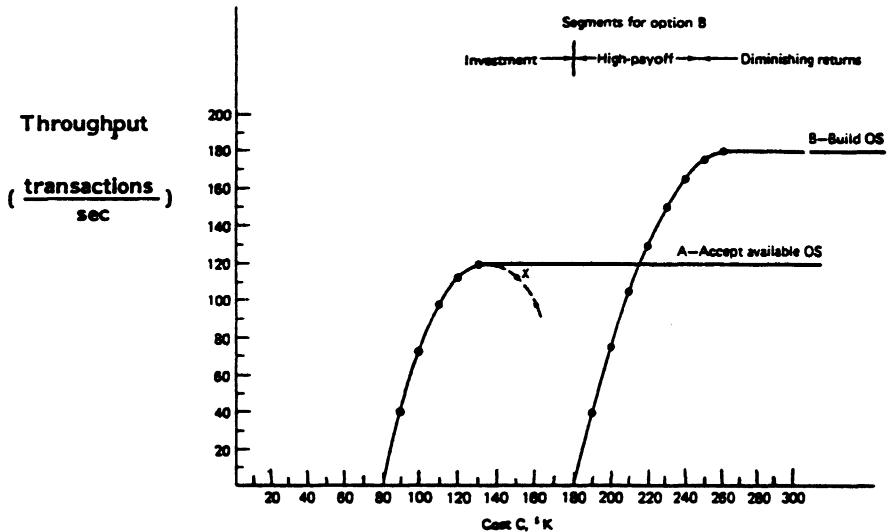


Fig. 2. Cost-effectiveness comparison, transaction processing system options.

More frequently, some of the resulting benefits from the software system are not expressible in dollars. In such situations, one alternative solution will not necessarily dominate another solution.

An example situation is shown in Fig. 2, which compares the cost and benefits (here, in terms of throughput in transactions per second) of two alternative approaches to developing an operating system for a transaction processing system.

- *Option A:* Accept an available operating system. This will require only $80K in software costs, but will achieve a peak performance of 120 transactions per second, using five $10K minicomputer processors, because of a high multiprocessor overhead factor.
- *Option B:* Build a new operating system. This system would be more efficient and would support a higher peak throughput, but would require $180K in software costs.

المنارة للاستشارات

The cost-versus-performance curve for these two options are shown in Fig. 2. Here, neither option dominates the other, and various cost-benefit decision-making techniques (maximum profit margin, cost/benefit ratio, return on investments, etc.) must be used to choose between Options A and B.

In general, software engineering decision problems are even more complex than Fig. 2, as Options A and B will have several important criteria on which they differ (e.g., robustness, ease of tuning, ease of change, functional capability). If these criteria are quantifiable, then some type of figure of merit can be defined to support a comparative analysis of the preferability of one option over another. If some of the criteria are unquantifiable (user goodwill, programmer morale, etc.), then some techniques for comparing unquantifiable criteria need to be used. As indicated in Fig. 1, techniques for each of these situations are available, and discussed in [11].

Analyzing Risk, Uncertainty, and the Value of Information

In software engineering, our decision issues are generally even more complex than those discussed above. This is because the outcome of many of our options cannot be determined in advance. For example, building an operating system with a significantly lower multiprocessor overhead may be achievable, but on the other hand, it may not. In such circumstances, we are faced with a problem of *decision making under uncertainty,* with a considerable *risk* of an undesired outcome.

The main economic analysis techniques available to support us in resolving such problems are the following.

1) Techniques for decision making under complete uncertainty, such as the maximax rule, the maximin rule, and the Laplace rule [38]. These techniques are generally inadequate for practical software engineering decisions.

المنارة للاستشارات

2) Expected-value techniques, in which we estimate the probabilities of occurrence of each outcome (successful or unsuccessful development of the new operating system) and complete the expected payoff of each option:

$$EV = \text{Prob(success)} * \text{Payoff(successful OS)}$$
$$+ \text{Prob(failure)} * \text{Payoff(unsuccessful OS)}.$$

These techniques are better than decision making under complete uncertainty, but they still involve a great deal of risk if the Prob(failure) is considerably higher than our estimate of it.

3) Techniques in which we reduce uncertainty by *buying information*. For example, *prototyping* is a way of buying information to reduce our uncertainty about the likely success or failure of a multiprocessor operating system; by developing a rapid prototype of its high-risk elements, we can get a clearer picture of our likelihood of successfully developing the full operating system.

In general, prototyping and other options for buying information[3] are most valuable aids for software engineering decisions. However, they always raise the following question: "how much information-buying is enough?"

In principle, this question can be answered via statistical decision theory techniques involving the use of Bayes' Law, which allows us to calculate the expected payoff from a software project as a function of our level of investment in a prototype or other information-buying option. (Some examples of the use of Bayes' Law to estimate the appropriate level of investment in a prototype are given in [11, ch. 20].)

In practice, the use of Bayes' Law involves the estimation of a number of conditional probabilities which are not easy to

[3] Other examples of options for buying information to support software engineering decisions include feasibility studies, user surveys, simulation, testing, and mathematical program verification techniques.

المنارة للاستشارات

estimate accurately. However, the Bayes' Law approach can be translated into a number of *value-of-information guidelines*, or conditions under which it makes good sense to decide on investing in more information before committing ourselves to a particular course of action.

Condition 1: There exist attractive alternatives whose payoff varies greatly, depending on some critical states of nature. If not, we can commit ourselves to one of the attractive alternatives with no risk of significant loss.

Condition 2: The critical states of nature have an appreciable probability of occurring. If not, we can again commit ourselves without major risk. For situations with extremely high variations in payoff, the appreciable probability level is lower than in situations with smaller variations in payoff.

Condition 3: The investigations have a high probability of accurately identifying the occurrence of the critical states of nature. If not, the investigations will not do much to reduce our risk of loss due to making the wrong decision.

Condition 4: The required cost and schedule of the investigations do not overly curtail their net value. It does us little good to obtain results which cost more than they can save us, or which arrive too late to help us make a decision.

Condition 5: There exist significant side benefits derived from performing the investigations. Again, we may be able to justify an investigation solely on the basis of its value in training, team-building, customer relations, or design validation.

Some Pitfalls Avoided by Using the Value-of-Information Approach

The guideline conditions provided by the value-of-information approach provide us with a perspective which helps us avoid some serious software engineering pitfalls. The pitfalls below are expressed in terms of some frequently expressed but faulty pieces of software engineering advice.

المنارة للاستشارات

Pitfall 1: Always use a simulation to investigate the feasibility of complex realtime software. Simulations are often extremely valuable in such situations. However, there have been a good many simulations developed which were largely an expensive waste of effort, frequently under conditions that would have been picked up by the guidelines above. Some have been relatively useless because, once they were built, nobody could tell whether a given set of inputs was realistic or not (picked up by Condition 3). Some have been taken so long to develop that they produced their first results the week after the proposal was sent out, or after the key design review was completed (picked up by Condition 4).

Pitfall 2: Always build the software twice. The guidelines indicate that the prototype (or build-it-twice) approach is often valuable, but not in all situations. Some prototypes have been built of software whose aspects were all straightforward and familiar, in which case nothing much was learned by building them (picked up by Conditions 1 and 2).

Pitfall 3: Build the software purely top-down. When interpreted too literally, the top-down approach does not concern itself with the design of low level modules until the higher levels have been fully developed. If an adverse state of nature makes such a low level module (automatically forecast sales volume, automatically discriminate one type of aircraft from another) impossible to develop, the subsequent redesign will generally require the expensive rework of much of the higher level design and code. Conditions 1 and 2 warn us to temper our top-down approach with a thorough top-to-bottom software risk analysis during the requirements and product design phases.

Pitfall 4: Every piece of code should be proved correct. Correctness proving is still an expensive way to get information on the fault-freedom of software, although it strongly satisfies Condition 3 by giving a very high assurance of a program's correctness. Conditions 1 and 2 recommend that proof

المنارة للاستشارات

techniques be used in situations where the operational cost of a software fault is very large, that is, loss of life, compromised national security, major financial losses. But if the operational cost of a software fault is small, the added information on fault-freedom provided by the proof will not be worth the investment (Condition 4).

Pitfall 5: Nominal-case testing is sufficient. This pitfall is just the opposite of Pitfall 4. If the operational cost of potential software faults is large, it is highly imprudent not to perform off-nominal testing.

Summary: The Economic Value of Information

Let us step back a bit from these guidelines and pitfalls. Put simply, we are saying that, as software engineers:

> "It is often worth paying for information because it helps us make better decisions."

If we look at the statement in a broader context, we can see that it is the primary reason why the software engineering field exists. It is what practically all of our software customers say when they decide to acquire one of our products: that it is worth paying for a management information system, a weather forecasting system, an air traffic control system, an inventory control system, etc., because it helps them make better decisions.

Usually, software engineers are *producers* of management information to be consumed by other people, but during the software life cycle we must also be *consumers* of management information to support our own decisions. As we come to appreciate the factors which make it attractive for us to pay for processed information which helps *us* make better decisions as software engineers, we will get a better appreciation for what our customers and users are looking for in the information processing systems we develop for *them.*

المنارة للاستشارات

III. Software Cost Estimation

Introduction

All of the software engineering economics decision analysis techniques discussed above are only as good as the input data we can provide for them. For software decisions, the most critical and difficult of these inputs to provide are estimates of the cost of a proposed software project. In this section, we will summarize:

1) the major software cost estimation techniques available, and their relative strengths and difficulties;

2) algorithmic models for software cost estimation;

3) outstanding research issues in software cost estimation.

A. Major Software Cost Estimation Techniques

Table I summarizes the relative strengths and difficulties of the major software cost estimation methods in use today.

1) *Algorithmic Models:* These methods provide one or more algorithms which produce a software cost estimate as a function of a number of variables which are considered to be the major cost drivers.

2) *Expert Judgment:* This method involves consulting one or more experts, perhaps with the aid of an expert-consensus mechanism such as the Delphi technique.

3) *Analogy:* This method involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project.

4) *Parkinson:* A Parkinson principle ("work expands to fill the available volume") is invoked to equate the cost estimate to the available resources.

5) *Price-to-Win:* Here, the cost estimate is equated to the price believed necessary to win the job (or the schedule believed necessary to be first in the market with a new product, etc.).

المنارة للاستشارات

6) *Top-Down:* An overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.

7) *Bottom-Up:* Each component of the software job is separately estimated, and the results aggregated to produce an estimate for the overall job.

The main conclusions that we can draw from Table I are the following.

- None of the alternatives is better than the others from all aspects.
- The Parkinson and price-to-win methods are unacceptable and do not produce satisfactory cost estimates.
- The strengths and weaknesses of the other techniques are complementary (particularly the algorithmic models versus expert judgment and top-down versus bottom-up).
- Thus, in practice, we should use combinations of the above techniques, compare their results, and iterate on them where they differ.

TABLE I
STRENGTHS AND WEAKNESSES OF SOFTWARE COST-ESTIMATION METHODS

Method	Strengths	Weaknesses
Algorithmic model	• Objective, repeatable, analyzable formula • Efficient, good for sensitivity analysis • Objectively calibrated to experience	• Subjective inputs • Assessment of exceptional circumstances • Calibrated to past, not future
Expert judgment	• Assessment of representativeness, interactions, exceptional circumstances	• No better than participants • Biases, incomplete recall
Analogy	• Based on representative experience	• Representativeness of experience
Parkinson	• Correlates with some experience	• Reinforces poor practice
Price to win	• Often gets the contract	• Generally produces large overruns
Top-down	• System level focus • Efficient	• Less detailed basis • Less stable
Bottom-up	• More detailed basis • More stable • Fosters individual commitment	• May overlook system level costs • Requires more effort

المنارة للاستشارات

Fundamental Limitations of Software Cost Estimation Techniques

Whatever the strengths of a software cost estimation technique, there is really no way we can expect the technique to compensate for our lack of definition or understanding of the software job to be done. Until a software specification is fully defined, it actually represents a range of software products, and a corresponding range of software development costs.

This fundamental limitation of software cost estimation technology is illustrated in Fig. 3, which shows the accuracy within which software cost estimates can be made, as a function of the software life-cycle phase (the horizontal axis), or of the level of knowledge we have of what the software is intended to do. This level of uncertainty is illustrated in Fig. 3 with respect to a human-machine interface component of the software.

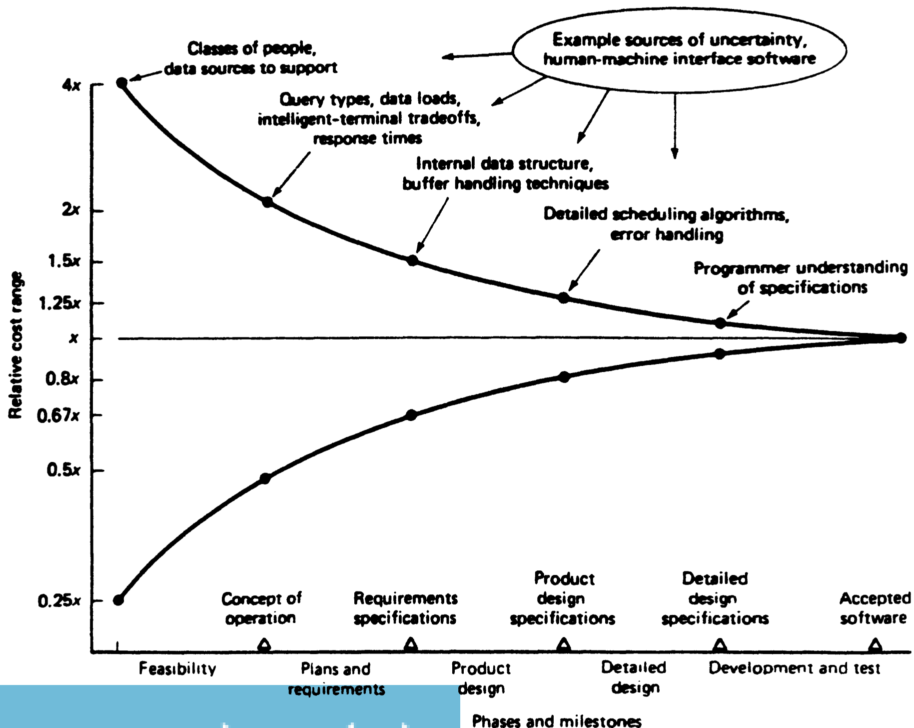


Fig. 3. Software cost estimation accuracy versus phase.

المنارة للاستشارات

When we first begin to evaluate alternative concepts for a new software application, the relative range of our software cost estimates is roughly a factor of four on either the high or low side.[4] This range stems from the wide range of uncertainty we have at this time about the actual nature of the product. For the human-machine interface component, for example, we do not know at this time what classes of people (clerks, computer specialists, middle managers, etc.) or what classes of data (raw or pre-edited, numerical or text, digital or analog) the system will have to support. Until we pin down such uncertainties, a factor of four in either direction is not surprising as a range of estimates.

The above uncertainties are indeed pinned down once we complete the feasibility phase and settle on a particular concept of operation. At this stage, the range of our estimates diminishes to a factor of two in either direction. This range is reasonable because we still have not pinned down such issues as the specific types of user query to be supported, or the specific functions to be performed within the microprocessor in the intelligent terminal. These issues will be resolved by the time we have developed a software requirements specification, at which point, we will be able to estimate the software costs within a factor of 1.5 in either direction.

By the time we complete and validate a product design specification, we will have resolved such issues as the internal data structure of the software product and the specific techniques for handling the buffers between the terminal microprocessor and the central processors on one side, and between the microprocessor and the display driver on the other. At this point, our software estimate should be accurate to within a factor of 1.25, the discrepancies being caused by some remaining sources of uncertainty such as the specific algorithms to be

[4] These ranges have been determined subjectively, and are intended to represent 80 percent confidence limits, that is, "within a factor of four on either side, 80 percent of the time."

المنارة للاستشارات

used for task scheduling, error handling, abort processing, and the like. These will be resolved by the end of the detailed design phase, but there will still be a residual uncertainty about 10 percent based on how well the programmers really understand the specifications to which they are to code. (This factor also includes such consideration as personnel turnover uncertainties during the development and test phases.)

B. Algorithmic Models for Software Cost Estimation

Algorithmic Cost Models: Early Development

Since the earliest days of the software field, people have been trying to develop algorithmic models to estimate software costs. The earliest attempts were simple rules of thumb, such as:

- on a large project, each software performer will provide an average of one checked-out instruction per man-hour (or roughly 150 instructions per man-month);
- each software maintenance person can maintain four boxes of cards (a box of cards held 2000 cards, or roughly 2000 instructions in those days of few comment cards).

Somewhat later, some projects began collecting quantitative data on the effort involved in developing a software product, and its distribution across the software life cycle. One of the earliest of these analyses was documented in 1956 in [8]. It indicated that, for very large operational software products on the order of 100 000 delivered source instructions (100 KDSI), that the overall productivity was more like 64 DSI/man-month, that another 100 KDSI of support-software would be required; that about 15 000 pages of documentation would be produced and 3000 hours of computer time consumed; and that the distribution of effort would be as follows:

Program Specs:	10 percent
Coding Specs:	30 percent

المنارة للاستشارات

Coding:	10 percent
Parameter Testing:	20 percent
Assembly Testing:	30 percent

with an additional 30 percent required to produce operational specs for the system. Unfortunately, such data did not become well known, and many subsequent software projects went through a painful process of rediscovering them.

During the late 1950's and early 1960's, relatively little progress was made in software cost estimation, while the frequency and magnitude of software cost overruns was becoming critical to many large systems employing computers. In 1964, the U.S. Air Force contracted with System Development Corporation for a landmark project in the software cost estimation field. This project collected 104 attributes of 169 software projects and treated them to extensive statistical analysis. One result was the 1965 SDC cost model [41] which was the best possible statistical 13-parameter linear estimation model for the sample data:

MM = −33.63

+9.15 (Lack of Requirements) (0-2)

+10.73 (Stability of Design) (0-3)

+0.51 (Percent Math Instructions)

+0.46 (Percent Storage/Retrieval Instructions)

+0.40 (Number of Subprograms)

+7.28 (Programming Language) (0-1)

−21.45 (Business Application) (0-1)

+13.53 (Stand-Alone Program) (0.1)

+12.35 (First Program on Computer) (0-1)

+58.82 (Concurrent Hardware Development) (0-1)

+30.61 (Random Access Device Used) (0-1)

المنارة للاستشارات

+29.55 (Difference Host, Target Hardware) (0-1)

+0.54 (Number of Personnel Trips)

−25.20 (Developed by Military Organization) (0-1).

The numbers in parentheses refer to ratings to be made by the estimator.

When applied to its database of 169 projects, this model produced a mean estimate of 40 MM and a standard deviation of 62 MM; not a very accurate predictor. Further, the application of the model is counterintuitive; a project with all zero ratings is estimated at minus 33 MM; changing language from a higher order language to assembly language adds 7 MM, independent of project size. The most conclusive result from the SDC study was that there were too many nonlinear aspects of software development for a linear cost-estimation model to work very well.

Still, the SDC effort provided a valuable base of information and insight for cost estimation and future models. Its cumulative distribution of productivity for 169 projects was a valuable aid for producing or checking cost estimates. The estimation rules of thumb for various phases and activities have been very helpful, and the data have been a major foundation for some subsequent cost models.

In the late 1960's and early 1970's, a number of cost models were developed which worked reasonably well for a certain restricted range of projects to which they were calibrated. Some of the more notable examples of such models are those described in [3], [54], [57].

The essence of the TRW Wolverton model [57] is shown in Fig. 4, which shows a number of curves of software cost per object instruction as a function of relative degree of difficulty (0 to 100), novelty of the application (new or old), and type of project. The best use of the model involves breaking the software into components and estimating their cost individu-

المنارة للاستشارات

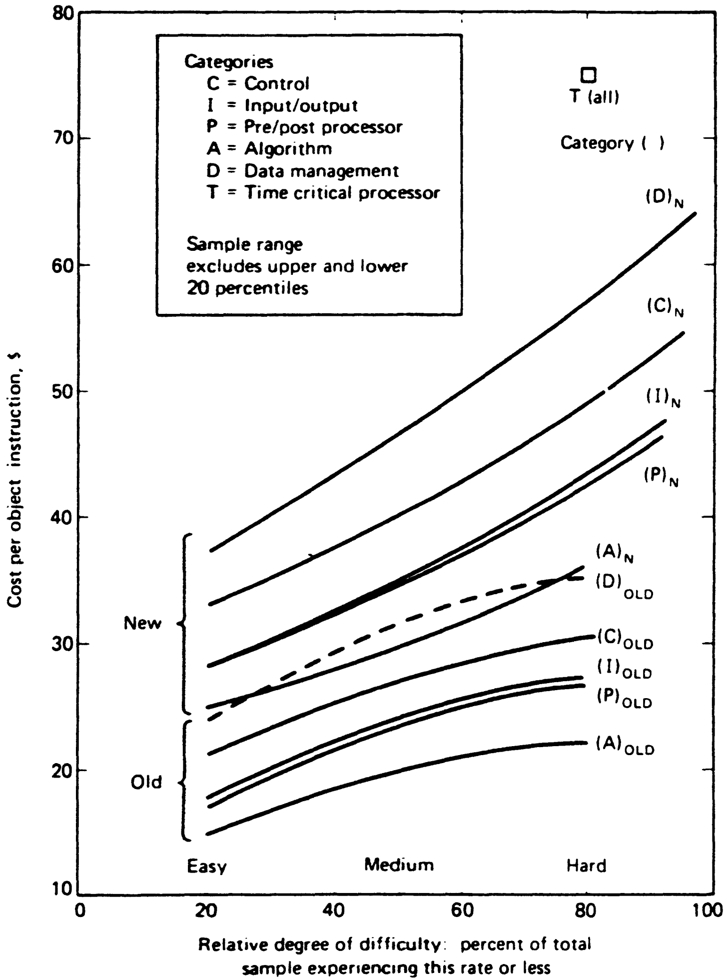


Fig. 4. TRW Wolverton model: Cost per object instruction versus relative degree of difficulty.

ally. This, a 1000 object-instruction module of new data management software of medium (50 percent) difficulty would be costed at $46/instruction, or $46 000.

This model is well-calibrated to a class of near-real-time government command and control projects, but is less accurate for some other classes of projects. In addition, the model provides a good breakdown of project effort by phase and activity.

المنارة للاستشارات

In the late 1970's, several software cost estimation models were developed which established a significant advance in the state of the art. These included the Putnam SLIM Model [44], the Doty Model [27], the RCA PRICE S model [22], the COCOMO model [11], the IBM-FSD model [53], the Boeing model [9], and a series of models developed by GRC [15]. A summary of these models, and the earlier SDC and Wolverton models, is shown in Table II, in terms of the size, program, computer, personnel, and project attributes used by each model to determine software costs. The first four of these models are discussed below.

The Putnam SLIM Model [44], [45]

The Putnam SLIM Model is a commercially available (from Quantitative Software Management, Inc.) software product based on Putnam's analysis of the software life cycle in terms of the Rayleigh distribution of project personnel level versus time. The basic effort macro-estimation model used in SLIM is

$$S_s = C_k K^{1/3} t_d^{4/3}$$

where

S_s = number of delivered source instructions
K = life-cycle effort in man-years
t_d = development time in years
C_k = a "technology constant."

Values of C_k typically range between 610 and 57 314. The current version of SLIM allows one to calibrate C_k to past projects or to past projects or to estimate it as a function of a project's use of modern programming practices, hardware constraints, personnel experience, interactive development, and other factors. The required development effort, DE, is estimated as roughly 40 percent of the life-cycle effort for large

للاستشارات المنارة

TABLE II
FACTORS USED IN VARIOUS COST MODELS

GROUP	FACTOR	SDC, 1965	TRW, 1972	PUTNAM, SLIM	DOTY	RCA, PRICE S	IBM	BOEING, 1977	GRC, 1979	COCOMO	SOFCOST	DSN	JENSEN
SIZE ATTRIBUTES	SOURCE INSTRUCTIONS			X	X		X	X		X	X	X	X
	OBJECT INSTRUCTIONS	X	X		X	X							
	NUMBER OF ROUTINES	X				X					X		
	NUMBER OF DATA ITEMS						X			X	X		
	NUMBER OF OUTPUT FORMATS								X			X	
	DOCUMENTATION				X		X				X		X
	NUMBER OF PERSONNEL			X			X	X			X		X
PROGRAM ATTRIBUTES	TYPE	X	X	X	X	X	X	X			X		
	COMPLEXITY		X	X		X	X			X	X	X	X
	LANGUAGE	X		X				X	X		X	X	
	REUSE			X		X		X	X	X	X	X	X
	REQUIRED RELIABILITY			X		X				X	X		X
	DISPLAY REQUIREMENTS				X						X		X
COMPUTER ATTRIBUTES	TIME CONSTRAINT		X	X	X	X }	X	X	X	X	X	X	X
	STORAGE CONSTRAINT			X	X	X }	X		X	X	X	X	X
	HARDWARE CONFIGURATION	X				X							
	CONCURRENT HARDWARE DEVELOPMENT	X			X	X	X			X	X	X	X
	INTERFACING EQUIPMENT, S/W										X	X	
PERSONNEL ATTRIBUTES	PERSONNEL CAPABILITY			X		X	X			X	X	X	X
	PERSONNEL CONTINUITY						X					X	
	HARDWARE EXPERIENCE	X		X	X	X	X		X }	X	X	X	X
	APPLICATIONS EXPERIENCE		X	X		X	X	X	X }	X	X	X	X
	LANGUAGE EXPERIENCE			X		X	X		X }	X	X	X	X
PROJECT ATTRIBUTES	TOOLS AND TECHNIQUES			X		X	X	X		X	X	X	X
	CUSTOMER INTERFACE	X					X				X	X	
	REQUIREMENTS DEFINITION	X			X		X				X	X	X
	REQUIREMENTS VOLATILITY	X			X	X	X		X	X	X	X	X
	SCHEDULE			X		X				X	X	X	X
	SECURITY						X				X	X	
	COMPUTER ACCESS			X	X		X	X		X	X	X	X
	TRAVEL/REHOSTING/MULTI-SITE	X			X	X					X	X	X
	SUPPORT SOFTWARE MATURITY									X		X	
CALIBRATION FACTOR				X		X				X			
EFFORT EQUATION	$MM_{NOM} = C(DSI)^X$, X =		1.0		1.047		0.91	1.0		1.05 - 1.2		1.0	1.2
SCHEDULE EQUATION	$t_D = C\,(MM)^X$, X =						0.35			0.32 - 0.38		0.356	0.333

systems. For smaller systems, the percentage varies as a function of system size.

The SLIM model includes a number of useful extensions to estimate such quantities as manpower distribution, cash flow, major-milestone schedules, reliability levels, computer time, and documentation costs.

The most controversial aspect of the SLIM model is its tradeoff relationship between development effort K and between development time t_d. For a software product of a given size, the SLIM software equation above gives

$$K = \frac{\text{constant}}{t_d^4}.$$

For example, this relationship says that one can cut the cost of a software project in half, simply by increasing its development time by 19 percent (e.g., from 10 months to 12 months). Fig. 5 shows how the SLIM tradeoff relationship compares with those of other models; see [11, ch. 27] for further discussion of this issue.

On balance, the SLIM approach has provided a number of useful insights into software cost estimation, such as the Rayleigh-curve distribution for one-shot software efforts, the explicit treatment of estimation risk and uncertainty, and the cube-root relationship defining the minimum development time achievable for a project requiring a given amount of effort.

The Doty Model [27]

This model is the result of an extensive data analysis activity, including many of the data points from the SDC sample. A number of models of similar form were developed for different application areas. As an example, the model for general application is

المنارة للاستشارات

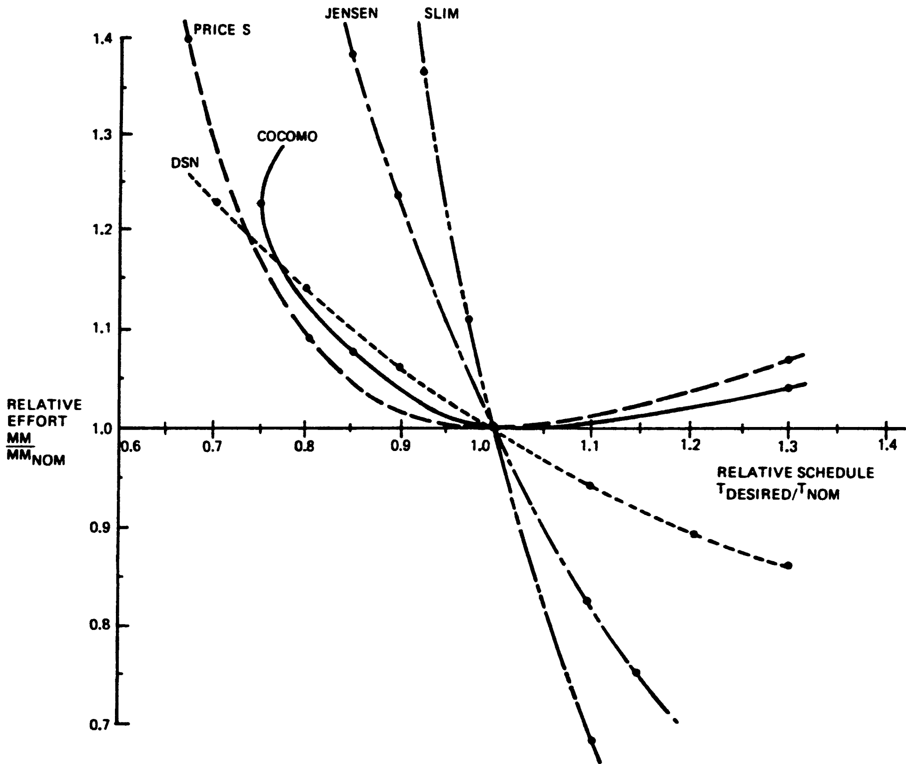


Fig. 5. Comparative effort-schedule tradeoff relationships.

$$MM = 5.288\,(KDSI)^{1.047}, \qquad \text{for KDSI} \geqslant 10$$

$$MM = 2.060\,(KDSI)^{1.047}\left(\prod_{j=1}^{14} f_j\right), \qquad \text{for KDSI} < 10.$$

The effort multipliers f_i are shown in Table III. This model has a much more appropriate functional form than the SDC model, but it has some problems with stability, as it exhibits a discontinuity at KDSI = 10, and produces widely varying estimates via the f factors (answering "yes" to "first software developed on CPU" adds 92 percent to the estimated cost).

The RCA PRICE S Model [22]

PRICE S is a commercially available (from RCA, Inc.) macro cost-estimation model developed primarily for embed-

المنارة للاستشارات

TABLE III
DOTY MODEL FOR SMALL PROGRAMS*

$$MM = 2.060\, I^{1.047} \prod_{j=1}^{j=14} f_j$$

Factor	f_j	Yes	No
Special display	f_1	1.11	1.00
Detailed definition of operational requirements	f_2	1.00	1.11
Change to operational requirements	f_3	1.05	1.00
Real-time operation	f_4	1.33	1.00
CPU memory constraint	f_5	1.43	1.00
CPU time constraint	f_6	1.33	1.00
First software developed on CPU	f_7	1.92	1.00
Concurrent development of ADP hardware	f_8	1.82	1.00
Timeshare versus batch processing, in development	f_9	0.83	1.00
Developer using computer at another facility	f_{10}	1.43	1.00
Development at operational site	f_{11}	1.39	1.00
Development computer different than target computer	f_{12}	1.25	1.00
Development at more than one site	f_{13}	1.25	1.00
Programmer access to computer	f_{14}	Limited	1.00
		Unlimited	0.90

* Less than 10,000 source instructions

ded system applications. It has improved steadily with experience; earlier versions with a widely varying subjective complexity factor have been replaced by versions in which a number of computer, personnel, and project attributes are used to modulate the complexity rating.

PRICE S has extended a number of cost-estimating relationships developed in the early 1970's such as the hardware constraint function shown in Fig. 6 [10]. It was primarily developed to handle military software projects, but now also includes rating levels to cover business applications.

PRICE S also provides a wide range of useful outputs on gross phase and activity distributions analyses, and monthly project cost-schedule-expected progress forecasts. Price S uses a two-parameter beta distribution rather than a Rayleigh curve to calculate development effort distribution versus calendar time.

PRICE S has recently added a software life-cycle support cost estimation capability called PRICE SL [34]. It involves the definition of three categories of support activities.

المنارة للاستشارات

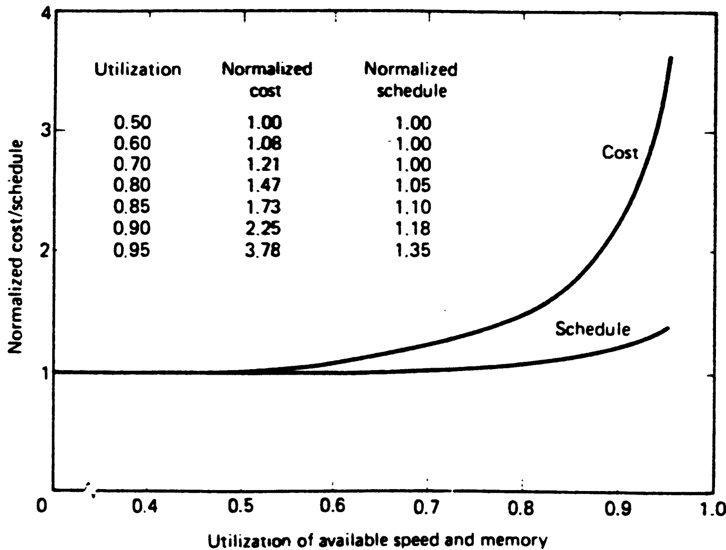


Fig. 6. RCA PRICE S model: Effect of hardware constraints.

- *Growth:* The estimator specifies the amount of code to be added to the product. PRICE SL then uses its standard techniques to estimate the resulting life-cycle-effort distribution.
- *Enhancement:* PRICE SL estimates the fraction of the existing product which will be modified (the estimator may provide his own fraction), and uses its standard techniques to estimate the resulting life-cycle effort distribution.
- *Maintenance:* The estimator provides a parameter indicating the quality level of the developed code. PRICE SL uses this to estimate the effort required to eliminate remaining errors.

The COnstructive COst MOdel (COCOMO) [11]

The primary motivation for the COCOMO model has been to help people understand the cost consequences of the decisions they will make in commissioning, developing, and supporting a software product. Besides providing a software cost estimation capability, COCOMO therefore provides a great

المنارة للاستشارات

deal of material which explains exactly what costs the model is estimating, and why it comes up with the estimates it does. Further, it provides capabilities for sensitivity analysis and tradeoff analysis of many of the common software engineering decision issues.

COCOMO is actually a hierarchy of three increasingly detailed models which range from a single macro-estimation scaling model as a function of product size to a micro-estimation model with a three-level work breakdown structure and a set of phase-sensitive multipliers for each cost driver attribute. To provide a reasonably concise example of a current state of the art cost estimation model, the intermediate level of COCOMO is described below.

Intermediate COCOMO estimates the cost of a proposed software product in the following way.

1) A nominal development effort is estimated as a function of the product's size in delivered source instructions in thousands (KDSI) and the project's development mode.

2) A set of effort multipliers are determined from the product's ratings on a set of 15 cost driver attributes.

3) The estimated development effort is obtained by multiplying the nominal effort estimate by all of the product's effort multipliers.

4) Additional factors can be used to determine dollar costs, development schedules, phase and activity distributions, computer costs, annual maintenance costs, and other elements from the development effort estimate.

Step 1–Nominal Effort Estimation: First, Table IV is used to determine the project's development mode. Organic-mode projects typically come from stable, familiar, forgiving, relatively unconstrained environments, and were found in the COCOMO data analysis of 63 projects have a different scaling equation from the more ambitious, unfamiliar, unforgiving, tightly constrained embedded mode. The resulting scaling equations for each mode are given in Table V; these are used

المنارة للاستشارات

to determine the nominal development effort for the project in man-months as a function of the project's size in KDSI and the project's development mode.

For example, suppose we are estimating the cost to develop the microprocessor-based communications processing software for a highly ambitious new electronic funds transfer network with high reliability, performance, development schedule, and interface requirements. From Table IV, we determine that these characteristics best fit the profile of an embedded-mode project.

We next estimate the size of the product as 10 000 delivered source instructions, or 10 KDSI. From Table V, we then determine that the nominal development effort for this Embedded-mode project is

TABLE IV
COCOMO SOFTWARE DEVELOPMENT MODES

Feature	Mode		
	Organic	Semidetached	Embedded
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300 KDSI	All sizes
Examples	Batch data reduction Scientific models Business models Familiar OS, compiler Simple inventory, production control	Most transaction processing systems New OS, DBMS Ambitious inventory, production control Simple command-control	Large, complex transaction processing systems Ambitious, very large OS Avionics Ambitious command-control

المنارة للاستشارات

TABLE V
COCOMO NOMINAL EFFORT AND SCHEDULE EQUATIONS

DEVELOPMENT MODE	NOMINAL EFFORT	SCHEDULE
Organic	$(MM)_{NOM} = 3.2(KDSI)^{1.05}$	$TDEV = 2.5(MM_{DEV})^{0.38}$
Semidetached	$(MM)_{NOM} = 3.0(KDSI)^{1.12}$	$TDEV = 2.5(MM_{DEV})^{0.35}$
Embedded	$(MM)_{NOM} = 2.8(KDSI)^{1.20}$	$TDEV = 2.5(MM_{DEV})^{0.32}$

(KDSI = thousands of delivered source instructions)

$$2.8(10)^{1.20} = 44 \text{ man-months (MM)}.$$

Step 2–Determine Effort Multipliers: Each of the 15 cost driver attributes in COCOMO has a rating scale and a set of effort multipliers which indicate by how much the nominal effort estimate must be multiplied to account for the project's having to work at its rating level for the attribute.

These cost driver attributes and their corresponding effort multipliers are shown in Table VI. The summary rating scales for each cost driver attribute are shown in Table VII, except for the complexity rating scale which is shown in Table VIII (expanded rating scales for the other attributes are provided in [11]).

The results of applying these tables to our microprocessor communications software example are shown in Table IX. The effect of a software fault in the electronic fund transfer system could be a serious financial loss; therefore, the project's RELY rating from Table VII is High. Then, from Table VI, the effort multiplier for achieving a High level of required reliability is 1.15, or 15 percent more effort than it would take to develop the software to a nominal level of required reliability.

المنارة للاستشارات

TABLE VI
INTERMEDIATE COCOMO SOFTWARE DEVELOPMENT EFFORT MULTIPLIERS

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience*	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

* For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks

The effort multipliers for the other cost driver attributes are obtained similarly, except for the Complexity attribute, which is obtained via Table VIII. Here, we first determine that communications processing is best classified under device-dependent operations (column 3 in Table VIII). From this column, we determine that communication line handling typically has a complexity rating of Very High; from Table VI, then, we determine that its corresponding effort multiplier is 1.30.

Step 3—Estimate Development Effort: We then compute the estimated development effort for the microprocessor communications software as the nominal development effort (44 MM) times the product of the effort multipliers for the 15 cost

المنارة للاستشارات

TABLE VII
COCOMO SOFTWARE COST DRIVER RATINGS

Cost Driver	Ratings: Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{\text{DB bytes}}{\text{Prog. DSI}} < 10$	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} \geq 1000$	
CPLX	See Table 8 →	→	→	→	→	→
Computer attributes						
TIME			≤ 50% use of available execution time	70%	85%	95%
STOR			≤ 50% use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround <4 hours	4–12 hours	>12 hours	
Personnel attributes						
ACAP	15th percentile[a]	35th percentile	55th percentile	75th percentile	90th percentile	
AEXP	≤4 months experience	1 year	3 years	6 years	12 years	
PCAP	15th percentile[a]	35th percentile	55th percentile	75th percentile	90th percentile	
VEXP	≤1 month experience	4 months	1 year	3 years		
LEXP	≤1 month experience	4 months	1 year	3 years		
Project attributes						
MODP	No use	Beginning use	Some use	General use	Routine use	
TOOL	Basic microprocessor tools	Basic mini tools	Basic midi/maxi tools	Strong maxi programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

[a] Team rating criteria: analysis (programming) ability, efficiency, ability to communicate and cooperate

المنارة للاستشارات

TABLE VIII
COCOMO MODULE COMPLEXITY RATINGS VERSUS TYPE OF MODULE

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
Very low	Straightline code with a few nonnested SP[a] operators: DOs, CASEs, IFTHENELSEs. Simple predicates	Evaluation of simple expressions: e.g., A = B + C * (D − E)	Simple read, write statements with simple formats	Simple arrays in main memory
Low	Straightforward nesting of SP operators. Mostly simple predicates	Evaluation of moderate-level expressions, e.g., D = SQRT (B**2−4.*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap	Single file subsetting with no data structure changes, no edits, no intermediate files
Nominal	Mostly simple nesting. Some intermodule control. Decision tables	Use of standard math and statistical routines. Basic matrix/vector operations	I/O processing includes device selection, status checking and error processing	Multi-file input and single file output. Simple structural changes, simple edits
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable intermodule control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns	Operations at physical I/O level (physical storage address translations; seeks, reads, etc). Optimized I/O overlap	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling	Difficult but structured N.A.: near-singular matrix equations, partial differential equations	Routines for interrupt diagnosis, servicing, masking. Communication line handling	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control	Difficult and unstructured N.A.: highly accurate analysis of noisy, stochastic data	Device timing-dependent coding, micro-programmed operations	Highly coupled, dynamic relational structures. Natural language data management

[a] SP = structured programming

المنارة للاستشارات

TABLE IX
COCOMO COST DRIVER RATINGS: MICROPROCESSOR COMMUNICATIONS SOFTWARE

Cost Driver	Situation	Rating	Effort Multiplier
RELY	Serious financial consequences of software faults	High	1.15
DATA	20,000 bytes	Low	0.94
CPLX	Communications processing	Very High	1.30
TIME	Will use 70% of available time	High	1.11
STOR	45K of 64K store (70%)	High	1.06
VIRT	Based on commercial microprocessor hardware	Nominal	1.00
TURN	Two hour average turnaround time	Nominal	1.00
ACAP	Good senior analysts	High	0.86
AEXP	Three years	Nominal	1.00
PCAP	Good senior programmers	High	0.86
VEXP	Six months	Low	1.10
LEXP	Twelve months	Nominal	1.00
MODP	Most techniques in use over one year	High	0.91
TOOL	At basic minicomputer tool level	Low	1.10
SCED	Nine months	Nominal	1.00
	Effort adjustment factor (product of effort multipliers)		1.35

المنارة للاستشارات

driver attributes in Table IX (1.35, in Table IX). The resulting estimated effort for the project is then

$$(44 \text{ MM})(1.35) = 59 \text{ MM}.$$

Step 4–Estimate Related Project Factors: COCOMO has additional cost estimating relationships for computing the resulting dollar cost of the project and for the breakdown of cost and effort by life-cycle phase (requirements, design, etc.) and by type of project activity (programming, test planning, management, etc.). Further relationships support the estimation of the project's schedule and its phase distribution. For example, the recommended development schedule can be obtained from the estimated development man-months via the embedded-mode schedule equation in Table V:

$$T_{\text{DEV}} = 2.5(59)^{0.32} = 9 \text{ months}.$$

As mentioned above, COCOMO also supports the most common types of sensitivity analysis and tradeoff analysis involved in scoping a software project. For example, from Tables VI and VII, we can see that providing the software developers with an interactive computer access capability (Low turnaround time) reduces the TURN effort multiplier from 1.00 to 0.87, and thus reduces the estimated project effort from 59 MM to

$$(59 \text{ MM})(0.87) = 51 \text{ MM}.$$

The COCOMO model has been validated with respect to a sample of 63 projects representing a wide variety of business, scientific, systems, real-time, and support software projects. For this sample, Intermediate COCOMO estimates come within 20 percent of the actuals about 68 percent of the time (see Fig. 7). Since the residuals roughly follow a normal distribution, this is equivalent to a standard deviation of roughly 20 percent of the project actuals. This level of accuracy is representative of the current state of the art in software cost models. One can do somewhat better with the aid

المنارة للاستشارات

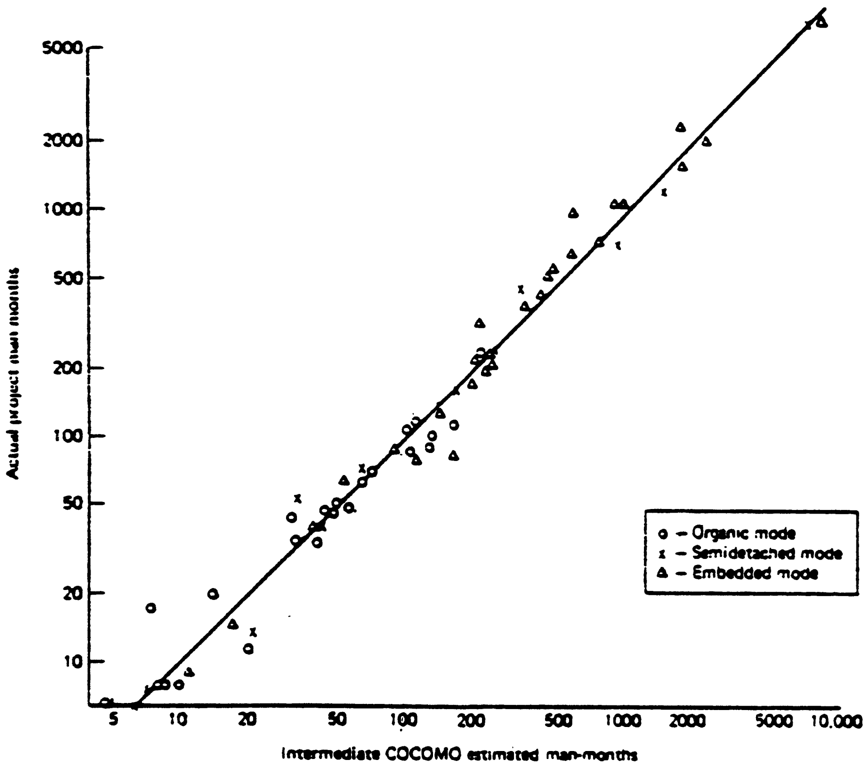


Fig. 7. Intermediate COCOMO estimates versus project actuals.

of a calibration coefficient (also a COCOMO option), or within a limited applications context, but it is difficult to improve significantly on this level of accuracy while the accuracy of software data collection remains in the "±20 percent" range.

A Pascal version of COCOMO is available for a nominal distribution charge from the Wang Institute, under the name WICOMO [18].

Recent Software Cost Estimation Models

Most of the recent software cost estimation models tend to follow the Doty and COCOMO models in having a nominal scaling equation of the form $MM_{NOM} = c(KDSI)^x$ and a set of multiplicative effort adjustment factors determined by a number of cost driver attribute ratings. Some of them use the Rayleigh curve approach to estimate distribution across the

المنارة للاستشارات

software life-cycle, but most use a more conservative effort/schedule tradeoff relation than the SLIM model. These aspects have been summarized for the various models in Table II and Fig. 5.

The Bailey-Basili meta-model [4] derived the scaling equation

$$MM_{NOM} = 3.5 + 0.73\ (KDSI)^{1.16}$$

and used two additional cost driver attributes (methodology level and complexity) to model the development effort of 18 projects in the NASA-Goddard Software Engineering Laboratory to within a standard deviation of 15 percent. Its accuracy for other project situations has not been determined.

The Grumman SOFCOST Model [19] uses a similar but unpublished nominal effort scaling equation, modified by 30 multiplicative cost driver variables rated on a scale of 0 to 10. Table II includes a summary of these variables.

The Tausworthe Deep Space Network (DSN) model [50] uses a linear scaling equation ($MM_{NOM} = a(KDSI)^{1.0}$) and a similar set of cost driver attributes, also summarized in Table II. It also has a well-considered approach for determining the equivalent KDSI involved in adapting existing software within a new product. It uses the Rayleigh curve to determine the phase distribution of effort, but uses a considerably more conservative version of the SLIM effort-schedule tradeoff relationship (see Fig. 5).

The Jensen model [30], [31] is a commercially available model with a similar nominal scaling equation, and a set of cost driver attributes very similar to the Doty and COCOMO models (but with different effort multiplier ranges); see Table II. Some of the multiplier ranges in the Jensen model vary as functions of other factors; e.g., increasing access to computer resources widens the multiplier ranges on such cost drivers as personnel capability and use of software tools. It uses the Rayleigh curve for effort distribution, and a somewhat more conservative ef-

المنارة للاستشارات

fort-schedule tradeoff relation than SLIM (see Fig. 5). As with the other commercial models, the Jensen model produces a number of useful outputs on resource expenditure rates, probability distributions on costs and schedules, etc.

C. Outstanding Research Issues in Software Cost Estimation

Although a good deal of progress has been made in software cost estimation, a great deal remains to be done. This section updates the state-of-the-art review published in [11], and summarizes the outstanding issues needing further research:

1) Software size estimation;
2) Software size and complexity metrics;
3) Software cost driver attributes and their effects;
4) Software cost model analysis and refinement;
5) Quantitative models of software project dynamics;
6) Quantitative models of software life-cycle evolution;
7) Software data collection.

1) Software Size Estimation: The biggest difficulty in using today's algorithmic software cost models is the problem of providing sound sizing estimates. Virtually every model requires an estimate of the number of source or object instructions to be developed, and this is an extremely difficult quantity to determine in advance. It would be most useful to have some formula for determining the size of a software product in terms of quantities known early in the software life cycle, such as the number and/or size of the files, input formats, reports, displays, requirements specification elements, or design specification elements.

Some useful steps in this direction are the function-point approach in [2] and the sizing estimation model of [29], both of which have given reasonably good results for small-to-medium sized business programs within a single data processing organization. Another more general approach is given by DeMarco in [17]. It has the advantage of basing its sizing estimates on the properties of specifications developed in conformance with

المنارة للاستشارات

DeMarco's paradigm models for software specifications and designs: number of functional primitives, data elements, input elements, output elements, states, transitions between states, relations, modules, data tokens, control tokens, etc. To date, however, there has been relatively little calibration of the formulas to project data. A recent IBM study [14] shows some correlation between the number of variables defined in a state-machine design representation and the product size in source instructions.

Although some useful results can be obtained on the software sizing problem, one should not expect too much. A wide range of functionality can be implemented beneath any given specification element or I/O element, leading to a wide range of sizes (recall the uncertainty ranges of this nature in Fig. 3). For example, two experiments, involving the use of several teams developing a software program to the same overall functional specification, yielded size ranges of factors of 3 to 5 between programs (see Table X).

TABLE X
SIZE RANGES OF SOFTWARE PRODUCTS PERFORMING SAME FUNCTION

Experiment	Product	No. of Teams	Size Range (source-instr.)
Weinberg & Schulman [55]	Simultaneous linear equations	6	33-165
Boehm, Gray, & Seewaldt [13]	Interactive cost model	7	1514-4606

The primary implication of this situation for practical software sizing and cost estimation is that *there is no royal road to software sizing*. This is no magic formula that will provide an easy and accurate substitute for the process of thinking through and fully understanding the nature of the software product to be developed. There are still a number of useful

المنارة للاستشارات

things that one can do to improve the situation, including the following.

- Use techniques which explicitly recognize the ranges of variability in software sizing. The PERT estimation technique [56] is a good example.
- Understand the primary sources of bias in software sizing estimates. See [11, ch. 21].
- Develop and use a corporate memory on the nature and size of previous software products.

2) Software Size and Complexity Metrics: Delivered source instructions (DSI) can be faulted for being too low-level a metric for use in early sizing estimation. On the other hand, DSI can also be faulted for being too high-level a metric for precise software cost estimation. Various complexity metrics have been formulated to more accurately capture the relative information content of a program's instructions, such as the Halstead Software Science metrics [24], or to capture the relative control complexity of a program, such as the metrics formulated by McCabe in [39]. A number of variations of these metrics have been developed; a good recent survey of them is given in [26].

However, these metrics have yet to exhibit any practical superiority to DSI as a predictor of the relative effort required to develop software. Most recent studies [48], [32] show a reasonable correlation between these complexity metrics and development effort, but no better a correlation than that between DSI and development effort.

Further, the recent [25] analysis of the software science results indicates that many of the published software science "successes" were not as successful as they were previously considered. It indicates that much of the apparent agreement between software science formulas and project data was due to factors overlooked in the data analysis: inconsistent definitions and interpretations of software science quantities, unrealistic or inconsistent assumptions about the nature of the proj-

المنارة للاستشارات

ects analyzed, overinterpretation of the significance of statistical measures such as the correlation coefficient, and lack of investigation of alternative explanations for the data. The software science use of psychological concepts such as the Stroud number have also been seriously questioned in [16].

The overall strengths and difficulties of software science are summarized in [47]. Despite the difficulties, some of the software science metrics have been useful in such areas as identifying error-prone modules. In general, there is a strong intuitive argument that more definitive complexity metrics will eventually serve as better bases for definitive software cost estimation than will DSI. Thus, the area continues to be an attractive one for further research.

3) Software Cost Driver Attributes and Their Effects: Most of the software cost models discussed above contain a selection of cost driver attributes and a set of coefficients, functions, or tables representing the effect of the attribute on software cost (see Table II). Chapters 24–28 of [11] contain summaries of the research to date on about 20 of the most significant cost driver attributes, plus statements of nearly 100 outstanding research issues in the area.

Since the publication of [11] in 1981, a few new results have appeared. Lawrence [35] provides an analysis of 278 business data processing programs which indicate a fairly uniform development rate in procedure lines of code per hour, some significant effects on programming rate due to batch turnaround time and level of experience, and relatively little effect due to use of interactive operation and modern programming practices (due, perhaps, to the relatively repetitive nature of the software jobs sampled). Okada and Azuma [42] analyzed 30 CAD/CAM programs and found some significant effects due to type of software, complexity, personnel skill level, and requirements volatility.

4) Software Cost Model Analysis and Refinement: The most useful comparative analysis of software cost models to

المنارة للاستشارات

date is the Thibodeau [52] study performed for the U.S. Air Force. This study compared the results of several models (the Wolverton, Doty, PRICE S, and SLIM models discussed earlier, plus models from the Boeing, SDC, Tecolote, and Aerospace corporations) with respect to 45 project data points from three sources.

Some generally useful comparative results were obtained, but the results were not definitive, as models were evaluated with respect to larger and smaller subsets of the data. Not too surprisingly, the best results were generally obtained using models with calibration coefficients against data sets with few points. In general, the study concluded that the models with calibration coefficients achieved better results, but that none of the models evaluated were sufficiently accurate to be used as a definitive Air Force software cost estimation model.

Some further comparative analyses are currently being conducted by various organizations, using the database of 63 software projects in [11], but to date none of these have been published.

In general, such evaluations play a useful role in model refinement. As certain models are found to be inaccurate in certain situations, efforts are made to determine the causes, and to refine the model to eliminate the sources of inaccuracy.

Relatively less activity has been devoted to the formulation, evaluation, and refinement of models to cover the effects of more advanced methods of software development (prototyping, incremental development, use of application generators, etc.) or to estimate other software-related life-cycle costs (conversion, maintenance, installation, training, etc.). An exception is the excellent work on software conversion cost estimation performed by the Federal Conversion Support Center [28]. An extensive model to estimate avionics software support costs using a weighted-multiplier technique has recently been developed [49]. Also, some initial experimental results have been obtained on the quantitative impact of prototyping in

المنارة للاستشارات

[13] and on the impact of very high level nonprocedural languages in [58]. In both studies, projects using prototyping and VHLL's were completed with significantly less effort.

5) Quantitative Models of Software Project Dynamics: Current software cost estimation models are limited in their ability to represent the internal dynamics of a software project, and to estimate how the project's phase distribution of effort and schedule will be affected by environmental or project management factors. For example, it would be valuable to have a model which would accurately predict the effort and schedule distribution effects of investing in more thorough design verification, of pursuing an incremental development strategy, of varying the staffing rate or experience mix, of reducing module size, etc.

Some current models assume a universal effort distribution, such as the Rayleigh curve [44] or the activity distributions in [57], which are assumed to hold for any type of project situation. Somewhat more realistic, but still limited are models with phase-sensitive effort multipliers such as PRICE S [22] and Detailed COCOMO [11].

Recently, some more realistic models of software project dynamics have begun to appear, although to date none of them have been calibrated to software project data. The Phister phase-by-phase model in [43] estimates the effort and schedule required to design, code, and test a software product as a function of such variables as the staffing level during each phase, the size of the average module to be developed, and such factors as interpersonal communications overhead rates and error detection rates. The Abdel Hamid-Madnick model [1], based on Forrester's System Dynamics world-view, estimates the time distribution of effort, schedule, and residual defects as a function of such factors as staffing rates, experience mix, training rates, personnel turnover, defect introduction rates, and initial estimation errors. Tausworthe [51] derives and calibrates alternative versions of the SLIM effort-schedule

المنارة للاستشارات

tradeoff relationship, using an intercommunication-overhead model of project dynamics. Some other recent models of software project dynamics are the Mitre SWAP model and the Duclos [21] total software life-cycle model.

6) Quantitative Models of Software Life-Cycle Evolution: Although most of the software effort is devoted to the software maintenance (or life-cycle support) phase, only a few significant results have been obtained to date in formulating quantitative models of the software life-cycle evolution process. Some basic studies by Belady and Lehman analyzed data on several projects and derived a set of fairly general "laws of program evolution" [7], [37]. For example, the first of these laws states:

> "A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version."

Some general quantitative support for these laws was obtained in several studies during the 1970's, and in more recent studies such as [33]. However, efforts to refine these general laws into a set of testable hypotheses have met with mixed results. For example, the Lawrence [36] statistical analysis of the Belady-Lahman data showed that the data supported an even stronger form of the first law ("systems grow in size over their useful life"); that one of the laws could not be formulated precisely enough to be tested by the data; and that the other three laws did not lead to hypotheses that were supported by the data.

However, it is likely that variant hypotheses can be found that are supported by the data (for example, the operating system data supports some of the hypotheses better than does the applications data). Further research is needed to clarify this important area.

المنارة للاستشارات

7) Software Data Collection: A fundamental limitation to significant progress in software cost estimation is the lack of unambiguous, widely-used standard definitions for software data. For example, if an organization reports its "software development man-months," do these include the effort devoted to requirements analysis, to training, to secretaries, to quality assurance, to technical writers, to uncompensated overtime? Depending on one's interpretations, one can easily cause variations of over 20 percent (and often over a factor of 2) in the meaning of reported "software development man-months" between organizations (and similarly for "delivered instructions," "complexity," "storage constraint," etc.) Given such uncertainties in the ground data, it is not surprising that software cost estimation models cannot do much better than "within 20 percent of the actuals, 70 percent of the time."

Some progress towards clear software data definitions has been made. The IBM FSD database used in [53] was carefully collected using thorough data definitions, but the detailed data and definitions are not generally available. The NASA-Goddard Software Engineering Laboratory database [5], [6], [40] and the COCOMO database [11] provide both clear data definitions and an associated project database which are available for general use (and reasonably compatible). The recent Mitre SARE report [59] provides a good set of data definitions.

But there is still no commitment across organizations to establish and use a set of clear and uniform software data definitions. Until this happens, our progress in developing more precise software cost estimation methods will be severely limited.

IV. Software Engineering Economics Benefits and Challenges

This final section summarizes the benefits to software engineering and software management provided by a software engineering economics perspective in general and by software cost

المنارة للاستشارات

estimation technology in particular. It concludes with some observations on the major challenges awaiting the field.

Benefits of a Software Engineering Economics Perspective

The major benefit of an economic perspective on software engineering is that it provides a balanced view of candidate software engineering solutions, and an evaluation framework which takes account not only of the programming aspects of a situation, but also of the human problems of providing the best possible information processing service within a resource-limited environment. Thus, for example, the software engineering economics approach does not say, "we should use these structured structures because they are mathematically elegant" or "because they run like the wind" or "because they are part of the structured revolution." Instead, it says "we should use these structured structures because they provide people with more benefits in relation to their costs than do other approaches." And besides the framework, of course, it also provides the techniques which help us to arrive at this conclusion.

Benefits of Software Cost Estimation Technology

The major benefit of a good software cost estimation model is that it provides a clear and consistent universe of discourse within which to address a good many of the software engineering issues which arise throughout the software life cycle. It can help people get together to discuss such issues as the following.

- Which and how many features should we put into the software product?
- Which features should we put in first?
- How much hardware should we acquire to support the software product's development, operation, and maintenance?
- How much money and how much calendar time should we allow for software development?

المنارة للاستشارات

- How much of the product should we adapt from existing software?
- How much should we invest in tools and training?

Further, a well-defined software cost estimation model can help avoid the frequent misinterpretations, underestimates, overexpectations, and outright buy-ins which still plague the software field. In a good cost-estimation model, there is no way of reducing the estimated software cost without changing some objectively verifiable property of the software project. This does not make it impossible to create an unachievable buy-in, but it significantly raises the threshold of credibility.

A related benefit of software cost estimation technology is that it provides a powerful set of insights on how a software organization can improve its productivity. Many of a software cost model's cost-driver attributes are management controllables: use of software tools and modern programming practices, personnel capability and experience, available computer speed, memory, and turnaround time, software reuse. The cost model helps us determine how to adjust these management controllables to increase productivity, and further provides an estimate of how much of a productivity increase we are likely to achieve with a given level of investment. For more information on this topic, see [11, ch. 33], [12] and the recent plan for the U.S. Department of Defense Software Initiative [20].

Finally, software cost estimation technology provides an absolutely essential foundation for software project planning and control. Unless a software project has clear definitions of its key milestones and realistic estimates of the time and money it will take to achieve them, there is no way that a project manager can tell whether his project is under control or not. A good set of cost and schedule estimates can provide realistic data for the PERT charts, work breakdown structures, manpower schedules, earned value increments, etc., necessary to establish management visibility and control.

المنارة للاستشارات

Note that this opportunity to improve management visibility and control requires a complementary management commitment to define and control the reporting of data on software progress and expenditures. The resulting data are therefore worth collecting simply for their management value in comparing plans versus achievements, but they can serve another valuable function as well: they provide a continuing stream of calibration data for evolving a more accurate and refined software cost estimation models.

Software Engineering Economics Challenges

The opportunity to improve software project management decision making through improved software cost estimation, planning, data collection, and control brings us back full-circle to the original objectives of software engineering economics: to provide a better quantitative understanding of how software people make decisions in resource-limited situations.

The more clearly we as software engineers can understand the quantitative and economic aspects of our decision situations, the more quickly we can progress from a pure seat-of-the-pants approach on software decisions to a more rational approach which puts all of the human and economic decision variables into clear perspective. Once these decision situations are more clearly illuminated, we can then study them in more detail to address the deeper challenge: achieving a quantitative understanding of how people work together in the software engineering process.

Given the rather scattered and imprecise data currently available in the software engineering field, it is remarkable how much progress has been made on the software cost estimation problem so far. But, there is not much further we can go until better data becomes available. The software field cannot hope to have its Kepler or its Newton until it has had its army of Tycho Brahes, carefully preparing the well-defined observational data from which a deeper set of scientific insights may be derived.

المنارة للاستشارات

References

[1] T. K. Abdel-Hamid and S. E. Madnick, "A model of software project management dynamics," in *Proc. IEEE COMPSAC 82,* Nov. 1982, pp. 539–554.

[2] A. J. Albrecht, "Measuring Application Development Productivity," in *SHARE-GUIDE*, 1979, pp. 83–92.

[3] J. D. Aron, "Estimating resources for large programming systems." NATO Sci. Committee, Rome, Italy, Oct. 1969.

[4] J. J. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng.*, IEEE/ACM/NBS, Mar. 1981, pp. 107–116.

[5] V. R. Basili, "Tutorial on models and metrics for software and engineering," IEEE Cat. EHO-167-7, Oct. 1980.

[6] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," Univ. Maryland Technol. Rep. TR-1235, Dec. 1982.

[7] L. A. Belady and M. M. Lehman, "Characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979.

[8] H. D. Benington, "Production of large computer programs," in *Proc. ONR Symp. Advanced Programming Methods for Digital Computers,* June 1956, pp. 15–27.

[9] R. K. D. Black, R. P. Curnow, R. Katz, and M. D. Gray, "BCS software production data," Boeing Comput. Services, Inc., Final Tech. Rep., RADC-TR-77-116, NTIS AD-A039852, Mar. 1977.

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

[11] ——, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[12] B. W. Boehm, J. F. Elwell, A. B. Pyster, E. D. Stuckle, and R. D. Williams, "The TRW software productivity system," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982.

[13] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. specifying: A multi-project experiment," *IEEE Trans. Software Eng.*, to be published.

[14] R. N. Britcher and J. E. Gaffney, "Estimates of software size from state machine designs," in *Proc. NASA-Goddard Software Eng. Workshop,* Dec. 1982.

[15] W. M. Carriere and R. Thibodeau, "Development of a logistics software cost estimating technique for foreign military sales," General Res. Corp., Rep. CR-3-839, June 1979.

[16] N. S. Coulter, "Software science and cognitive psychology," *IEEE Trans. Software Eng.*, pp. 166–171, Mar. 1983.

[17] T. DeMarco, *Controlling Software Projects.* New York: Yourdon, 1982.

المنارة للاستشارات

[18] M. Demshki, D. Ligett, B. Linn, G. McCluskey, and R. Miller, "Wang Institute cost model (WICOMO) tool user's manual," Wang Inst. Graduate Studies, Tyngsboro, MA, June 1982.

[19] H. F. Dircks, "SOFCOST: Grumman's software cost eliminating model," in *IEEE NAECON 1981,* May 1981.

[20] L. E. Druffel, "Strategy for DoD software initiative," RADC/DACS, Griffiss AFB, NY, Oct. 1982.

[21] L. C. Duclos, "Simulation model for the life-cycle of a software product: A quality assurance approach," Ph.D. dissertation, Dep. Industrial and Syst. Eng., Univ. Southern California, Dec. 1982.

[22] F. R. Freiman and R. D. Park, "PRICE software model—Version 3: An overview," in *Proc. IEEE-PINY Workshop on Quantitative Software Models,* IEEE Cat. TH0067-9, Oct. 1979, pp. 32–41.

[23] R. Goldberg and H. Lorin, *The Economics of Information Processing.* New York: Wiley, 1982.

[24] M. H. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.

[25] P. G. Hamer and G. D. Frewin, "M. H. Halstead's software science—A critical examination," in *Proc. IEEE 6th Int. Conf. Software Eng.,* Sept. 1982, pp. 197–205.

[26] W. Harrison, K. Magel, R. Kluczney, and A. DeKock, "Applying software complexity metrics to program maintenance," *Computer,* pp. 65–79, Sept. 1982.

[27] J. R. Herd, J. N. Postak, W. E. Russell, and K. R. Stewart, "Software cost estimation study—Study results," Doty Associates, Inc., Rockville, MD, Final Tech. Rep. RADC-TR-77-220, vol. 1 (of two), June 1977.

[28] C. Houtz and T. Buschbach, "Review and analysis of conversion cost-estimating techniques," GSA Federal Conversion Support Center, Falls Church, VA, Rep. GSA/FCSC-81/001, Mar. 1981.

[29] M. Itakura and A. Takayanagi, " A model for estimating program size and its evaluation," in *Proc. IEEE 6th Software Eng.,* Sept. 1982, pp. 104–109.

[30] R. W. Jensen, "An improved macrolevel software development resource estimation model," in *Proc. 5th ISPA Conf.,* Apr. 1983, pp. 88–92.

[31] R. W. Jensen and S. Lucas, "Sensitivity analysis of the Jensen software model," in *Proc. 5th ISPA Conf.,* Apr. 1983, pp. 384–389.

[32] B. A. Kitchenham, "Measures of programming complexity," *ICL Tech. J.,* pp. 298–316, May 1981.

[33] ——, "Systems evolution dynamics of VME/B," *ICL Tech. J.,* pp. 43–57, May 1982.

[34] W. W. Kuhn, "A software lifecycle case study using the PRICE model," in *Proc. IEEE NAECON,* May 1982.

[35] M. J. Lawrence, "Programming methodology, organizational environment,and programming productivity," *J. Syst. Software,* pp. 257–270, Sept. 1981.

المنارة للاستشارات

[36] ——, "An examination of evolution dynamics," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982, pp. 188–196.

[37] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, pp. 1060–1076, Sept. 1980.

[38] R. D. Luce and H. Raiffa, *Games and Decisions*. New York: Wiley, 1957.

[39] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, pp. 308–320, Dec. 1976.

[40] F. E. McGarry, "Measuring software development technology: What have we learned in six years," in *Proc. NASA-Goddard Software Eng. Workshop*, Dec. 1982.

[41] E. A. Nelson, "Management handbook for the estimation of computer programming costs," Syst. Develop. Corp., AD-A648750, Oct. 31, 1966.

[42] M. Okada and M. Azuma, "Software development estimation study—A model from CAD/CAM system development experiences," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 555–564.

[43] M. Phister, Jr., "A model of the software development process," *J. Syst. Software*, pp. 237–256, Sept. 1981.

[44] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, pp. 345–361, July 1978.

[45] L. H. Putnam and A. Fitzsimmons, "Estimating software costs," *Datamation*, pp. 189–198, Sept. 1979; continued in *Datamation*, pp. 171–178, Oct. 1979 and pp. 137–140, Nov. 1979.

[46] L.H. Putnam, "The real economics of software development," in *The Economics of Information Processing*, R. Goldberg and H. Lorin. New York: Wiley, 1982.

[47] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, pp. 155–165, Mar. 1983.

[48] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. IEEE 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 100–106.

[49] SYSCON Corp., "Avionics software support cost model," USAF Avionics Lab., AFWAL-TR-1173, Feb. 1, 1983.

[50] R. C. Tausworthe, "Deep space network software cost estimation model," Jet Propulsion Lab., Pasadena, CA, 1981.

[51] ——, "Staffing implications of software productivity models," in *Proc. 7th Annu. Software Eng. Workshop*, NASA/Goddard, Greenbelt, MD, Dec. 1982.

[52] R. Thibodeau, "An evaluation of software cost estimating models," General Res. Corp., Rep. T10-2670, Apr. 1981.

[53] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54–73, 1977.

المنارة للاستشارات

[54] G. F. Weinwurm, Ed., *On the Management of Computer Programming*. New York: Auerbach, 1970.
[55] G. M. Weinberg and E. L. Schulman, "Goals and performance in computer programming," *Human Factors*, vol. 16, no. 1, pp. 70–77, 1974.
[56] J. D. Wiest and F. K. Levy, *A Management Guide to PERT/CPM*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
[57] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, pp. 615–636, June 1974.
[58] E. Harel and E. R. McLean, "The effects of using a nonprocedural computer language on programmer productivity," UCLA Inform. Sci. Working Paper 3-83, Nov. 1982.
[59] R. L. Dumas, "Final report: Software acquisition resource expenditure (SARE) data collection methodology," MITRE Corp., MTR 9031, Sept. 1983.

Barry W. Boehm received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957 and the M.A. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, respectively.

From 1978 to 1979 he was a Visiting Professor of Computer Science at the University of Southern California. He is currently a Visiting Professor at the University of California, Los Angeles, and Chief Engineer of TRW's Software Information Systems Division. He was previously Head of the Information Sciences Department at The Rand Corporation, and Director of the 1971 Air Force CCIP-85 study. His responsibilities at TRW include direction of TRW's internal software R&D program, of contract software technology projects, of the TRW software development policy and standards program, of the TRW Software Cost Methodology Program, and the TRW Software Productivity Program. His most recent book is *Software Engineering Economics*, by Prentice-Hall.

Dr. Boehm is a member of the IEEE Computer Society and the Association for Computing Machinery, and an Associate Fellow of the American Institute of Aeronautics and Astronautics.

المنارة للاستشارات

Fred Brooks

G.H. Mealy, B.I. Witt, W.A. Clark

The Functional Structure of OS/360

IBM Systems Journal, Vol. 5 (1), 1966
pp. 2–51

المنارة للاستشارات

In the design of OS/360, *a modular operating system being implemented for a range of* SYSTEM/360 *configurations, the fundamental objective has been to bring a variety of application classes under the domain of one coherent system. The conceptual framework of the system as a whole, as well as the most distinctive structural features of the control program, are heavily influenced by this objective.*

The purpose of this paper is to present the planned system in a unified perspective by discussing design objectives, historical background, and structural concepts and functions. The scope of the system is surveyed in Part I, whereas the rest of the paper is devoted to the control program. Design features relevant to job scheduling and task management are treated in Part II. The third part discusses the principal activities involved in cataloging, storing, and retrieving data and programs.

The functional structure of OS/360

Part I **Introductory survey**
by G. H. Mealy

Part II **Job and task management**
by B. I. Witt

Part III **Data management**
by W. A. Clark

Individual acknowledgements cannot feasibly be given. Contributing to OS/360 are several IBM programming centers in America and Europe. The authors participated in the design of the control program.

للاستشارات

A brief outline of the structural elements of OS/360 *is given in preparation for the subsequent sections on control-program functions.*

Emphasis is placed on the functional scope of the system, on the motivating objectives and basic design concepts, and on the design approach to modularity.

The functional structure of OS/360

Part I Introductory survey

by G. H. Mealy

The environment that may confront an operating system has lately undergone great change. For example, in its several compatible models, SYSTEM/360 spans an entire spectrum of applications and offers an unprecedented range of optional devices.[1] It need come as no surprise, therefore, that OS/360—the Operating System for SYSTEM/360—evinces more novelty in its scope than in its functional objectives.

In a concrete sense, OS/360 consists of a library of programs. In an abstract sense, however, the term OS/360 refers to one articulated response to a composite set of needs. With integrated vocabularies, conventions, and modular capabilities, OS/360 is designed to answer the needs of a SYSTEM/360 configuration with a standard instruction set and thirty-two thousand or more bytes of main storage.[2]

The main purpose of this introductory survey is to establish the scope of OS/360 by viewing the subject in a number of different perspectives: the historical background, the design objectives, and the functional types of program packages that are provided. An effort is made to mention problems and design compromises, i.e., to comment on the forces that shaped the system as a whole.

Basic objectives

The notion of an operating system dates back at least to 1953 and

المنارة للاستشارات

MIT's Summer Session Computer and Utility System.[3] Then, as now, the operating system aimed at non-stop operation over a span of many jobs and provided a computer-accessible library of utility programs. A number of operating systems came into use during the last half of the decade.[4] In that all were oriented toward overlapped setup in a sequentially executed job batch, they may be termed "first generation" operating systems.

A significant characteristic of batched-job operation has been that each job has, more or less, the entire machine to itself, save for the part of the system permanently resident in main storage. During the above-mentioned period of time, a number of large systems—typified by SAGE, MERCURY, and SABRE—were developed along other lines; these required total dedication of machine resources to the requirements of one "real-time" application. It is interesting that one of the earliest operating systems, the Utility Control Program developed by the Lincoln Laboratory, was developed solely for the checkout of portions of the SAGE system. By and large, however, these real-time systems bore little resemblance to the first generation of operating systems, either from the point of view of intended application or system structure.

Because the basic structure of OS/360 is equally applicable to batched-job and real-time applications, it may be viewed as one of the first instances of a "second-generation" operating system. The new objective of such a system is to accommodate an environment of diverse applications and operating modes. Although not to be discounted in importance, various other objectives are not new—they have been recognized to some degree in prior systems. Foremost among these secondary objectives are:

- Increased throughput
- Lowered response time
- Increased programmer productivity
- Adaptability (of programs to changing resources)
- Expandability

throughput

OS/360 seeks to provide an effective level of machine throughput in three ways. First, in handling a stream of jobs, it assists the operator in accomplishing setup operations for a given job while previously scheduled jobs are being processed. Second, it permits tasks from a number of different jobs to concurrently use the resources of the system in a multiprogramming mode, thus helping to ensure that resources are kept busy. Also, recognizing that the productivity of a shop is not solely a function of machine utilization, heavy emphasis is placed on the variety and appropriateness in source languages, on debugging facilities, and on input convenience.

response time

Response time is the lapse of time from a request to completion of the requested action. In a batch processing context, response time (often called "turn-around time") is relatively long: the user gives a deck to the computing center and later obtains printed re-

المنارة للاستشارات

sults. In a mixed environment, however, we find a whole spectrum of response times. Batch turn-around time is at the "red" end of the spectrum, whereas real-time requirements fall at the "violet" end. For example, some real-time applications need response times in the order of milliseconds or lower. Intermediate in the spectrum are the times for simple actions such as line entry from a keyboard where a response time of the order of one or two seconds is desirable. Faced with a mixed environment in terms of applications and response times, OS/360 is designed to lend itself to the whole spectrum of response times by means of control-program options and priority conventions.

productivity

For the sake of programmer productivity and convenience, OS/360 aims to provide a novel degree of versatility through a relatively large set of source languages. It also provides macroinstruction capabilities for its assembler language, as well as a concise job-control language for assistance in job submission.

adaptability

A second-generation operating system must be geared to change and diversity. SYSTEM/360 itself can exist in an almost unlimited variety of machine configurations: different installations will typically have different configurations as well as different applications. Moreover, the configuration at a given installation may change frequently. If we look at application and configuration as the environment of an operating system, we see that the operating system must cope with an unprecedented number of environments. All of this puts a premium on system modularity and flexibility.

Adaptability is also served in OS/360 by the high degree to which programs can be device-independent. By writing programs that are relatively insensitive to the actual complement of input/output devices, an installation can reduce or circumvent the problems historically associated with device substitutions.

expandability

As constructed, OS/360 is "open-ended"; it can support new hardware, applications, and programs as they come along. It can readily handle diverse currency conventions and character sets. It can be tailored to communicate with operators and programmers in languages other than English. Whenever so dictated by changing circumstances, the operating system itself can be expanded in its functional capabilities.

Design concepts

In the notion of an "extended machine," a computing system is viewed as being composed of a number of layers, like an onion.[5,6] Few programmers deal with the innermost layer, which is that provided by the hardware itself. A FORTRAN programmer, for instance, deals with an outer layer defined by the FORTRAN language. To a large extent, he acts as though he were dealing with hardware that accepted and executed FORTRAN statements directly. The SYSTEM/360 instruction set represents two inner layers, one when operating in the supervisor state, another when operating in the problem state.

The supervisor state is employed by OS/360 for the *supervisor* portion of the control program. Because all other programs operate in the problem state and must rely upon unprivileged instructions, they use *system macroinstructions* for invoking the supervisor. These macroinstructions gain the attention of the supervisor by means of SVC, the supervisor-call instruction.

All OS/360 programs with the exception of the supervisor operate in the problem state. In fact, one of the fundamental design tenets is that these programs (compilers, sorts, or the like) are, to all intents and purposes, problem programs and must be treated as such by the supervisor. Precisely the same set of facilities is offered to system and problem programs. At any point in time, the system consists of its given supervisor plus all programs that are available in on-line storage. Inasmuch as an installation may introduce new compilers, payroll programs, etc., the extended machine may grow.

In designing a method of control for a second-generation system, two opposing viewpoints must be reconciled. In the first-generation operating systems, the point of view was that the machine executed an incoming stream of programs; each program and its associated input data corresponded to one application or problem. In the first-generation real-time systems, on the other hand, the point of view was that incoming pieces of data were routed to one of a number of processing programs. These attitudes led to quite different system structures; it was not recognized that these points of view were matters of degree rather than kind. The basic consideration, however, is one of emphasis: programs are used to process data in both cases. Because it is the combination of program and data that marks a unit of work for control purposes, OS/360 takes such a combination as the distinguishing property of a *task*. As an example, consider a transaction processing program and two input transactions, A and B. To process A and B, two tasks are introduced into the system, one consisting of A plus the program, the second consisting of B plus the program. Here, the two tasks use the same program but different sets of input data. As a further illustration, consider a master file and two programs, X and Y, that yield different reports from the master file. Again, two tasks are introduced into the system, the first consisting of the master file plus X, and the second of the master file plus Y. Here the same input data join with two different programs to form two different tasks.

In laying down conceptual groundwork, the OS/360 designers have employed the notion of multitask operation wherein, at any time, a number of tasks may contend for and employ system resources. The term *multiprogramming* is ordinarily used for the case in which one CPU is shared by a number of tasks, the term *multiprocessing*, for the case in which a separate task is assigned to each of several CPU's. Multitask operation, as a concept, gives recognition to both terms. If its work is structured entirely in the form of tasks, a job may lend itself without change to either environment.

المنارة للاستشارات

In OS/360, any named collection of data is termed a *data set*. A data set may be an accounting file, a statistical array, a source program, an object program, a set of job control statements, or the like. The system provides for a cataloged library of data sets. The library is very useful in program preparation as well as in production activities; a programmer can store, modify, recompile, link, and execute programs with minimal handling of card decks.

System elements

As seen by a user, OS/360 will consist of a set of language translators, a set of service programs, and a control program. Moreover, from the viewpoint of system management, a SYSTEM/360 installation may look upon its own application programs as an integral part of the operating system.

translators

A variety of translators are being provided for FORTRAN, COBOL, and RPGL (a Report Program Generator Language). Also to be provided is a translator for PL/I, a new generalized language.[7] The programmer who chooses to employ the assembler language can take advantage of macroinstructions; the assembler program is supplemented by a macro generator that produces a suitable set of assembly language statements for each macroinstruction in the source program.

service programs

Groups of individually translated programs can be combined into a single executable program by a linkage editor. The linkage editor makes it possible to change a program without re-translating more than the affected segment of the program. Where a program is too large for the available main-storage area, the function of handling program segments and overlays falls to the linkage editor.

The sort/merge is a generalized program that can arrange the fixed- or variable-length records of a data set into ascending or descending order. The process can employ either magnetic-tape or direct-access storage devices for input, output, and intermediate storage. The program is adaptable in the sense that it takes advantage of all the input/output resources allocated to it by the control program. The sort/merge can be used independently of other programs or can be invoked by them directly; it can also be used via COBOL and PL/I.

Included in the service programs are routines for editing, arranging, and updating the contents of the library; revising the index structure of the library catalog; printing an inventory list of the catalog; and moving and editing data from one storage medium to another.

the control program

Roughly speaking, the control program subdivides into master scheduler, job scheduler, and supervisor. Central control lodges in the supervisor, which has responsibility for the storage allocation, task sequencing, and input/output monitoring functions. The master scheduler handles all communications to and from the operator, whereas the job scheduler is primarily concerned with

job-stream analysis, input/output device allocation and setup, and job initiation and termination.

supervisor

Among the activities performed by the supervisor are the following:

- Allocating main storage
- Loading programs into main storage
- Controlling the concurrent execution of tasks
- Providing clocking services
- Attempting recoveries from exceptional conditions
- Logging errors
- Providing summary information on facility usage
- Issuing and monitoring input/output operations

The supervisor ordinarily gains control of the central processing unit by way of an interruption. Such an interruption may stem from an explicit request for services, or it may be implicit in SYSTEM/360 conventions, such as in the case of an interruption that occurs at the completion of an input/output operation. Normally, a number of data-access routines required by the data management function are coordinated with the supervisor. The access routines available at any given time are determined by the requirements of the user's program, the structure of the given data sets, and the types of input/output devices in use.

job scheduler

As the basic independent unit of work, a job consists of one or more steps. Inasmuch as each job step results in the execution of a major program, the system formalizes each job step as a task, which may then be inserted into the task queue by the initiator-terminator (a functional element of the job scheduler). In some cases, the output of one step is passed on as the input to another. For example, three successive job steps might involve file maintenance, output sorting, and report tabulation.

The primary activities of the job scheduler are as follows:

- Reading job definitions from source inputs
- Allocating input/output devices
- Initiating program execution for each job step
- Writing job outputs

In its most general form, the job scheduler allows more than one job to be processed concurrently. On the basis of job priorities and resource availabilities, the job scheduler can modify the order in which jobs are processed. Jobs can be read from several input devices and results can be recorded on several output devices—the reading and recording being performed concurrently with internal processing.

master scheduler

The master scheduler serves as a communication control link between the operator and the system. By command, the operator can alert the system to a change in the status of an input/output unit, alter the operation of the system, and request status information. The master scheduler is also used by the operator to alert the job scheduler of job sources and to initiate the reading or processing of jobs.

المنارة للاستشارات

The control program as a whole performs three main functions: job management, task management, and data management. Since Part II of this paper discusses job and task management, and Part III is devoted entirely to data management, we do not further pursue these functions here.

System modularity

Two distinguishable, but by no means independent, design problems arise in creating a system such as OS/360. The first one is to prescribe the range of functional capabilities to be provided; essentially, this amounts to defining two operating systems, one of maximum capability and the other a nucleus of minimum capability. The second problem is to ascertain a set of building blocks that will answer reasonably well to the two predefined operating systems as well as to the diverse needs bounded by the two. In resolving the second problem, which brings us to the subject of modularity, no single consideration is more compelling than the need for efficient utilization of main storage.

As stated earlier, the tangible OS/360 consists of a library of program modules. These modules are the blocks from which actual operating systems can be erected. The OS/360 design exploits three basic principles in designing blocks that provide the desired degree of modularity. Here, these well-known principles are termed parametric generality, functional redundancy, and functional optionality.

parametric generality

The degree of generality required by varying numbers of input/output devices, control units, and channels can be handled to a large extent by writing programs that lend themselves to variations in parameters. This has long been practiced in sorting and merging programs, for example, as well as in other generalized routines. In OS/360, this principle also finds frequent application in the process that generates a specific control program.

functional redundancy

In the effort to optimize performance in the face of two or more conflicting objectives, the most practical solution (at least at the present state of the art) is often to write two or more programs that exploit dissimilar programming techniques. This principle is most relevant to the program translation function, which is especially sensitive to conflicting performance measures. The same installation may desire to effect one compilation with minimum use of main storage (even at some expense of other objectives) and another compilation with maximum efficacy in terms of object-program running time (again at the expense of other objectives). Where conflicting objectives could not be reconciled by other means, the OS/360 designers have provided more than one program for the same general translation or service function. For the COBOL language, for example, there are two translation programs.

For the nucleus of the control program that resides in main storage, the demand for efficient storage utilization is especially

المنارة للاستشارات

functional optionality

pressing. Hence, each functional capability that is likely to be unused in some installations is treated as a separable option. When a control program is generated, each omitted option yields a net saving in the main-storage requirement of the control program.

The most significant control program options are those required to support various job scheduling and multitask modes of operation. These modes carry with them needs for optional functions of the following kinds:

- Task synchronization
- Job-input and job-output queues
- Distinctive methods of main-storage allocation
- Main-storage protection
- Priority-governed selection among jobs

In the absence of any options, the control program is capable of ordinary stacked-job operation. The activities of the central processing unit and the input/output channels are overlapped. Many error checking and recovery functions are provided, interruptions are handled automatically, and the standard data-management and service functions are included. Job steps are processed sequentially through single task operations.

The span of operating modes permitted by options in the control program can be suggested by citing two limiting cases of multitask operation. The first and least complicated permits a scheduled job step to be processed concurrently with an initial-input task, say A, and a result-output task, say B. Because A and B are governed by the control program, they do not correspond to job steps in the usual sense. The major purpose of this configuration is to reduce delays between the processing of successive job steps: tasks A and B are devoted entirely to input/output functions.

In the other limiting case, up to n jobs may be in execution on a concurrent basis, the parameter n being fixed at the time the control program is generated. Contending tasks may arise from different jobs, and a given task can dynamically define other tasks (see the description of the ATTACH macroinstruction in Part II) and assign task priorities. Provision is made for removal of an entire job step (from the job of lowest priority) to auxiliary storage in the event that main storage is exhausted. The affected job step is resumed as soon as the previously occupied main-storage area becomes available again.

In selecting the options to be included in a control program, the user is expected to avail himself of detailed descriptions and accompanying estimates of storage requirements.

system generation

To obtain a desired operating system, the user documents his machine configuration, requests a complement of translators and service programs, and indicates desired control-program options—all via a set of macroinstructions provided for the purpose. Once this has been done, the fabrication of a specific operating system from the OS/360 systems library reduces to a process of two stages.

المنارة للاستشارات

First, the macroinstructions are analyzed by a special program and formulated into a job stream. In the second stage, the assembler program, the linkage editor, and the catalog service programs join in the creation of a resident control program and a desired set of translators and service programs.

Summary comment

Intended to serve a wide variety of computer applications and to support a broad range of hardware configurations, OS/360 is a modular operating system. The system is not only open-ended for the class of functions discussed in this paper, but is based on a conceptual framework that is designed to lend itself to additional functions whenever warranted by cumulative experience.

The ultimate purpose of an operating system is to increase the productivity of an entire computer installation; personnel productivity must be considered as well as machine productivity. Although many avenues to increased productivity are reflected in OS/360, each of these avenues typically involves a marginal investment on the part of an installation. The investment may take the form of additional personnel training, storage requirements, or processing time. It repays few installations to seek added productivity through every possible avenue; for most, the economies of installation management dictate a well-chosen balance between investment and return. Much of the modularity in OS/360 represents a design attempt to permit each installation to strike its own economic balance.

CITED REFERENCES AND FOOTNOTES

1. For an introduction to SYSTEM/360, see G. A. Blaauw and F. P. Brooks, Jr., "The structure of SYSTEM/360, Part I, outline of the logical structure," *IBM Systems Journal* 3, No. 2, 119–135 (1964).
2. The restrictions exclude MODEL 44, as well as MODEL 20. The specialized operating systems that support these excluded models are not discussed here.
3. C. W. Adams and J. H. Laning, Jr., "The MIT systems of automatic coding: Comprehensive, Summer Session, Algebraic," *Symposium on Automatic Coding for Digital Computers*, Office of Naval Research, Department of the Navy (May 1954).
4. In the case of the IBM 709 and 704 computers, the earliest developments were largely due to the individual and group efforts of SHARE installations. The first operating systems developed jointly by IBM and SHARE were the SHARE Operating System (SOS) and the FORTRAN Monitor System (FMS).
5. G. F. Leonard and J. R. Goodroe, "An environment for an operating system," *Proceedings of the 19th National ACM Conference* (August 1964).
6. A. W. Holt and W. J. Turanski, "Man to machine communication and automatic code translation," *Proceedings Western Joint Computer Conference* (1960).
7. G. Radin and H. P. Rogoway, "NPL: highlights of a new programming language," *Communications of the ACM* 8, No. 1, 9–17 (January 1965).

المنارة للاستشارات

This part of the paper discusses the control-program functions most closely related to job and task management.

Emphasized are design features that facilitate diversity in application environments as well as those that support multitask operation.

The functional structure of OS/360

Part II Job and task management

by B. I. Witt

One of the basic objectives in the development of OS/360 has been to produce a general-purpose monitor that can jointly serve the needs of real-time environments, multiprogramming for peripheral operations, and traditional job-shop operations. In view of this objective, the designers found it necessary to develop a more generalized framework than that of previously reported systems. After reviewing salient aspects of the design setting, we will discuss those elements of OS/360 most important to an understanding of job and task management.

Background

Although the conceptual roots of OS/360 task management are numerous and tangled, the basic notion of a task owes much to the systems that have pioneered the use of on-line terminals for inventory problems. This being the case, the relevant characteristics of an on-line inventory problem are worthy of review. We may take the airline seat-reservation application as an example: a reservation request reduces the inventory of available seats, whereas a cancellation adds to the inventory. Because a reply to a ticket agent must be sent within a matter of seconds, there is no opportunity to collect messages for later processing. In the contrasting environment where files are updated and reports made on a daily or weekly basis, it suffices to collect and sort transactions before posting them against a master file.

Three significant consequences of the on-line environment can be recognized:

- Each message must be processed as an independent task
- Because there is no opportunity to batch related requests, each task expends a relatively large amount of time in references to the master file
- Many new messages may be received by the system before the task of processing an older message is completed

What is called for, then, is a control program that can recognize the existence of a number of concurrent tasks and ensure that whenever one task cannot use the CPU, because of input/output delays, another task be allowed to use it. Hence, the CPU is considered a resource that is allocated to a task.

Another major consideration in on-line processing is the size and complexity of the required programs. Indeed, the quantity of code needed to process a transaction can conceivably exceed main storage. Furthermore, subprogram selection and sequence depend upon the content of an input message. Lastly, subprograms brought into main storage on behalf of one transaction may be precisely those needed to process a subsequent transaction. These considerations dictate that subprograms be callable by name at execution time and relocatable at load time (so that they may be placed in any available storage area); they also urge that a single copy of a subprogram be usable by more than one transaction.

The underlying theme is that a task—the work required to process a message—should be an identifiable, controllable element. To perform a task, a variety of system resources are required: the CPU itself, subprograms, space in main and auxiliary storage, data paths to auxiliary storage (e.g., a channel and a control unit), interval timer and others.

Since a number of tasks may be competing for a resource, an essential control program function is to manage the system's resources, i.e., to recognize requests, resolve conflicting demands, and allocate resources as appropriate. In this vein, the general purpose multitask philosophy of the OS/360 control program design has been strongly influenced by task-management ideas that have already been tested in on-line systems.[1] But there is no reason to limit the definition of "task" to the context of real-time inventory transactions. The notion of a task may be extended to any unit of work required of a computing system, such as the execution of a compiler, a payroll program, or a data-conversion operation.

Basic definitions

In the interests of completeness, this section briefly redefines terms introduced in Part I. Familiarity with the general structure of SYSTEM/360 is assumed.[2]

From the standpoint of installation accounting and machine

room operations, the basic unit of work is the *job*. The essential characteristic of a job is its independence from other jobs. There is no way for one job to abort another. There is also no way for the programmer to declare that one job must be contingent upon the output or the satisfactory completion of another job. Job requirements are specified by control statements (usually punched in cards), and may be grouped to form an input job stream. For the sake of convenience, the job stream may include input data, but the main purpose of the job stream is to define and characterize jobs. Because jobs are independent, the way is open for their concurrent execution.

job step

By providing suitable control statements, the user can divide a job into *job steps*. Thus, a job is the sum of all the work associated with its component job steps. In the current OS/360, the steps of a given job are necessarily sequential: only one step of a job can be processed at a time. Furthermore, a step may be conditional upon the successful completion of one or more preceding steps; if the specified condition is not met, the step in question can be bypassed.

task

Whenever the control program recognizes a job step (as the result of a job control statement), it formally designates the step as a *task*. The task consists, in part or in whole, of the work to be accomplished under the direction of the program named by the job step. This program is free to invoke other programs in two ways, first within the confines of the original task, and second within the confines of additionally created tasks. A task is created (except in the special case of initial program loading) as a consequence of an ATTACH macroinstruction. At the initiation of a job step, ATTACH is issued by the control program; during the course of a job step, ATTACH's may be issued by the user's programs.

From the viewpoint of the control system, all tasks are independent in the sense that they may be performed concurrently. But in tasks that stem from one given job (which implies that they are from the same job step), dependency relationships may be inherent because of program logic. To meet this possibility, the system provides means by which tasks from the same job can be synchronized and confined within a hierarchical relationship. As a consequence, one task can await a designated point in the execution of another task. Similarly, a task can wait for completion of a subtask (a task lower in the hierarchy). Also, a task can abort a subtask.

Although a job stream may designate many jobs, each of which consists of many job steps and, in turn, leads to many tasks, a number of quite reasonable degenerate cases may be imagined; e.g., in an on-line inventory environment, the entire computing facility may be dedicated to a single job that consists of a single job step. At any one time, this job step may be comprised of many tasks, one for each terminal transaction. On the other hand, in many installations, it is quite reasonable to expect almost all jobs to consist of several steps (e.g., compile/link-edit/execute) with

المنارة للاستشارات

no step consisting of more than one task.

In most jobs, the executable programs and the data to be processed are not new to the system—they are carried over from earlier jobs. They therefore need not be resubmitted for the new job; it is sufficient that they be identified in the control statements submitted in their place as part of a job stream. A job stream consists of such control statements, and optionally of data that is new to the system (e.g., unprocessed keypunched information). Control statements are of six types; the three kinds of interest here are *job*, *execute*, and *data definition* statements.

control statements

The first statement of each job is a job statement. Such a statement can provide a job name, an account number, and a programmer's name. It can place the job in one of fifteen priority classes; it can specify various conditions which, if not met at the completion of each job step, inform the system to bypass the remaining steps.

The first statement of each job step is an execute statement. This statement typically identifies a program to be executed, although it can be used to call a previously cataloged procedure into the job stream. The first statement can designate accounting modes, conditional tests that the step must meet with respect to prior steps, permissable execution times, and miscellaneous operating modes.

A data definition statement permits the user to identify a data set, to state needs for input/output devices, to specify the desired channel relationships among data sets, to specify that an output data set be passed to a subsequent job step, to specify the final disposition of a data set, and to incorporate other operating details.

program structure

In OS/360, a ready-for-execution program consists of one or more subprograms called *load modules*; the first load module to be executed is the one that is named in the execute control statement. At the option of the programmer, a program can take one of the following four structures:

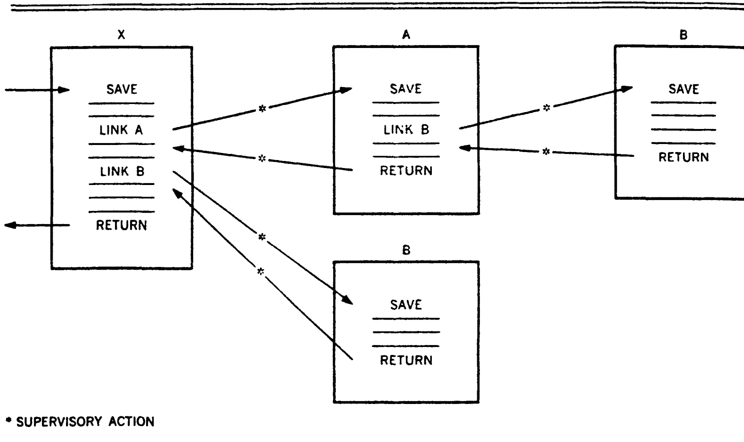
Simple structure. One load module, loaded into main storage as an entity, contains the entire program.

Planned overlay structure. The program exists in the library as a single load module, but the programmer has identified program segments that need not be in main storage at the same given time. As a consequence, one area of storage can be used and reused by the different segments. The OS/360 treatment of this structure follows the guide lines previously laid down by Heising and Larner.[3] A planned overlay structure can make very effective use of main storage. Because the control system intervenes only once to find a load module, and linkages from segment to segment are aided by symbol resolution in advance of execution, this structure also serves the interest of execution efficiency.

Dynamic serial structure. The advantages of planned overlay tend to diminish as job complexity increases, particularly if the selection

ات

Figure 1

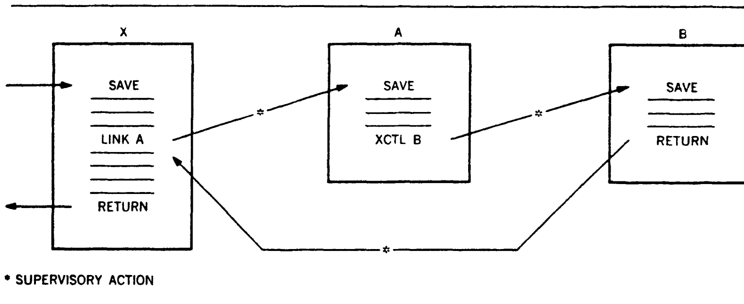


of segments is data dependent (as is the case in most on-line inventory problems). For this situation, OS/360 provides means for calling load modules dynamically, i.e., when they are named during the execution of other load modules. This capability is feasible because main storage is allocated as requests arise, and the conventions permit any load module to be executed as a subroutine. It is consistent with the philosophy that tasks are the central element of control, and that all resources required by a task for its successful performance—the CPU, storage, *and programs*—may be requested whenever the need is detected. In the dynamic serial structure, more than one load module is called upon during the course of program execution. Following standard linkage conventions, the control system acts as intermediary in establishing subroutine entry and return. Three macroinstructions are provided whereby one load module can invoke another: LINK, XCTL (transfer control), and LOAD.

The action of LINK is illustrated in Figure 1. Of the three programs (i.e., load modules) involved, X is the only one named at task-creation time. One of the instructions generated by LINK is a supervisor call (SVC), and the program name (such as A or B in the figure) is a linkage parameter. When the appropriate program of the control system is called, it finds, allocates space for, fetches, and branches to the desired load module. Upon return from the module (effected by the macroinstruction RETURN), the occupied space is liberated but not reused unless necessary. Thus, in the example, if program B is still intact in main storage at the second call, it will not be fetched again (assuming that the user is operating under "reusable" programming conventions, as discussed below).

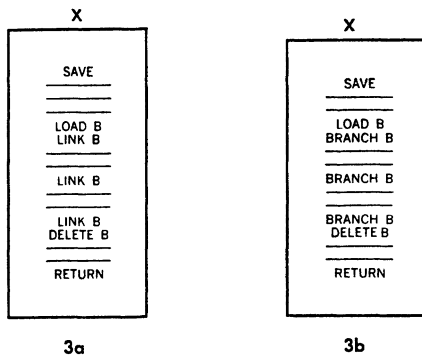
As suggested by Figure 2, XCTL can be used to pass control to successive phases of a program. Standard linkage conventions are observed, parameters are passed explicitly, and the supervisor

Figure 2



* SUPERVISORY ACTION

Figure 3a, 3b



3a 3b

functions are similar to those needed for LINK. However, a program transferring via XCTL is assumed to have completed its work, and all its allocated areas are immediately liberated for reuse.

The LOAD macroinstruction is designed primarily for those cases in which tasks make frequent use of a load module, and reusable conventions are followed. LOAD tells the supervisor to bring in a load module and to preserve the module until liberated by a DELETE macroinstruction (or automatically upon task termination). Control can be passed to the module by a LINK, as in Figure 3a, or by branch instructions, as in Figure 3b.

Dynamic parallel structure. In the three foregoing structures, execution is serial. The ATTACH macroinstruction, on the other hand, creates a task that can proceed in parallel with other tasks, as permitted by availability of resources. In other respects, ATTACH is much like LINK. But since ATTACH leads to the creation of a new task, it requires more supervisor time than LINK and should not be used unless a significant degree of overlapped operation is assured.

program usability

Load modules in the library are of three kinds (as specified by the programmer at link-edit time): *not reusable*, *serially reusable*, and *reenterable*. Programs in the first category are fetched directly

المنارة للاستشارات

from the library whenever needed. This is required because such programs may alter themselves during execution in a way that prevents the version in main storage from being executed more than once.

A serially reusable load module, on the other hand, is designed to be self-initializing; any portion modified in the course of execution is restored before it is reused. The same copy of the load module may be used repeatedly during performance of a task. Moreover, the copy may be shared by different tasks created from the same job step; if the copy is in use by one task at the time it is requested by another task, the latter task is placed in a queue to wait for the load module to become available.

A reenterable program, by design, does not modify itself during execution. Because reenterable load modules are normally loaded in storage areas protected by the storage key used for the supervisor, they are protected against accidental modification from other programs. A reenterable load module can be loaded once and used freely by any task in the system at any time. (A reenterable load module fetched from a private library, rather than from the main library, is made available only to tasks originating from the same job step.) Indeed, it can be used concurrently by two or more tasks in multitask operations. One task may use it, and before the module execution is completed, an interruption may give control to a second task which, in turn, may reenter the module. This in no way interferes with the first task resuming its execution of the module at a later time.

In a multitask environment, concurrent use of a load module by two or more tasks is considered normal operation. Such use is important in minimizing main storage requirements and program reloading time. Many OS/360 control routines are written in reenterable form.

A reenterable program uses machine registers as much as possible; moreover, it can use temporary storage areas that "belong" to the task and are protected with the aid of the task's storage key. Temporary areas of this sort can be assigned to the reenterable program by the calling program, which uses a linkage parameter as a pointer to the area. They can also be obtained dynamically with the aid of the GETMAIN macroinstruction in the reenterable program itself. GETMAIN requests the supervisor to allocate additional main storage to the task and to point out the location of the area to the requesting program. Note that the storage obtained is assigned to the task, and not to the program that requested the space. If another task requiring the same program should be given control of the CPU before the first task finishes its use of the program, a *different* block of working storage is obtained and allocated to the second task.

Whenever a reenterable program (or for that matter any program) is interrupted, register contents and program status word are saved by the supervisor in an area associated with the interrupted task. The supervisor also keeps all storage belonging to the

للاستشارات

task intact—in particular, the working storage being used by the reenterable program. No matter how many intervening tasks use the program, the original task can be allowed to resume its use of the program by merely restoring the saved registers and program status word. The reenterable program is itself unaware of which task is using it at any instant. It is only concerned with the contents of the machine registers and the working storage areas pointed to by designated registers.

Job management

The primary functions of job management are

- Allocation of input/output devices
- Analysis of the job stream
- Overall scheduling
- Direction of setup activities

In the interests of efficiency, job management is also empowered to transcribe input data onto, and user output from, a direct-access device.

In discussing the functions of OS/360, a distinction must be made between job management and task management. Job management turns each job step over to task management as a formal task, and then has no further control over the job step until completion or abnormal termination. Job management primes the pump by defining work for task management; task management controls the flow of work. The functions of task management (and to some degree of data management) consist of the fetching of required load modules; the dynamic allocation of CPU, storage space, channels, and control units on behalf of competing tasks; the services of the interval timer; and the synchronization of related tasks.

schedulers

Job management functions are accomplished by a *job scheduler* and a *master scheduler*. The job scheduler consists mainly of control programs with three types of functions: read/interpret, initiate/terminate, and write. The master scheduler is limited in function to the handling of operator commands and messages to the console operator.

In its most general form, the job scheduler permits priority scheduling as well as sequential scheduling. The *sequential scheduling system* is suggested by Figure 4. A reader/interpreter scans the control statements for one job step at a time. The initiator allocates input/output devices, notifies the operator of the physical volumes (tape reels, removable disks, or the like) to be mounted, and then turns the job step over to task management.

In a *priority scheduling system*, as suggested by Figure 5, jobs are not necessarily executed as encountered in an input job stream. Instead, control information associated with each job enters an input work queue, which is held on a direct-access device. Use of this queue, which can be fed by more than one input job stream, permits the system to react to job priorities and delays caused by

the mounting and demounting of input/output volumes. The initiator/terminator can look ahead to future job steps (in a given job) and issue volume-mounting instructions to the operator in advance.

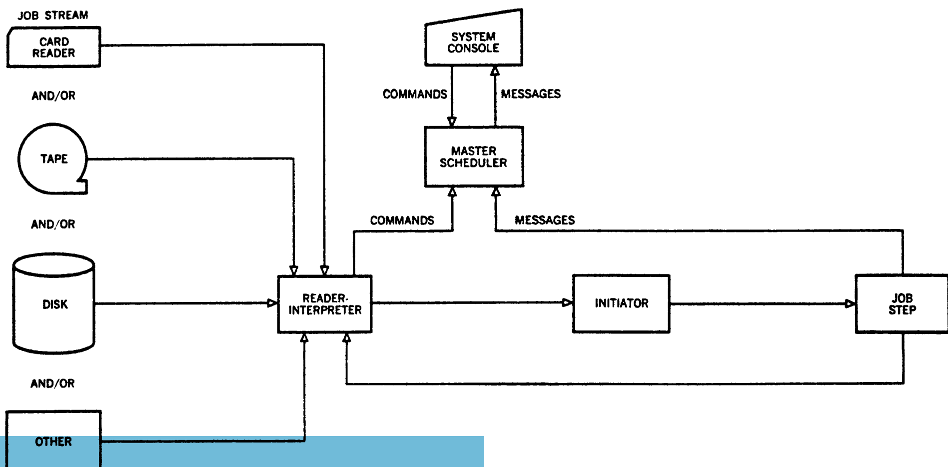
Some versions of the system have the capability of processing jobs in which control information is submitted from remote on-line terminals. A reader/interpreter task is attached to handle the job control statements, and control information is placed in the input work queue and handled as in the case of locally submitted jobs. Output data sets from remote jobs are routed to the originating terminal.

For each step of a selected job, the initiator ensures that all necessary input/output devices are allocated, that direct-access storage space is allocated as required, and that the operator has mounted any necessary tape and direct-access volumes. Finally, the initiator requests that the supervisor lend control to the program named in the job step. At job step completion, the terminator removes the work description from control program tables, freeing input/output devices, and disposing of data sets.

multijob initiation

One version of the initiator/terminator, optional for larger systems where it is practical to have more than one job from the input work queue under way, permits *multijob initiation*. When the system is generated, the maximum number of jobs that are allowed to be executed concurrently can be specified. Although each selected job is run one step at a time, jobs are selected from the queue and initiated as long as (1) the number of jobs specified

Figure 4



المنارة للاستشارات

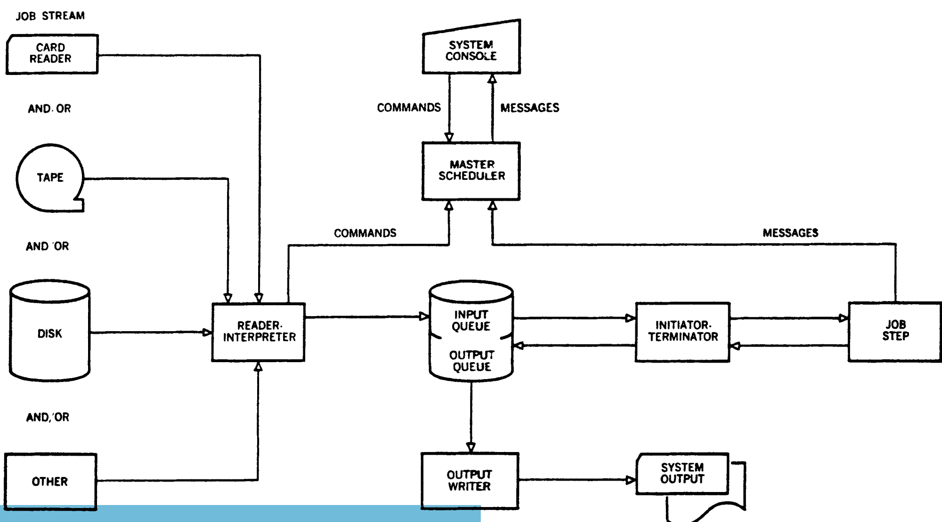
by the user is not exceeded; (2) enough input/output devices are available; (3) enough main storage is available; (4) jobs are in the input work queue ready for execution; and (5) the initiator has not been detached by the operator.

Multijob initiation may be used to advantage where a series of local jobs is to run simultaneously with an independent job requiring input from remote terminals. Typically, telecommunication jobs have periods of inactivity, due either to periods of low traffic or to delays for direct-access seeks. During such delays, the locally available jobs may be executed.

During execution, output data sets may be stored on a direct-access storage device. Later, an output writer can transcribe the data to a system output device (normally a printer or punch). Each system output device is controlled by an output writer task. Moreover, output devices can be grouped into usage classes. For example, a single printer might be designated as a class for high-priority, low-volume printed output, and two other printers as a class for high-volume printing. The data description statement allows output data sets to be directed to a class of devices; it also allows a specification that places a reference to the data on the output work queue. Because the queue is maintained in priority sequence, the output writers can select data sets on a priority basis.

In systems with input and output work queues, the output writer is the final link in a chain of control routines designed to ensure low turn-around time, i.e., time from entry of the work

Figure 5



المنارة للاستشارات

statement to a usable output. At two intermediate stages of the work flow, data are accessible as soon as prepared, without any requirement for batching; and at each of these stages, priorities are used to place important work ahead of less important work that may have been previously prepared. These stages occur when the job has just entered the input work queue, and when the job is completed with its output noted in the output work queue.

Note that a typical priority scheduling system, even one that handles only a single job at a time, may require multitask facilities to manage the concurrent execution of a reader, master scheduler, and a single user's job.

Task management

As stated earlier, job management turns job steps over to task management, which is implemented in a number of supervisory routines. All work submitted for processing must be formalized as a task. (Thus, a program is treated as data until named as an element of a task.) A task may be performed in either a *single-task* or *multitask* environment. In the single task environment, only one task can exist at any given time. In the multitask environment, several tasks may compete for available resources on a priority basis. A program that is written for the single-task environment and follows normal conventions will work equally well in the multitask environment.

single-task operation

In a single-task environment, the job scheduler operates as a task that entered the system when the system was initialized. Each job step is executed as part of this task, which, as the only task in the system, can have all available resources. Programs can have a simple, overlay, or dynamic serial structure.

The control program first finds the load module named in the EXEC statement. Then it allocates main storage space according to program attributes stated in the library directory entry for the load module, and loads the program into main storage. Once the load module (or root segment, in the case of overlay) is available in main storage, control is passed to the entry point. If the load module fetched is the first subprogram of a dynamic serial program, the subsequent load modules required are fetched in the same way as the first, with one exception: if the needed module is reusable and a copy is already in main storage, that copy is used for the new requirement.

When the job step is completed, the supervisor informs the job scheduler, noting whether completion was normal or abnormal.

By clearly distinguishing among tasks, the control system can allow tasks to share facilities where advantageous to do so.

multitask operation

Although the resource allocation function is not absent in a single-task system, it comes to the fore in a multitask system. The system must assign resources to tasks, keep track of all assignments, and ensure that resources are appropriately freed upon task completion. If several tasks are waiting for the same resource, queuing of requests is required.

الاستشارات

Each kind of resource is managed by a separate part of the control system. The CPU manager, called the *task dispatcher*, is part of the supervisor; the queue on the CPU is called the *task queue*. The task queue consists of task control blocks ordered by priority. There is one task control block for each task in the system. Its function is to contain or point to all control information associated with a task, such as register and program-status-word contents following an interrupt, locations of storage areas allocated to the task, and the like. A task is *ready* if it can use the CPU, and *waiting* if some event must occur before the task again needs the CPU.

A task can enter the waiting state directly via the WAIT macroinstruction, or it may lapse into a waiting state as a result of other macroinstructions. An indirect wait may occur, for example, as a result of a GET macroinstruction, which requests the next input record. If the record is already in a main storage buffer area, the control program is not invoked and no waiting occurs; otherwise, a WAIT is issued by the GET routine and the task delayed until the record becomes available.

Whenever the task dispatcher gains control, it issues the Load Program Status Word instruction that passes control to the ready task of highest priority. If none of the tasks are ready, the task dispatcher then instructs the CPU to enter the hardware waiting condition.

By convention, the completed use of a resource is always signaled by an interruption, whereupon the appropriate resource manager seizes control.

Let *subtask* denote a task attached by an existing task within a job step. Subtasks can share some of the resources allocated to the attaching task—notably the storage protection key, main storage areas, serially reusable programs (if not already in use), reenterable programs, and data sets (as well as the devices on which they reside). Data sets for a job step are initially presented to the job scheduler by data definition statements. When the job scheduler creates a task for the job step, these data sets become available to all load modules operating under the task, with no restriction other than that data-set boundaries be heeded. When the task attaches a subtask, it may pass on the location of any data control block: using this, the subtask gains access to the data set.

synchronized events

We have mentioned the ways by which an active task can enter a waiting state in anticipation of some specific event. After the event has occurred, the required notification is effected with the aid of the POST macroinstruction. If the event is governed by the control program, as in the instance of a read operation, the supervisor issues the POST; for events unknown to the supervisor, a user's program (obviously not part of the waiting task) must issue a POST.

A task program may issue several requests and then await the completion of a given number of them. For example, a task may specify by READ, WRITE, and ATTACH macroinstructions that

للاستشارات

three asynchronous activities be performed, but that as soon as two have been completed, the task be placed in the ready condition. When each of these requests is initially made to the control program, the location of a one-word *event control block* is stated. The event control block provides communication between the task (which issued the original request and the subsequent WAIT) and the posting agency—in this case, the control program. When the WAIT macroinstruction is issued, its parameters supply the addresses of the relevant event control blocks. Also supplied is a *wait count* that specifies how many of the events must occur before the task is ready to continue.

When an event occurs, a *complete* flag in the appropriate event control block is set by the POST macroinstruction, and the number of complete flags is tested against the wait count. If they match, the task is placed in the ready condition. A *post code* specified in the POST macroinstruction is also placed in the event control block; this code gives information regarding the manner in which completion occurred. After the task again gains control, the user program can determine which events occurred and in what manner.

Requests for services may result in waits of no direct concern to the programmer, as, for example, in the case of the GET macroinstruction previously mentioned. In all such instances, event control blocks and wait specifications are handled entirely by the supervisor.

Another form of synchronization allows cooperating tasks to share certain resources in a "serially reusable" way. The idea (already invoked in the discussion of programs) may be applied to any shared facility. For example, the facility may be a table that has to be updated by many tasks. In order to produce the desired result, each task must complete its use of the table before another task gains access to it (just as each task had to complete its use of a self-initiating program before another task was allowed to use the program). To control access to such a facility, the programmer may create a queue of all tasks requiring access, and limit access to one task at a time. Queuing capabilities are provided by two macroinstructions: enqueue (ENQ) and dequeue (DEQ). The nature of the facility, known only to the tasks that require it, is of no concern to the operating system so long as a queue control block associated with the facility is provided by the programmer. ENQ causes a request to be placed in a queue associated with the queue control block. If the *busy indicator* in the control block is on, the task issuing the ENQ is placed in the wait condition pending its turn at the facility. If the busy indicator is off, the issuing task becomes first in the queue, the busy indicator is turned on, and control is returned to the task. When finished with the facility, a task liberates the facility and posts the next task on the queue by issuing DEQ.

In a multitask operation, competing requests for service or resources must be resolved. In some cases, choices are made by

للاستشارات

considering hardware optimization, as, for example, servicing requests for access to a disk in a fashion that minimizes disk seeking time. In most cases, however, the system relies upon a priority number provided by the user. The reason for this is that the user can best select priority criteria. He may reconcile such factors as the identification of the job requestor, response-time requirements, the amount of time already allocated to a task, or the length of time that a job has been in the system without being processed. The net result is stated in a priority number ranging from 0 to 14 in order of increasing importance.

task priority

Initial priorities, specified in job statements, affect the sequence in which jobs are selected for execution. The operator is free to modify such priorities up to the time that the job is actually selected. Changes to priorities may be made dynamically by the change priority (CHAP) macroinstruction, which allows a program to modify the priority of either the active task or of any of its subtasks. Means are provided whereby unauthorized modification can be prevented.

When the job scheduler initiates a job step, the current priority of the job is used to establish a *dispatch priority* and a *limit priority*. The former is used by the resource managers, where applicable, to resolve contention for a resource. The limit priority, on the other hand, serves to control dynamic priority assignments. CHAP permits each task to change its dispatching priority to any point in the range between zero and its limit. Furthermore, when a task attaches a subtask, it is free to set the subtask's dispatching and limit priorities at any point in the range between zero and the limit of the attacher; the subtask's dispatching priority can however exceed that of the attacher. For example, were task A, with limit and dispatching priorities both equal to 10, to attach subtask B with a higher relative dispatching priority than itself, it could use CHAP to lower its own dispatching priority to 7 and attach B with limit and dispatching priorities set to 8.

It is expected that most installations will ordinarily use three levels of priority for batch-processing jobs. Normal work will automatically be assigned a median priority. A higher number will be used for urgent jobs, and a lower one for low-priority work.

task termination

Normally, programs are expected to signal completion of their execution by RETURN or XCTL. If the program at the highest control level within the task executes a RETURN, the supervisor treats it as an end-of-task signal. Whenever RETURN is used, one of the general registers is used to transmit a return code to the caller. The return code at task termination may be inspected by the attaching task, and is used by the job scheduler to evaluate the condition parameters in job control statements. It may, for example, find that all remaining steps are to be skipped.

In addition, any program operating on behalf of a task can execute a macroinstruction to discontinue task execution abnormally. The control program then takes appropriate action to liberate resources, dispose of data sets, and remove the task con-

المنارة للاستشارات

trol block. Although abnormal termination of a task causes abnormal termination of all subtasks, it is possible for abnormal subtasks to terminate without causing termination of the attaching task.

main storage allocation

The supervisor is designed to allocate main storage dynamically, when space is demanded by a task or the control program itself. An *implicit* request is generated internally within the control program, on behalf of some other control program service. An example is LINK, in which the supervisor finds a program, allocates space, and fetches it. To make *explicit* requests for additional main storage areas, a user program may employ the GETMAIN or GETPOOL macroinstructions.

Also provided are means for dynamic release of main storage areas. Implicit release may take place when a program is no longer in use, as signaled by RETURN, XCTL, or DELETE. Explicit release is requested by the FREEMAIN or FREEPOOL macroinstructions.

Explicit allocation by GETMAIN can be for fixed or variable areas, and can be conditional or unconditional:

- *Fixed area.* The amount of storage requested is explicitly given.
- *Variable area.* A minimum acceptable amount of storage is specified, as well as a larger amount preferred. If the larger amount is not available, the supervisor responds to the request with the largest available block that equals or exceeds the stated minimum.
- *Conditional.* Space is requested if available, but the program can proceed without it.
- *Unconditional.* The task cannot proceed without the requested space.

storage protection

The operating system uses the SYSTEM/360 storage protection feature to protect storage areas controlled by the supervisor from damage by user jobs and to protect user jobs from each other. This is done by assigning different protection keys to each of the job steps selected for concurrent execution. However, if multiple tasks result from a single job step (by use of the ATTACH macroinstruction), all such tasks are given the same protection key to allow them to write in common communication areas.

Each job step is assigned two logically different pools, each consisting of one or more storage blocks. The first of these is used to store non-reusable and serially-reusable programs from any source, and reenterable programs from sources other than the main library. This pool is not designated by number. The second pool, numbered 00, is used for any task work areas obtained by the supervisor and for filling all GETMAIN or GETPOOL requests—unless a non-zero pool number is specified.

When the highest-level task of a job step is terminated, all storage pools are released for reassignment. However, when a task attaches a subtask, and makes storage areas available to the

للاستشارات

subtask, it may suit the purposes of the task not to have the storage areas released upon completion of the subtask. To provide for this possibility, programs may call for the creation of pools numbered 01 or higher. Such a pool may be made available to a subtask in either of two ways—that is, by *passing* or *sharing*. If a pool created by a task is passed to a subtask, termination of the subtask results in release of the pool. On the other hand, subtask termination does not result in the release of a shared pool. In both cases, the subtask that receives a pool may add to the pool, delete from it, or release it in the same way as the originating task.

passing and sharing

Because Pool 00 refers to the same set of storage blocks for all tasks in a job step, it need not be passed or shared, and is not released until the job step is completed.

If two or more job steps are being executed, and one requires more additional main storage than is available for allocation, the control program intervenes. First, the supervisor attempts to free space occupied by a program that is neither in use nor reserved by a task. Failing that, it may suspend the execution of one or more tasks by storing the associated information in auxiliary storage. The storage and retrieval operations occasioned by competing demands for main-storage space are termed *roll-out* and *roll-in.*

roll-out and roll-in

The decision to roll out one or more tasks is made on the basis of task priorities. A main storage demand by a task can cause as many lower-priority tasks to be rolled out as necessary to satisfy the demand. If the lowest-priority task in the system needs more space to continue, it is placed in a wait state pending main storage availability.

During roll-out, all tasks operating under a single job step are removed as a group. Input/output operations under way at the time of the roll-out are allowed to reach completion.

Roll-in takes place automatically as soon as the original space is again available, and execution continues where it left off. Since its task control block remains in a wait status and its input/output units are not altered, a task may still be considered in the system after roll-out.

Significance of multitask operations

It may be expected that multitask operations will not only provide powerful capabilities for many existing environments, but will also serve as a foundation for more complex environments for some time to come.

Fast turnaround in job-shop operations is achieved by allowing concurrent operation of input readers, output writers, and user's programs. It is possible to handle a wide variety of telecommunication activities, each of which is characterized by many tasks (most of them in wait conditions). Also, complex problems can be programmed in segments that concurrently share system resources and hence optimize the use of those resources. With some versions of the job scheduler, multitask operations permit

job steps from several different jobs to be established as concurrent tasks. To serve such current multitask needs, the structure of the control system consists of two primary classes of elements: (1) queues representing unsatisfied requirements of tasks for certain resources, and (2) tables identifying available resources. Some of the control information is in main storage; some is in auxiliary storage. This structure facilitates dynamic configuration changes, such as addition or removal of programs in main storage, and attached input/output devices.

Perhaps more important for future systems, the structure may prove adaptable in the management of additional CPU's. For example, if multiple CPU's were given access to the job queue (now stored on a disk), each CPU could queue new jobs as well as initiate jobs already on the queue. Similarly, if multiple CPU's were given access to main storage, each CPU could add tasks to the task queue and dispatch tasks already on the task queue. That is, a system could be designed wherein, by executing the task-dispatcher control routine (which itself is in the shared main storage), any CPU could be assigned a ranking task on the queue; and while executing a task, any CPU could add new tasks to the queue by means of the ATTACH macroinstruction.

Summary

In OS/360, for which the basic unit of work is the task, resources are allocated only to tasks. In general, resources are allocated dynamically to permit easier planning on the part of the programmer, achieve more efficient utilization of storage space, and open the way for concurrent execution of a number of tasks.

Users notify the system of work requirements by defining each job as a sequence of job-control statements. The number of tasks entailed by a job depends upon the nature of the job. The system permits job definitions to be cataloged, thereby simplifying the job resubmittal process. Reading of job specifications and source data, printing of job results, and job execution can occur simultaneously for different jobs. Job inputs and outputs may be queued in direct-access storage, thereby avoiding the need for external batching and permitting priority-governed job execution. In its multijob-initiation mode, the system can process a number of jobs concurrently.

CITED REFERENCES AND FOOTNOTE

1. An historic review of operating systems, with emphasis on I/O control and job scheduling, appears in Reference 4. Operating systems that provided for multiprogramming are described in References 5, 6, and 7. One on-line inventory application is described in Reference 8, and some indication of techniques used in its solution are given in References 9 and 10.
2. G. A. Blaauw and F. P. Brooks, Jr., "The structure of SYSTEM/360: Part I—outline of the logical structure," *IBM Systems Journal* **3**, No. 2, 119–135 (1964).
3. W. P. Heising and R. A. Larner, "A semi-automatic storage allocation

المنارة للاستشارات

system at loading time," *Communications of the ACM* 4, No. 10, 446–449 (October 1961).
4. T. B. Steel, Jr., "Operating systems," *Datamation* **10**, No. 5, 26–28 (May 1964).
5. E. S. McDonough, "STRETCH experiment in multiprogramming," *Digest of Technical Papers*, ACM 62 National Conference, 28 (1962).
6. E. F. Codd, "Multi-programming," *Advances in Computers*, Volume 3, Edited by Franz L. Alt and Morris Rubinoff.
7. G. F. Leonard, "Control techniques in the CL-II Programming System," *Digest of Technical Papers*, ACM 62 National Conference, 29 (1962).
8. M. N. Perry and W. R. Plugge, "American Airlines SABRE electronic reservation system," WJCC *Proceedings*, 593–601 (May 1961).
9. M. N. Perry, "Handling of very large programs," *Proceedings of IFIP Congress 65*, Volume 1, 243–247 (1965).
10. W. B. Elmore and G. J. Evans, Jr., "Dynamic control of core memory in a real-time system," *Proceedings of IFIP Congress 65*, Volume 1, 261–266 (1965).

المنارة للاستشارات

Concepts underlying the data-management capabilities of OS/360 *are introduced; distinctive features of the access methods, catalog, and relevant system macroinstructions are discussed.*

To illustrate the way in which the control program adapts to actual input/output requirements, a read operation is examined in considerable detail.

The functional structure of OS/360

Part III Data management

by W. A. Clark

The typical computer installation is confronted today with an imposing mass of data and programs. Moreover, with the applicable technologies developing at a rapid pace, the current trend is toward increasing diversity and change in input/output and auxiliary storage devices. Together, these factors dictate that the so-called "input/output" process be viewed in new perspective. Whereas the support provided by a conventional input/output control system is usually limited to data transfer and label processing, the current need is for a data management system that encompasses identification, storage, survey, and retrieval needs—for programs as well as data. Not only should the system employ the capabilities of both direct-access and serial-access devices, but ideally should be able to satisfy a storage or input/output requirement with any storage device that meets the functional specifications of the given requirement.

Our purpose here is to discuss the main structural aspects of OS/360 from the standpoint of data management. In identifying, storing, and retrieving programs and data via OS/360, a programmer normally reckons with device classes rather than specific devices. Because actual devices are not assigned until job-step execution time, a novel degree of device independence is achieved. Moreover, as befits a system intended for a wide range of applications, OS/360

المنارة للاستشارات

provides for several data organizations and search schemes. Various buffering and transmittal options are provided.

Background

Although the data management services provided by OS/360 are deliberately similar to those provided by predecessor systems, the system breaks with the past in the manner in which it adapts to specific needs.

compiled I/O routines

For mobilizing the input/output routines needed in a given job step, one well-known scheme places these routines into the user's program during the compilation process. No post-assembly program fetching or editing is then required; a complete, executable program results. This scheme has significant disadvantages. It requires that a fairly complete description of device types and intended modes of operation be stated in the source program. Compilation is made more difficult by having to concern itself with details of the input/output function, and compiled programs can be made obsolete by environmental changes that affect the input/output function.

interpretive I/O routines

These disadvantages led the designers of some prior operating systems, for example, IBSYS/IBJOB, to circumvent the inclusion of input/output routines in assembled programs by providing a set of input/output "packages" that could be mobilized at program-loading time.[1] Designed to operate interpretively, these optional packages permitted a source program to be less specific about devices and operating modes; moreover, they permitted change in the input/output environment without program reassembly. On the other hand, interpretive execution tends to reduce the efficiency of packages and limit the feasible degree of system complexity and expandability.

generated I/O routines

Faced with unprecedented diversity in storage devices and potential applications in addition to the complexities of multitask operation, the OS/360 designers have carried the IBSYS/IBJOB philosophy further, but with a number of significant tactical differences. Data-management control facilities are not obtained at program-loading time; instead, they are tailored to current needs during the very course of program execution (wherever the programmer uses an OPEN macroinstruction). The data-access routines are reenterable, and different tasks with similar needs may share the same routines. Because routines do not act interpretively, they can be highly specialized as well as economical of space. A program chooses one of the available access methods and requests input/output operations using appropriate macroinstructions. Device types, buffering techniques, channel affinities, and data attributes are later specified via data-definition statements in the job stream. In fact, the OS/360 job stream permits final specification of nearly every data or processing attribute that does not require re-resolution of main-storage addresses in an assembled program. These attributes include blocking factors, buffering techniques, error checks, number of buffers, and the like.

System definitions

An operating system deals with many different categories of information. Examples from a number of categories are a source program, an assembled program, a set of related subroutines, a message queue, a statistical table, and an accounting file. Each of these examples consists of a collection of related data items. In the OS/360 context, such a collection is known as a *data set.* In the operational sense, a data set is defined by a data-set label that contains a name, boundaries in physical storage, and other parameters descriptive of the data set. The data-set label is normally stored with the data set itself.

volume

A standard unit of auxiliary storage is called a *volume.* Each direct-access volume (disk pack, data cell, drum, or disk area served by one access mechanism) is identified by a volume label. This label always contains a volume serial number; in the case of direct-access devices, it also includes the location of a *volume table of contents* (VTOC) that contains the labels of each data set stored in the corresponding volume. A label to describe the VTOC and an additional label to account for unused space are created. Before being used in the system, each direct-access volume is initialized by a utility program that generates the volume label and, for direct-access devices, constructs the table of contents. This table is designed to hold labels for the data sets to be written on the volume.

Given the volume serial number and data-set name, the control program can obtain enough information from the label to access the data set itself.

A job step can place a data set in direct-access storage via a data definition (DD) statement that requests space, specifies the kind of volume, and gives the data-set name. At job-step initiation, the system allocates space and creates a label for each area requested by a DD statement. Finally, during job-step execution, the label is completed and updated via OPEN and CLOSE macro-instructions.

Each reel of magnetic tape is considered a volume. In view of the serial properties of tape, the method used for identifying volumes and data sets departs somewhat from the method used for direct-access devices. The standard procedure still employs volume labels and data-set labels; but each data-set label exists in two parts: a header label preceding its data set, and a trailer label that follows it. The location of a data set in a tape volume is represented by a sequence number that facilitates tape searching.

Although the system includes a generalized labeling procedure, it permits a user to employ his own tape-label conventions and label-checking routines if so desired. Unlabeled tapes may be used, in which case the responsibility for mounting the right volumes reverts to the operator.

To free the programmer of the need to maintain inventories of his data sets, the system provides a *data-set catalog.* Held in

المنارة للاستشارات

direct-access storage, this catalog consists of a tree-organized set of indexes. To best serve the needs of individual installations, the organization of the tree structure is left to the user. Each qualifier of a data-set name corresponds to an additional level in the tree. For example, the data set PAYROLL.MASTER.SEGMENT1 is found by searching a master index for PAYROLL, a second-level index for MASTER, and a third-level index for SEGMENT1. Stored with the latter argument are entries that identify the volume containing the data set and the device type; in the case of serial-access devices, a sequence number is also stored.

catalog

A volume containing all or part of the catalog is called a *control volume*. Normally, the operating system resides in a control volume known as the *system residence volume*. The use of a distinctive control volume for a group of related data sets makes it convenient to move the portion of the catalog that is relevant to the group. This is particularly important in planning for the possibility that groups of data sets may be moved from one computer to another.

control volume

A data-set search starts in a system residence volume and continues, level by level, until a volume identification number is obtained. If the required volume is not already mounted, a message is issued to the operator. Then, if the data set is stored in a direct-access device, the search for the data-set location resumes with the volume label of the indicated volume, continues in the volume table of contents, and proceeds from there to the data set's starting location. On the other hand, if the data set is held on a serial-access device, the search continues using a sequence number as an argument.

To simplify DD (data definition) statements for recurrent updating jobs, data sets related by name and cataloging sequence can be identified as a *generation group*. In applications that regularly use the n most prior generations of a group to produce a new generation, the new generation may be named (and later referred to) relative to the most recent generation. Thus, the DD statement need not be changed from run to run. When the index for the generation group is established, the programmer specifies n. As each new generation is cataloged, the oldest generation is deleted from the catalog. Provision is also made for the special case in which n varies systematically, starting at 1 and increasing by 1 until it reaches a user-specified number N, at which time it starts over at 1.

generation group

To safeguard sensitive data, any data set may be flagged in its label as "protected." This protection flag is tested as a consequence of the OPEN instruction; if the flag is on, a correct *password* must be entered from the console. The data set name and appended password serve as an argument for searching a control table. The OPEN routine is not permitted to continue unless a matching entry is found in the table.

password

Because the control table has its own security flag and *master password*, it can be reached only by the control program and those programmers privileged to know the master password.

record and block

In discussing the internal structure and disposition of a data set, it is necessary to distinguish between the *record*, an application-defined entity, and the *block*, which has hardware-defined boundaries and is governed by operational considerations. Let b denote block length (in bytes) and B a maximum block length. Although OS/360 requires that B be specified for each given data set, conventions permit three block-format categories: unspecified, variable, and fixed. The first category requires that $b \leq B$ for all blocks. The second is similar to the first, except that each b is stored in a count field at the beginning of its block. The third category dictates that all blocks be of length B.

A fixed or variable block may contain multiple records. A fixed block contains records of fixed size. In the variable block, records may vary in size, and each record is preceded by a field that records its size. For storage devices that employ interblock gaps, it is well known that record blocking can increase effective data rates, conserve storage, and reduce the needed number of input/output operations in processing a data set. For data sets of unspecified block format, the system makes no distinction between block and record; any applicable blocking and deblocking must be done by the user's program. The unspecified format is intended for use with peripheral equipment, such as transmission devices, address label printers, and the like.

buffer

A *buffer* is a main storage area allocated to the input/output function. The portion of a buffer that holds one record is called a *buffer segment*. A group of buffers in an area of storage formatted by the system is called a *buffer pool*; a data set associated with a buffer pool is assigned buffers from the pool. Unless a programmer assigns a buffer pool to a data set, OS/360 does so; unless buffer size is specified by the programmer, OS/360 sets the size to B.

In processing records from magnetic tape, it is customary to read and process records from one or more data sets, and to create one or more new data sets. A number of buffering considerations come into play. It may suffice to process a record within a buffer; it may be preferable to move the input record to a work area and the updated record from the work area into an output buffer; other possibilities may suggest themselves. Moreover, in processing records from direct-access storage, the same data set may be accessed for input and output.

transmittal modes

To allow flexibility in buffer usage, the OS/360 record-transfer routines invoked by the GET and PUT macroinstructions permit three *transmittal modes*. In the "move" mode, each record is moved from an input buffer to a work area and finally to an output buffer. In the "locate" mode, a record is never actually moved, but a pointer to the record's buffer segment is made available to the application program. In the "substitute" mode, which also uses pointers, the application program provides a work area equal in size to a record, and the buffer segment and work area effectively change roles.

To supplement the transmittal modes in special cases, three

المنارة للاستشارات

methods of buffer allocation are defined. *Simple buffering*, the most general method, allocates one or more buffers to each data set. *Exchange buffering*, used with fixed-length records, utilizes data-chaining facilities to effect record gather and record scatter operations. Buffer segments from an input data set are exchanged with buffer segments of an output data set or work area. Not only can each buffer segment be treated, in turn, as an input area, work area, and output area, but chaining allows noncontiguous segments to simulate a block. Exchange buffering is particularly useful in updating sequential files, merging, and array manipulation.

buffer allocation

Chained-segment buffering is designed for messages of variable size. Segments are established dynamically, with chaining being used to relate physically separate segments. This method is designed to circumvent the need for a static allocation of space to a remote terminal: of the many terminals that can be present in a system, only a fraction are ordinarily in use at a given time.

Access principles

To fall within the OS/360 data-management framework, a data set must belong to one of five organizational categories. As will be seen, the classification is based mainly on search considerations.

Data sets consisting of records held in serial-access storage media (such as magnetic tapes, paper tapes, card decks, or printed listings) are said to possess *sequential* organization. If so desired, a data set held in a direct-access device may also be organized sequentially.

data-set categories

Three of the five categories apply solely to direct-access devices. The *indexed sequential* organization stores records in sequence on a key (record-contained identifier). Because the system maintains an index table that contains the locations of selected records in the sequence, records can be accessed randomly as well as sequentially. A *direct* organization is similar, but dispenses with the index table and leaves record addressing entirely up to the programmer. A *partitioned* organization divides a sequentially organized data set into *members*; member names and locations are held in a directory for the data set. A member consists simply of one or more blocks. Included primarily for data sets consisting of programs or subroutines, this organization is suitable for any data set of randomly retrieved sequences of blocks.

A *telecommunications* organization is provided for queues of messages received from or enroute to remote on-line terminals. Provision is made for forming message queues and for retrieving messages from queues. Queues may be held in direct-access storage as well as in main storage.

A broad distinction is made between two classes of data-management languages. Designed for programming simplicity, the *queued access* languages apply only to organizations with sequential properties. The programmer typically uses the macro-

language categories

Table 1

Organization	Language category Queued	Basic
Sequential	QSAM	BSAM
Indexed Sequential	QISAM	BISAM
Direct		BDAM
Partitioned		BPAM
Telecommunication	QTAM	BTAM

instructions GET and PUT to retrieve and store records, and buffers are managed automatically by the system. On the other hand, the *basic access* languages provide for automatic device control, but not for automatic buffering and blocking. Typically, the READ and WRITE macroinstructions are used to retrieve and store blocks of data. Because the programmer retains control over device-dependent operations (such as card reader or punch-stacker selection, tape backspacing, and the like), he may use any desired searching, buffering, or blocking methods.

access methods

Of the ten possible combinations of data-set and language categories, eight are recognized by the system as *access methods*. These eight methods bear the mnemonic names given in Table 1: QSAM denotes "queued sequential access method," and so on. For each access method, a vocabulary of suitable macroinstructions is provided.

To employ a given access method, a programmer resorts to the vocabulary of macroinstructions provided for that method. Vocabularies for six of the methods are summarized in Table 2. Although six macroinstructions are common to all of these methods, the parameters to be specified in a macroinstruction may vary from method to method. If so desired for specialized applications, a programmer can circumvent the system-supported access methods and employ the *execute channel program* (EXCP) macroinstruction in fashioning his own access method. In this case, he must prepare his own channel program (sequence of channel command words).

A few words on each vocabulary element of Table 2 help to clarify access principles. At assembly time, the DCB macroinstruction reserves space for a *data control block* and fills in control block fields that designate the desired access method, name a relevant DD statement, and select some of the possible options. The application programmer is expected to provide symbolic addresses of any applicable supplementary routines, as for example, special label-processing routines.

The programmer issues an OPEN macroinstruction for each data control block. At execution time, OPEN supplies information not declared in the DCB macroinstruction, selects access routines and establishes linkages, issues volume mounting messages to the operator, verifies labels, allocates buffer pools, and positions

للاستشارات

Table 2 Access-method vocabularies

Macro-instruction	QSAM	QISAM	BSAM	BPAM	BISAM	BDAM	*Macroinstruction function in brief*
DCB	•	•	•	•	•	•	Generate a data control block
OPEN	•	•	•	•	•	•	Open a data control block
CLOSE	•	•	•	•	•	•	Close a data control block
BUILD	•	•	•	•	•	•	Structure named area as a buffer pool
GETPOOL	•	•	•	•	•	•	Allocate space to and format buffer pool
FREEPOOL	•	•	•	•	•	•	Liberate buffer-pool space
GET	•	•					Obtain a record from an input data set
PUT	•	•					Include a record in an output data set
PUTX	•	•					Include an input record in an output data set
RELSE	•	•					Force end of input block
TRUNC	•						Force end of output block
FEOV	•		•				Force end of volume
CNTRL	•		•				Control reader or printer operation
PRTOV	•		•				Test for printer carriage overflow
SETL		•					Set lower limit for scan
ESETL		•					Postpone fetching during scan
CHECK			•	•			Wait for I/O completion and verify proper operation
NOTE			•	•			Note where a block is read or written
POINT			•	•			Point to a designated block
FIND				•			Obtain the address of a named member
BLDL				•			Build a special directory in main store
STOW				•			Update the directory
RELEX					•	•	Release exclusive control of a block
FREEDBUF					•	•	Free dynamically obtained buffer
GETBUF			•	•	•	•	Assign a buffer from the pool
FREEBUF			•	•	•	•	Return a buffer to the pool
WAIT			•	•	•	•	Wait for I/O completion
READ			•	•	•	•	Read a block
WRITE			•	•	•	•	Write a block

volumes. The programmer may free a data control block and return associated buffers to the pool by the CLOSE macroinstruction; if he omits CLOSE, the system performs the corresponding functions at task termination.

The programmer can request the system to allocate and format a buffer pool at execution time by issuing a GETPOOL macroinstruction, which specifies the address of the data control block, the buffer length, and the desired number of buffers. When a pool area is no longer needed, it can be returned to the system by FREEPOOL.

Where the programmer's knowledge permits him to allocate space more wisely than the control program, he may choose to designate the area to be set aside for a buffer pool. The area may,

for example, supplant a subroutine that is no longer needed. By issuing a BUILD macroinstruction, he can request the system to employ the reserved area as a buffer pool, the details being similar to GETPOOL. With subsequent BUILD's, moreover, he can restructure the area again and again.

QSAM

QSAM corresponds closely to the schemes most favored in previous input/output systems. QSAM yields a great deal of service to the programmer for a minimum investment in programming effort. Retrieval is afforded by GET, which supplies one record to the program; disposition of an output record is afforded by PUT or PUTX. PUT transfers a record from a work area or buffer to a data set; PUTX transfers a record from one data set to another. In consequence, PUT involves one data control block, whereas PUTX involves two.

To aid the programmer in creating short blocks and in disposing of a block before all records therein have been processed, two macroinstructions permit intervention in buffer control. RELSE requests the system to release the remaining buffer segments in an input buffer, i.e., to view the buffer as empty. Analogously, TRUNC asks the system to view an output buffer as full, and to go on to another buffer.

FEOV requests the system to force an end-of-volume status for a designated data set, and thereupon to undertake the normal volume-switching procedure. CNTRL provides for specialized card-reader, printer, or tape control functions.

QISAM

The QISAM scheme is closely akin to QSAM, but the macroinstructions provide the additional functions required of indexed sequential data sets and direct-access devices. Records are arranged in logical sequence on the key, a field that is part of each record. Record keys are related to physical addresses by indexes. For a record with a given data key, a *cylinder index* yields cylinder address, and a *track index* yields track-within-cylinder address. To facilitate in-channel searches, the key of the last record in each block is placed in a hardware-defined control field.

In the initial creation of a data set, PUT's are used in the "load" mode to store records and generate indexes. Successive GET's in the "scan" mode retrieve records sequentially; SETL (set lower limit) may be issued to designate the first record obtained. Unless a SETL is issued, retrieval starts from the first record of the data set. In scan mode, PUTX may follow a GET to return an updated record to the data set. ESETL (end of scan) halts any anticipatory buffering on the part of the system until issuance of a subsequent GET.

BISAM

BISAM applies to the same sequential data organization as QISAM, but selective reading and writing is permitted through the READ and WRITE macroinstructions. Using BISAM, new records can be inserted without destroying sequence. If an insertion does not fit into the intended track, the system moves one or more records from the track to an overflow area and then reflects this overflow status in the appropriate indexes. (The existence of over-

المنارة للاستشارات

flows does not alter the ability of QISAM to scan records in logical sequence.)

To permit other operations to be synchronized with BISAM input/output operations, a WAIT macroinstruction supplements READ and WRITE. (Because WAIT serves a general function in synchronizing tasks, it is discussed in Part II.)

In a multitask environment, it is possible that one task may want to use or update a record while the record is being updated by another task. To forestall confusion in the order that updating operations are accomplished, READ can request exclusive control of the record during updating. For a record being updated in place, WRITE releases exclusive control. If the record is not updated in place, the RELEX macroinstruction can be used to release control.

Because record insertions may lead to overflows, and overflows tend to reduce input/output performance, the system is designed to provide statistics that can help a programmer in determining when data-set reorganization is desirable. Held in the data control block are the number of unused tracks in an independent overflow area and, optionally, the number of full cylinder areas, as well as the number of accesses to overflow records not appearing at the head of overflow chains. Reorganization can be accomplished via the QISAM load mode, using the existing data set as input.

As implied by the above discussion, QISAM and BISAM complement one another and may be used together where the user needs to access a data set randomly as well as sequentially. For the sake of convenience, a data control block for an indexed sequential data set can be opened jointly for QISAM and BISAM.

BSAM

BSAM assumes a sequentially organized data set and deals with blocks rather than records. A block is called into a specified buffer by READ. Unless program execution is deliberately suspended during the retrieval period by a CHECK macroinstruction, the program may continue during reading. Similarly, after an output operation is initiated, CHECK can be used to postpone further processing until the operation is completed.[2] Following a READ or WRITE, the macroinstruction NOTE saves the applicable block address in a standard register; subsequently, the preserved address may be helpful in logically repositioning the volume by POINT.

BDAM

Of the access methods for direct-access devices, BDAM offers the greatest variety of access possibilities. Using WRITE and READ, the programmer can store or retrieve a block from a data set by specifying a track address and block number. Optionally, he may specify a number relative to the data set itself, either (1) a relative track number at which a search should start for a given key or (2) a relative block number. The relative numbers, which help to isolate application programs from device peculiarities, are converted to actual track addresses and block numbers by the system. GETBUF and FREEBUF are the means by which buffers can be explicitly requested and released. A dynamic buffer option, requested in the DCB macroinstruction, enables the programmer to obtain automatic buffer management (BUILD and GETPOOL are

not used in conjunction with the option). The FREEDBUF macroinstruction permits release of a buffer under the dynamic option.

BPAM

BPAM is designed for storing and retrieving members of a partitioned data set held on a direct-access device. Associated with the data set is a *directory* that relates member name to track address. To prepare for access, a FIND performs the directory search. A located member can be retrieved using one or more READ's, as required by the number of blocks in the member. New members can be placed by one or more WRITE's, followed by a STOW that enters the member's name and location in the directory. CHECK again serves to synchronize the program with data-transmission operations.

A summary of the main characteristics of the eight access-methods appear in Table 3.

Control elements and system operation

data control block

With general definitions and access methods in mind, we turn to the internal structure of OS/360 as it pertains to data management.

Associated with each data set of a problem program is a data control block (DCB), which must be opened before any data transfer takes place. However, some data sets, e.g., the catalog data set, are opened automatically by the control program, and may be indirectly referred to or used in a problem program without additional opening or closing. Data-access macroinstructions, such as GET and PUT, logically refer to a data set, but actual reference is always via a data control block.

The data control block is generated and partially filled when the DCB macroinstruction is encountered at compilation time. The routine called at execution time by OPEN completes the data control block with information gained principally from a job-stream DD statement or cataloged procedure. For input data sets, a final source of such information is the data-set label. In the case of an output data set where the label has yet to be created, the final source can be the label of another data set or another DD statement.

In addition to completing the data control block, the OPEN routine ensures that needed access routines are loaded and address relations are completed. The routine prepares buffer areas and generates channel command word lists; it initializes data sets by reading or writing labels and performs a number of other housekeeping operations.

The selection of access routines is governed by choices in data organization, buffering technique, access language, input/output unit characteristics, and other factors. The selection is relayed to the supervisor, which allocates main storage space and performs the loading.

In operation, some access routines are treated as part of the user's program and are entered directly rather than through a supervisor-call interruption. These routines block and deblock

المنارة للاستشارات

Table 3 Access-method summary

Characteristic	QSAM	QTAM	QISAM	BSAM	BTAM	BPAM	BISAM	BDAM
Data set organization	Sequential or member of partitioned	Telecom	Indexed sequential	Sequential	Telecom	Partitioned	Indexed Sequential	Direct
Basic element of data set	Record	Message or message segment	Record	Record	Message	Member	Record	Record
Basic concern of access method	Record	Message	Record	Block	Block	Block	Block	Block
Primary input and output macroinstructions	GET PUT PUTX	GET PUT RETRIEVE	Scan SETL Scan GET Load PUT Scan PUTX	READ WRITE	READ WRITE	FIND READ WRITE STOW	READ WRITE	READ WRITE
Buffer pool acquisition	BUILD GETPOOL Automatic	BUFFER	BUILD GETPOOL Automatic	BUILD GETPOOL Automatic	BUILD GETPOOL Automatic	BUILD GETPOOL Automatic	BUILD GETPOOL Automatic	BUILD GETPOOL Automatic
Buffer management for a data set	Automatic (simple or exchange)	Automatic (chained-segment)	Automatic (simple)	GETBUF FREEBUF	GETBUF FREEBUF	GETBUF FREEBUF	Dynamic FREEDBUF GETBUF FREEBUF	Dynamic FREEDBUF GETBUF FREEBUF
Transmittal mode	Move Locate Substitute	Move	Move Locate					
Synchronization	Automatic	Automatic	Automatic	CHECK WAIT	WAIT	CHECK WAIT	WAIT	WAIT
Record/block format*	F, V record	U message	F, V record	F, V record	U block	F, V, U block	F, V record	F, V, U block
Special provisions for data-set search		Sequence number; relative addressing	Cylinder & track indexes			Directory of members	Cylinder & track indexes	Can use relative record or track number

* F, V, and U denote fixed, variable, and unspecified lengths.

المنارة للاستشارات

records, control the buffers, and call the input/output supervisor when a request for data input or output is needed. Other routines, treated as part of the I/O supervisor and therefore executed in the privileged mode, perform error checks, prepare user-oriented completion codes, post interruptions, and bridge discontinuities in the storage areas assigned to a data set.

I/O supervisor

The input/output supervisor performs all actual device control (as it must if contending programs are not to conflict in device usage); it accepts input/output requests, queues the requests if necessary, and issues instructions when a path to the desired input/output unit becomes available. The I/O supervisor also ensures that input/output requests do not exceed the storage areas allocated to a data set. The completion of each input/output operation is posted and, where necessary, standard input/output error-recovery procedures are performed. EXCP, the execute channel program macroinstruction, is employed in all communication between access routines and the input/output supervisor.

To portray the mechanics of data management, let us consider one job step and the data-management operations that support a READ macroinstruction for a cataloged data set in the BSAM context.

To begin with, we observe the state of the system just before the job is introduced; of interest at this point are the devices, control blocks, programs, and catalog elements that exist prior to job entry. Next to be considered are the data-management activities involved in DD-statement processing, and in establishment by the job scheduler of a task for the given job step. Third, we consider the activities governed by the OPEN macroinstruction; these activities tailor the system to the requirements of the job step. Finally, operation of the READ macroinstruction is considered, with special attention to the use of the EXCP macroinstruction. Essential to the four stages of the discussion are four cumulative displays. Frequent reference to numbered points within the figures is made by means of parenthetical superscripts in the text. The description refers more often to the objects generated and manipulated by the system than to the functional programs that implement the system.

catalog organization

The basic aspects of catalog implementation become apparent when we consider the manner in which the system finds a volume containing a cataloged data set. Recall that each direct-access volume contains a volume label that locates its VTOC (volume table of contents) and that the VTOC contains a data-set label for each volume-contained data set. Identified by data-set name, the data-set label holds attributes (such as record length) and specifies the location in the volume of the data set.

Search for a data set begins (see Figure 1) in the VTOC of the system residence volume, where a data-set label identifying the portion of the catalog in this volume[(1)] appears. This part of the catalog is itself organized as a partitioned data set whose directory is the highest level (most significant) index of the catalog. For

المنارة للاستشارات

Figure 1 Control elements: before job entry

VTOC SYS RES VOL
1
CATALOG DIRECTORY SYS RES VOL
* 3
VOLUME LABEL CONTROL VOL
4
VTOC CONTROL VOL
5
CATALOG DIRECTORY CONTROL VOL
* 6
* 2
CATALOG INDEX SYS RES OR CONTROL VOL
* 7 *
CATALOG INDEX SYS RES OR CONTROL VOL
* 8
VOLUME LABEL OBJ VOL
9
VTOC OBJ VOL
* 10
DATA OBJ VOL

DEVICE

CONTROL UNIT

CHANNEL

UCB
* 11
CODE DEVICE DEPENDENT
12
LOGICAL CHANNEL QUEUE

LEGEND

- MAIN STORAGE ADDRESS
- RELATIVE MAIN STORAGE ADDRESS
- DASD TRACK OR BLOCK ADDRESS
- RELATIVE DASD TRACK OR BLOCK ADDRESS
- VOLUME IDENTIFICATION NUMBER
- INTERRUPTION
- DATA SET NAME
- DATA DEFINITION STATEMENT NAME
- OBJECT DATA FLOW
- CONTROL DATA FLOW
- o INDICATES THAT ITEM POINTED TO IS ONE OF A CHAIN OF SIMILAR ITEMS
- * INDICATES THAT THE SOURCE OF A POINTER IS A TABLE WHICH IDENTIFIES SIMILAR ITEMS

data sets cataloged on the system residence volume, entries in this directory contain the addresses of lower-level indexes;(2) for data sets cataloged on other control volumes,(3) directory entries contain the appropriate volume identification numbers.

Assume for the moment that the search is for a data set cataloged on control volume V and that V is not the system residence volume. In this case, the volume label of V contains the location of V's VTOC.(4) (Volume label and VTOC are recorded separately to allow for device peculiarities.) One of the data-set labels in this VTOC identifies the part of the catalog on V;(5) just as in the case of the residence volume, this part is organized as a partitioned data set. Inasmuch as the directory of this partitioned data set is the subset of the highest-level index governing that part of the catalog recorded on V, directory entries contain the addresses of the next-level indexes on V.(6) It should be added that all index levels needed to catalog a data set appear on a single control volume; the part of the catalog on any given control volume is known to other control volumes, because the directory entries of the given control volume appear in the directories of the others.

Each index level below the directory(7) is used to resolve one qualification in the name of a data set. For example, were the name of a data set A.B.C, a directory entry A would locate an index containing an entry B, which in turn would locate an index containing the entry C. This last entry identifies the volume(8) that holds the data set named A.B.C.[3]

unit control block

During the system generation process, one *unit control block* (UCB) is created for each I/O device attached to the system (each tape drive, disk drive, drum, card reader/punch, etc). Each UCB contains device-status information, the relevant device address or addresses, the locations of the input/output supervisor subroutines(11) that treat device peculiarities (such as start-I/O, queue-manipulation, and error routines), and the location of the *logical channel queue*(12) used with the device.[4]

DD-statement processing

The principal purpose of the DD statement (Figure 2) is to supply the (variable) name of a data set to be located via the catalog,(13) and to relate the data set to the (constant) name of the DD statement. However, a great amount of additional information may be supplied if the user desires. This information may include: the device type together with a list of volume identification numbers which serve to locate the data set without recourse to the catalog;(14,15) label information used to create new labels; attributes that determine the nature of the data set created or processed; and processing options that modify the operation of the program. After being encoded by the job scheduler, most of this information is included in a *job file control block* (JFCB)(16) that is used in lieu of the original DD statement.

As was suggested above, a data set can be located either by an explicit list of volume identification numbers and an indication of the device type (if this information is given on the DD statement), or by data-set name alone. In the latter case, a list of volume

المنارة للاستشارات

Figure 2 Control elements: job scheduling—hexagonal blocks denote elements of first concern at time job is scheduled

VTOC SYS RES VOL
DD STATEMENT
CATALOG DIRECTORY SYS RES VOL
TCB
VOLUME LABEL CONTROL VOL
TIOT
VTOC CONTROL VOL
JFCB SYS RES VOL
CATALOG DIRECTORY CONTROL VOL
UCB
CODE DEVICE DEPENDENT
CATALOG INDEX SYS RES OR CONTROL VOL
LOGICAL CHANNEL QUEUE
CATALOG INDEX SYS RES OR CONTROL VOL
VOLUME LABEL OBJ VOL
VTOC OBJ VOL
DATA OBJ VOL
DEVICE
CONTROL UNIT
CHANNEL

LEGEND

MAIN STORAGE ADDRESS
RELATIVE MAIN STORAGE ADDRESS
DASD TRACK OR BLOCK ADDRESS
RELATIVE DASD TRACK OR BLOCK ADDRESS
VOLUME IDENTIFICATION NUMBER
INTERRUPTION
DATA SET NAME
DATA DEFINITION STATEMENT NAME
OBJECT DATA FLOW
CONTROL DATA FLOW
o INDICATES THAT ITEM POINTED TO IS ONE OF A CHAIN OF SIMILAR ITEMS
* INDICATES THAT THE SOURCE OF A POINTER IS A TABLE WHICH IDENTIFIES SIMILAR ITEMS

identification numbers is extracted from the catalog and placed in the JFCB.[17]

Prior to establishing a task for the job step, the job scheduler assigns devices to the step. To represent this assignment, the job scheduler constructs a *task input/output table* (TIOT). An entry is made in this table for each DD statement supplied by the user; each entry relates a DD-statement name to the location of the corresponding JFCB[18] and the unit or units assigned to the data set.[19] The assignment of a specific device derives from the specification of device type supplied through the DD statement[20] or the catalog,[21] together with a table of available units maintained by the job scheduler.

The job scheduler then assures that all volumes initially required by the step are mounted. As each volume is mounted, its volume label is read; the volume identification number and the location of its VTOC are placed in the corresponding UCB for future reference.[22] Finally, the job scheduler "attaches" a task for the step. In the process, the supervisor constructs a *task control block* (TCB). The TCB is used by the supervisor as an area in which to store the general registers and program status word of a task at a point of interruption; it contains the address of the TIOT.[23]

OPEN

Execution of the OPEN macroinstruction (Figure 3) identifies one or more data control blocks (DCB's) to be initialized:[24] since an SVC interruption results, the TCB of the calling task[25] is also identified. The name of the DD statement, contained in the DCB, is used to locate the entry in the TIOT corresponding to the data set to be processed.[23,26] The related JFCB is then retrieved.[18]

After assuring that the required volumes are mounted,[19] the open subroutines read the data-set label(s) and place in the JFCB all data-set attributes that were not specified (or overridden) by the DD statement.[27] At this point, the DCB and JFCB comprise a complete specification of the attributes of the data set and the access method to be used. Next, data-set attributes and processing options not specified by the DCB macroinstruction are passed from the JFCB to the DCB.[28]

The system then constructs a *data extent block* (DEB), logically a protected extension of the DCB. This block contains a description of the *extent* (devices and track boundaries) of the data set,[29,30] flags which indicate the set of channel commands that may be used with the data set,[37] and a priority indicator.[31] The DEB is normally located via the DCB;[32] but in order to purge a failing task or close the DCB upon task termination, it may be located via the TCB.[33] If the data set is to be retrieved sequentially, the address of the first block of the data set is moved to the DCB.[34]

Next, the access-method routines are selected and loaded. The addresses of these routines are placed in the DCB.[35] If privileged interrupt-handling or error routines are required, they are loaded and their addresses recorded in the DEB.[36] Finally, the channel programs which will later be used to access the data set are generated. For each channel program, an *input/output block* (IOB) is

المنارة للاستشارات

Figure 3 Control elements: OPEN macroinstruction—oblate blocks denote elements of first concern during execution of OPEN macroinstruction

LEGEND

- MAIN STORAGE ADDRESS
- RELATIVE MAIN STORAGE ADDRESS
- DASD TRACK OR BLOCK ADDRESS
- RELATIVE DASD TRACK OR BLOCK ADDRESS
- VOLUME IDENTIFICATION NUMBER
- INTERRUPTION
- DATA SET NAME
- DATA DEFINITION STATEMENT NAME
- OBJECT DATA FLOW
- CONTROL DATA FLOW
- O INDICATES THAT ITEM POINTED TO IS ONE OF A CHAIN OF SIMILAR ITEMS
- * INDICATES THAT THE SOURCE OF A POINTER IS A TABLE WHICH IDENTIFIES SIMILAR ITEMS

created.[38] The IOB is the major interface between the problem program (or the access-method routines) and the I/O supervisor. It contains flags that govern the channel program, the location of the DCB,[59] the location of an event control block used with the channel program, the location of the channel program itself, the "seek address," and an area into which the I/O supervisor can move the channel status word at the completion of the channel program. IOB's are linked in a chain originating at the DCB.[39]

READ

The READ macroinstruction (see Figure 4) identifies a parameter list, called the *data event control block* (DECB),[40] that is prepared either by the user or the READ macroinstruction. This block contains the address of a buffer,[41] the length of the block to be read (or the length of the buffer), the address of the DCB associated with the data set,[43] an event control block, and the like. Buffer address and block or buffer length are obtained from the DCB if not supplied by the user.[44] Using an address previously placed in the DCB,[35] the READ macroinstruction branches to an access-method routine that assigns an IOB and a channel program to the DECB. Subsequently, the routine modifies the channel program to reflect the block length and the location of the buffer;[42] it then records the address of the DECB in the IOB.[45] In addition, the routine computes the track and block addresses of the next block and updates the IOB and channel program using the results.[42,46,47,48] The access method routine then issues the EXCP macroinstruction.

EXCP

The EXCP macroinstruction causes an SVC interruption[49] that calls the I/O supervisor and passes to it the addresses of the IOB and, indirectly, the DCB.[59] Using the DCB, the address of the DEB is obtained and verified.[32] Next, assuming that other requests for the device are pending, the IOB is placed in a seek queue to await the availability of the access mechanism. Queues maintained by the IOS take the form of chains of *request queue elements* (RQE's) which identify the IOB's in queues.[51] An RQE contains a priority byte obtained from the DEB,[52] the address of the DEB,[53] and the address of the TCB of the requesting task[54] (used to purge the system of the IOB's upon task termination). Seek queues originate from UCB's,[50] and are (optionally) maintained in ascending sequence by cylinder address to reduce average seek time.

When, as a result of the completion of other requests, the access mechanism becomes available to the current IOB, a seek operation is initiated using the track address in the IOB. Just prior to this, the track address is verified (using the contents of the DEB) to ensure that the seek address lies within the extent of the data set. Assuming that the seek operation was not immediately completed, seek commands to other devices are issued; the channel is then used for other operations if possible. At the completion of the relevant seek operation,[56] the RQE is removed from the top of the seek queue and placed in the appropriate logical channel queue[55] in priority sequence. For the performance of all of these functions,

المنارة للاستشارات

Figure 4 Control elements: READ and EXCP macroinstructions—elliptical blocks denote elements of first concern during execution of READ or EXCP macroinstruction

VTOC SYS RES VOL
CATALOG DIRECTORY SYS RES VOL
VOLUME LABEL CONTROL VOL
VTOC CONTROL VOL
CATALOG DIRECTORY CONTROL VOL
CATALOG INDEX SYS RES OR CONTROL VOL
CATALOG INDEX SYS RES or CONTROL VOL
VOLUME LABEL OBJ VOL
VTOC OBJ VOL
DATA OBJ VOL
DD STATEMENT
OPEN MACRO
READ MACRO
TCB
TIOT
JFCB SYS RES VOL
DECB
DCB
DEB
UCB
CODE NON PRIV ACC METH
CODE PRIV ACC METH
CODE DEVICE DEPENDENT
LOGICAL CHANNEL QUEUE
RQE
IOB
CHANNEL PROGRAM
DEVICE
CONTROL UNIT
CHANNEL
BUFFER

LEGEND

- MAIN STORAGE ADDRESS
- RELATIVE MAIN STORAGE ADDRESS
- DASD TRACK OR BLOCK ADDRESS
- RELATIVE DASD TRACK OR BLOCK ADDRESS
- VOLUME IDENTIFICATION NUMBER
- INTERRUPTION
- DATA SET NAME
- DATA DEFINITION STATEMENT NAME
- OBJECT DATA FLOW
- CONTROL DATA FLOW
- o INDICATES THAT ITEM POINTED TO IS ONE OF A CHAIN OF SIMILAR ITEMS
- * INDICATES THAT THE SOURCE OF A POINTER IS A TABLE WHICH IDENTIFIES SIMILAR ITEMS

device-dependent routines addressed by the UCB[11] are executed by the I/O supervisor.

When the IOB reaches the top of the logical channel queue and a related channel is free, the channel program associated with the IOB is logically prefixed with a short supervisory channel program and the result executed. The control unit is initialized by the supervisory channel program to inhibit the channel program from executing commands that might destroy information outside of the extent of the data set, leave the channel and control unit unused for significant periods, or attempt to write in a data set that is to be used in a read-only manner.[5] When the channel program finishes,[56] its completion is posted in the event control block within the DECB.[45]

At any time after issuing a READ macroinstruction, the program may issue a WAIT or CHECK macroinstruction which refers to the same DECB as the READ macroinstruction. Either of these macroinstructions suspends the task[57,58] until the READ operation has been completed, i.e., until the I/O supervisor posts the completion of the operation in the DECB.

Although the foregoing discussion applies specifically to the READ macroinstruction in the BSAM context and to the use of a direct-access device, the first three displays (Figures 1, 2, and 3) are applicable to other operations as well. In fact, the discussion introduces most of the control elements that bear on data-management operations in any context.

Summary

The design of OS/360 assures that data sets of all kinds can be systematically identified, stored, retrieved, and surveyed. Versatility is served by a variety of techniques for structuring data sets, catalogs, buffers, and data transfers. In the interest of operational adaptability, the system tailors itself to actual needs on a dynamic basis. For programming efficiency, source programs may be device-independent to a novel degree.

CITED REFERENCE AND FOOTNOTES

1. A. S. Noble, Jr., "Design of an integrated programming and operating system, Part I, system considerations and the monitor," *IBM Systems Journal* **2**, 153–161 (June 1963).
2. Although the CHECK macroinstruction includes the effect of the WAIT macroinstruction, the latter may also be used prior to CHECK.
3. Ordinarily, the results of a catalog search include the device type, the identification number of the desired volume, and label verification information. If the data set is a generation of a generation group (a case not considered in the main discussion), the results are the location of an index of generations and an archetype data-set label.
4. Generally, "logical channel" and physical channel are indistinguishable. The logical channel is taken to be the set of physical channels by which a device is accessible. All devices (independent of their type) that share exactly the same set of physical channels are associated with the same logical channel queue. For example, a set of tape drives attached to physical channels 1 and 2 would share a logical channel distinct from that of a printer attached only to physical channel 1.

المنارة للاستشارات

5. In general, the control unit is initialized to inhibit seek operations that move the access mechanism. More stringent restrictions are placed on channel programs that actually refer to cylinders shared by two or more data sets. This is not to say that inter-cylinder seek operations are disallowed; rather, the I/O supervisor verifies that these operations refer to areas within the extent of the data set. During inter-cylinder seek operations, the channel and control unit are freed for other uses.

المنارة للاستشارات

Peter Chen

The Entity Relationship Model – Toward a Unified View of Data

ACM Transactions on Database Systems, Vol. 1 (1), 1976

المنارة للاستشارات

The Entity-Relationship Model—Toward a Unified View of Data

PETER PIN-SHAN CHEN

Massachusetts Institute of Technology

A data model, called the entity-relationship model, is proposed. This model incorporates some of the important semantic information about the real world. A special diagrammatic technique is introduced as a tool for database design. An example of database design and description using the model and the diagrammatic technique is given. Some implications for data integrity, information retrieval, and data manipulation are discussed.

The entity-relationship model can be used as a basis for unification of different views of data: the network model, the relational model, and the entity set model. Semantic ambiguities in these models are analyzed. Possible ways to derive their views of data from the entity-relationship model are presented.

Key Words and Phrases: database design, logical view of data, semantics of data, data models, entity-relationship model, relational model, Data Base Task Group, network model, entity set model, data definition and manipulation, data integrity and consistency

CR Categories: 3.50, 3.70, 4.33, 4.34

1. INTRODUCTION

The logical view of data has been an important issue in recent years. Three major data models have been proposed: the network model [2, 3, 7], the relational model [8], and the entity set model [25]. These models have their own strengths and weaknesses. The network model provides a more natural view of data by separating entities and relationships (to a certain extent), but its capability to achieve data independence has been challenged [8]. The relational model is based on relational theory and can achieve a high degree of data independence, but it may lose some important semantic information about the real world [12, 15, 23]. The entity set model, which is based on set theory, also achieves a high degree of data independence, but its viewing of values such as "3" or "red" may not be natural to some people [25].

This paper presents the entity-relationship model, which has most of the advantages of the above three models. The entity-relationship model adopts the more natural view that the real world consists of entities and relationships. It

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

A version of this paper was presented at the International Conference on Very Large Data Bases, Framingham, Mass., Sept. 22–24, 1975.

Author's address: Center for Information System Research, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139.

المنارة للاستشارات

incorporates some of the important semantic information about the real world (other work in database semantics can be found in [1, 12, 15, 21, 23, and 29]). The model can achieve a high degree of data independence and is based on set theory and relation theory.

The entity-relationship model can be used as a basis for a unified view of data. Most work in the past has emphasized the difference between the network model and the relational model [22]. Recently, several attempts have been made to reduce the differences of the three data models [4, 19, 26, 30, 31]. This paper uses the entity-relationship model as a framework from which the three existing data models may be derived. The reader may view the entity-relationship model as a generalization or extension of existing models.

This paper is organized into three parts (Sections 2–4). Section 2 introduces the entity-relationship model using a framework of multilevel views of data. Section 3 describes the semantic information in the model and its implications for data description and data manipulation. A special diagrammatric technique, the entity-relationship diagram, is introduced as a tool for database design. Section 4 analyzes the network model, the relational model, and the entity set model, and describes how they may be derived from the entity-relationship model.

2. THE ENTITY-RELATIONSHIP MODEL

2.1 Multilevel Views of Data

In the study of a data model, we should identify the levels of logical views of data with which the model is concerned. Extending the framework developed in [18, 25], we can identify four levels of views of data (Figure 1):

(1) Information concerning entities and relationships which exist in our minds.

(2) Information structure—organization of information in which entities and relationships are represented by data.

(3) Access-path-independent data structure—the data structures which are not involved with search schemes, indexing schemes, etc.

(4) Access-path-dependent data structure.

In the following sections, we shall develop the entity-relationship model step by step for the first two levels. As we shall see later in the paper, the network model, as currently implemented, is mainly concerned with level 4; the relational model is mainly concerned with levels 3 and 2; the entity set model is mainly concerned with levels 1 and 2.

2.2 Information Concerning Entities and Relationships (Level 1)

At this level we consider entities and relationships. An *entity* is a "thing" which can be distinctly identified. A specific person, company, or event is an example of an entity. A *relationship* is an association among entities. For instance, "father-son" is a relationship between two "person" entities.[1]

[1] It is possible that some people may view something (e.g. marriage) as an entity while other people may view it as a relationship. We think that this is a decision which has to be made by the enterprise administrator [27]. He should define what are entities and what are relationships so that the distinction is suitable for his environment.

المنارة للاستشارات

LEVELS OF LOGICAL VIEWS

MODELS

ENTITY-RELATIONSHIP NETWORK RELATIONAL ENTITY-SET

LEVEL 1

INFORMATION CONCERNING ENTITIES AND RELATIONSHIPS

ENTITIES
ENTITY SETS
RELATIONSHIPS
RELATIONSHIP SETS
ATTRIBUTES
VALUES
VALUE SETS
ROLES

ENTITIES
RELATIONSHIPS
ATTRIBUTES
VALUES

ENTITIES
ENTITY SETS
ROLES

LEVEL 2

INFORMATION STRUCTURE

ENTITY/RELATIONSHIP RELATION

SIMILAR

3NF RELATIONS

ENTITY DESCRIPTION SETS

ENTITY-RELATIONSHIP DIAGRAM

DECOMPOSITION APPROACH

LEVEL 3

ACCESS-PATH-INDEPENDENT DATA STRUCTURE

TABLE

RELATIONS (TABLES)

LEVEL 4

ACCESS-PATH-DEPENDENT DATA STRUCTURE

RECORDS
DATA-STRUCTURE-SETS

DATA-STRUCTURE-DIAGRAM

Fig. 1. Analysis of data models using multiple levels of logical views

The database of an enterprise contains relevant information concerning entities and relationships in which the enterprise is interested. A complete description of an entity or relationship may not be recorded in the database of an enterprise. It is impossible (and, perhaps, unnecessary) to record every potentially available piece of information about entities and relationships. From now on, we shall consider only the entities and relationships (and the information concerning them) which are to enter into the design of a database.

2.2.1 Entity and Entity Set. Let e denote an entity which exists in our minds. Entities are classified into different *entity sets* such as EMPLOYEE, PROJECT, and DEPARTMENT. There is a predicate associated with each entity set to test whether an entity belongs to it. For example, if we know an entity is in the entity set EMPLOYEE, then we know that it has the properties common to the other entities in the entity set EMPLOYEE. Among these properties is the aforementioned test predicate. Let E_i denote entity sets. Note that entity sets may not be mutually disjoint. For example, an entity which belongs to the entity set MALE–PERSON also belongs to the entity set PERSON. In this case, MALE–PERSON is a subset of PERSON.

2.2.2 Relationship, Role, and Relationship Set. Consider associations among entities. A *relationship set*, R_i, is a mathematical relation [5] among n entities,

المنارة للاستشارات

each taken from an entity set:

$$\{[e_1, e_2, \ldots, e_n] \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\},$$

and each tuple of entities, $[e_1, e_2, \ldots, e_n]$, is a *relationship*. Note that the E_i in the above definition may not be distinct. For example, a "marriage" is a relationship between two entities in the entity set PERSON.

The *role* of an entity in a relationship is the function that it performs in the relationship. "Husband" and "wife" are roles. The ordering of entities in the definition of relationship (note that square brackets were used) can be dropped if roles of entities in the relationship are explicitly stated as follows: $(r_1/e_1, r_2/e_2, \ldots, r_n/e_n)$, where r_i is the role of e_i in the relationship.

2.2.3 Attribute, Value, and Value Set. The information about an entity or a relationship is obtained by observation or measurement, and is expressed by a set of attribute-value pairs. "3", "red", "Peter", and "Johnson" are values. Values are classified into different *value sets*, such as FEET, COLOR, FIRST–NAME, and LAST–NAME. There is a predicate associated with each value set to test whether a value belongs to it. A value in a value set may be equivalent to another value in a different value set. For example, "12" in value set INCH is equivalent to "1" in value set FEET.

An *attribute* can be formally defined as a function which maps from an entity set or a relationship set into a value set or a Cartesian product of value sets:

$$f\colon E_i \text{ or } R_i \rightarrow V_i \text{ or } V_{i_1} \times V_{i_2} \times \cdots \times V_{i_n}.$$

Figure 2 illustrates some attributes defined on entity set PERSON. The attribute AGE maps into value set NO–OF–YEARS. An attribute can map into a Cartesian product of value sets. For example, the attribute NAME maps into value sets FIRST–NAME, and LAST–NAME. Note that more than one attribute may map from the same entity set into the same value set (or same group of value sets). For example, NAME and ALTERNATIVE–NAME map from the entity set EMPLOYEE into value sets FIRST–NAME and LAST–NAME. Therefore, attribute and value set are different concepts although they may have the same name in some cases (for example, EMPLOYEE–NO maps from EMPLOYEE to value set EMPLOYEE–NO). This distinction is not clear in the network model and in many existing data management systems. Also note that an attribute is defined as a function. Therefore, it maps a given entity to a single value (or a single tuple of values in the case of a Cartesian product of value sets).

Note that relationships also have attributes. Consider the relationship set PROJECT–WORKER (Figure 3). The attribute PERCENTAGE–OF–TIME, which is the portion of time a particular employee is committed to a particular project, is an attribute defined on the relationship set PROJECT–WORKER. It is neither an attribute of EMPLOYEE nor an attribute of PROJECT, since its meaning depends on both the employee and project involved. The concept of attribute of relationship is important in understanding the semantics of data and in determining the functional dependencies among data.

2.2.4 Conceptual Information Structure. We are now concerned with how to organize the information associated with entities and relationships. The method proposed in this paper is to separate the information about entities from the infor-

المنارة للاستشارات

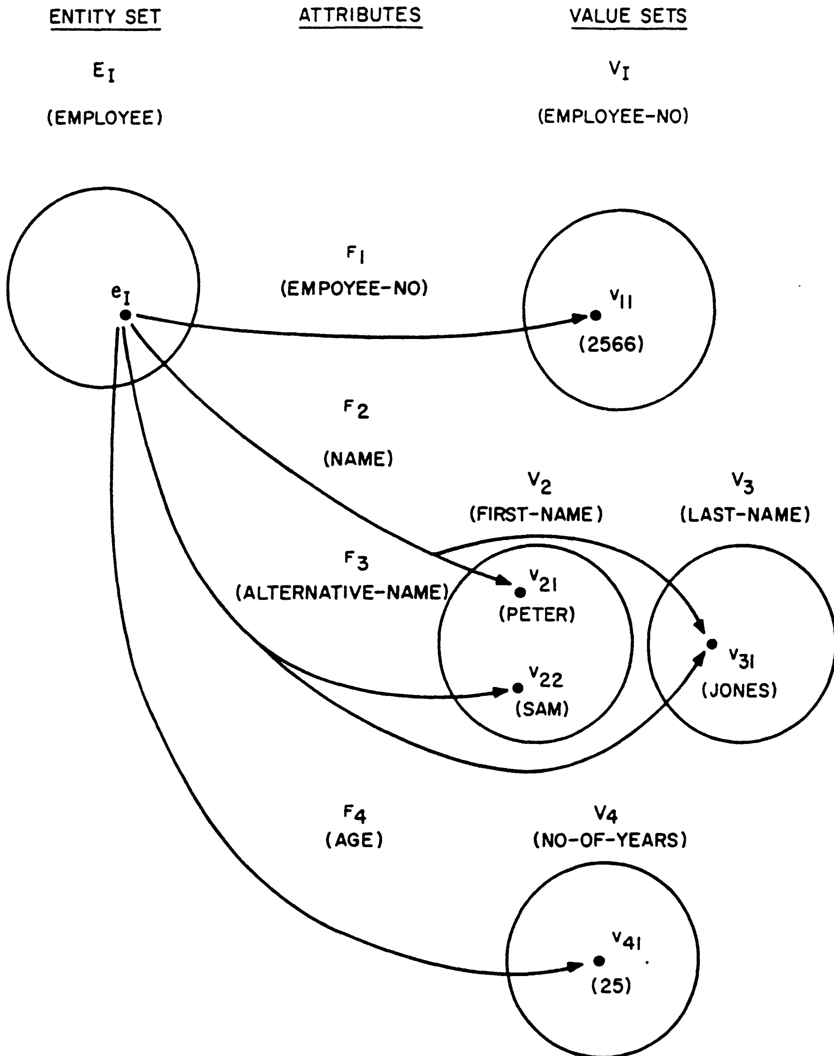


Fig. 2. Attributes defined on the entity set PERSON

mation about relationships. We shall see that this separation is useful in identifying functional dependencies among data.

Figure 4 illustrates in table form the information about entities in an entity set. Each row of values is related to the same entity, and each column is related to a value set which, in turn, is related to an attribute. The ordering of rows and columns is insignificant.

Figure 5 illustrates information about relationships in a relationship set. Note that each row of values is related to a relationship which is indicated by a group of entities, each having a specific role and belonging to a specific entity set.

Note that Figures 4 and 2 (and also Figures 5 and 3) are different forms of the same information. The table form is used for easily relating to the relational model.

المنارة للاستشارات

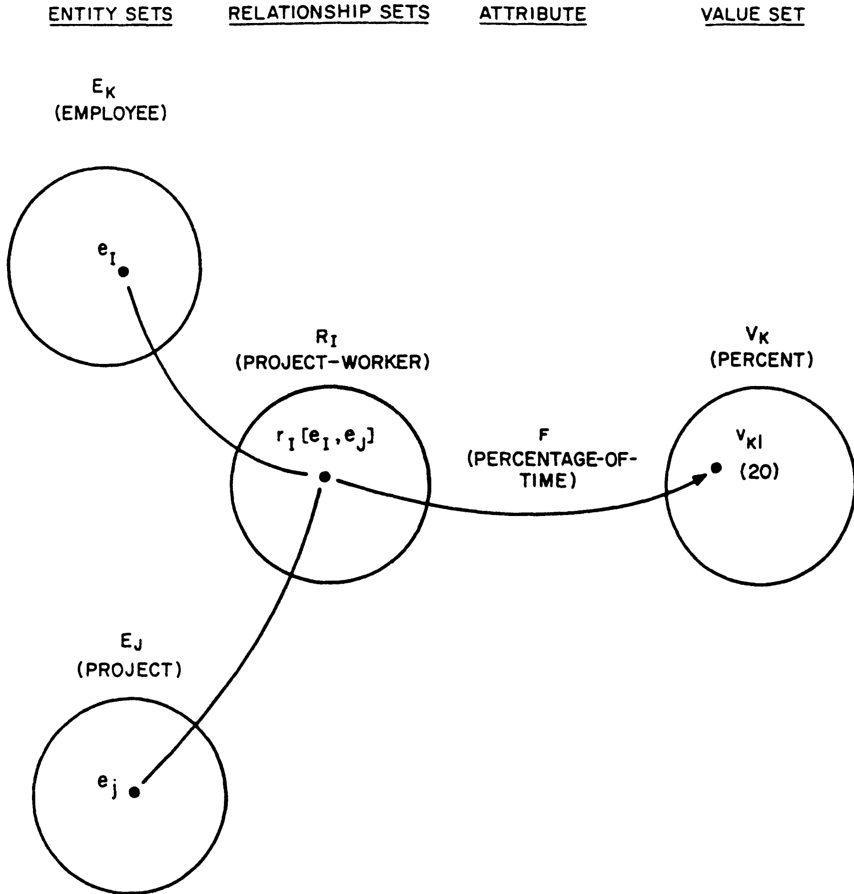


Fig. 3. Attributes defined on the relationship set PROJECT–WORKER

2.3 Information Structure (Level 2)

The entities, relationships, and values at level 1 (see Figures 2–5) are conceptual objects in our minds (i.e. we were in the conceptual realm [18, 27]). At level 2, we consider representations of conceptual objects. We assume that there exist direct representations of values. In the following, we shall describe how to represent entities and relationships.

2.3.1 Primary Key. In Figure 2 the values of attribute EMPLOYEE–NO can be used to identify entities in entity set EMPLOYEE if each employee has a different employee number. It is possible that more than one attribute is needed to identify the entities in an entity set. It is also possible that several groups of attributes may be used to identify entities. Basically, an *entity key* is a group of attributes such that the mapping from the entity set to the corresponding group of value sets is one-to-one. If we cannot find such one-to-one mapping on available data, or if simplicity in identifying entities is desired, we may define an artificial attribute and a value set so that such mapping is possible. In the case where

المنارة للاستشارات

ATTRIBUTE → ; ENTITY SET AND VALUE SET ↘	F_1 (EMPLOYEE-NO)	F_2 (NAME)		F_3 (ALTERNATIVE-NAME)		F_4 (AGE)
E_I (EMPLOYEE)	V_1 (EMPLOYEE-NO)	V_2 (FIRST-NAME)	V_3 (LAST-NAME)	V_2 (FIRST-NAME)	V_3 (LAST-NAME)	V_4 (NO-OF-YEARS)
e_1	v_{11} (2566)	v_{21} (PETER)	v_{31} (JONES)	v_{22} (SAM)	v_{31} (JONES)	v_{41} (25)
e_2	v_{12} (3378)	v_{23} (MARY)	v_{32} (CHEN)	v_{24} (BARB)	v_{33} (CHEN)	v_{42} (23)
⋮	⋮	⋮		⋮		⋮

Fig. 4. Information about entities in an entity set (table form)

ROLE →	WORKER	PROJECT		
			F (PERCENTAGE-OF-TIME)	RELATIONSHIP ATTRIBUTE
ENTITY SET →	E_I (EMPLOYEE)	E_J (PROJECT)	V_K (PERCENTAGE)	VALUE SET
	e_{I1}	e_{J1}	v_{K1} (20)	
	⋮	⋮	⋮	

Fig. 5. Information about relationships in a relationship set (table form)

المنارة للاستشارات

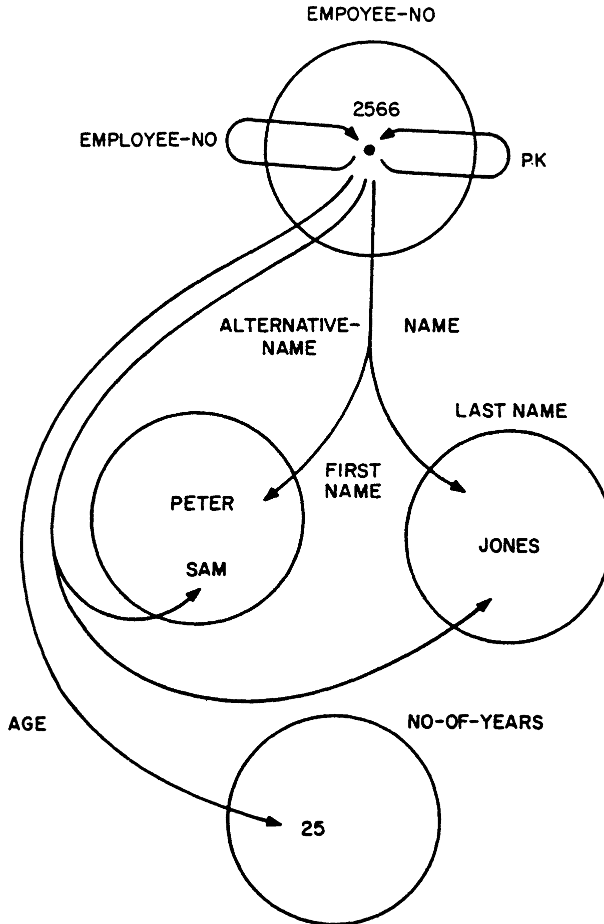


Fig. 6. Representing entities by values (employee numbers)

several keys exist, we usually choose a semantically meaningful key as the *entity primary key* (PK).

Figure 6 is obtained by merging the entity set EMPLOYEE with value set EMPLOYEE-NO in Figure 2. We should notice some semantic implications of Figure 6. Each value in the value set EMPLOYEE-NO represents an entity (employee). Attributes map from the value set EMPLOYEE-NO to other value sets. Also note that the attribute EMPLOYEE-NO maps from the value set EMPLOYEE-NO to itself.

2.3.2 Entity/Relationship Relations. Information about entities in an entity set can now be organized in a form shown in Figure 7. Note that Figure 7 is similar to Figure 4 except that entities are represented by the values of their primary keys. The whole table in Figure 7 is an *entity relation*, and each row is an *entity tuple*.

Since a relationship is identified by the involved entities, the *primary key of a relationship* can be represented by the primary keys of the involved entities. In

المنارة للاستشارات

	PRIMARY KEY					
ATTRIBUTE	EMPLOYEE-NO	NAME		ALTERNATIVE-NAME		AGE
VALUE SET (DOMAIN)	EMPLOYEE-NO	FIRST-NAME	LAST-NAME	FIRST-NAME	LAST-NAME	NO-OF-YEARS
ENTITY (TUPLE)	2566	PETER	JONES	SAM	JONES	25
	3378	MARY	CHEN	BARB	CHEN	23
	⋮	⋮	⋮	⋮	⋮	⋮

(RELATIONAL VIEW)

Fig. 7. Regular entity relation EMPLOYEE

Figure 8, the involved entities are represented by their primary keys EMPLOYEE–NO and PROJECT–NO. The role names provide the semantic meaning for the values in the corresponding columns. Note that EMPLOYEE–NO is the primary key for the involved entities in the relationship and is not an attribute of the relationship. PERCENTAGE–OF–TIME is an attribute of the relationship. The table in Figure 8 is a *relationship relation*, and each row of values is a *relationship tuple.*

In certain cases, the entities in an entity set cannot be uniquely identified by the values of their own attributes; thus we must use a relationship(s) to identify them. For example, consider dependents of employees: dependents are identified by their names and by the values of the primary key of the employees supporting them (i.e. by their relationships with the employees). Note that in Figure 9,

	PRIMARY KEY			
ENTITY RELATION NAME	EMPLOYEE	PROJECT		
ROLE	WORKER	PROJECT		
ENTITY ATTRIBUTE	EMPLOYEE-NO	PROJECT-NO	PERCENTAGE-OF-TIME	RELATIONSHIP ATTRIBUTE
VALUE SET (DOMAIN)	EMPLOYEE-NO	PROJECT-NO	PERCENTAGE	
RELATIONSHIP TUPLE	2566	31	20	
	2173	25	100	
	⋮	⋮	⋮	

(RELATIONAL VIEW)

Fig. 8. Regular relationship relation PROJECT–WORKER

المنارة للاستشارات

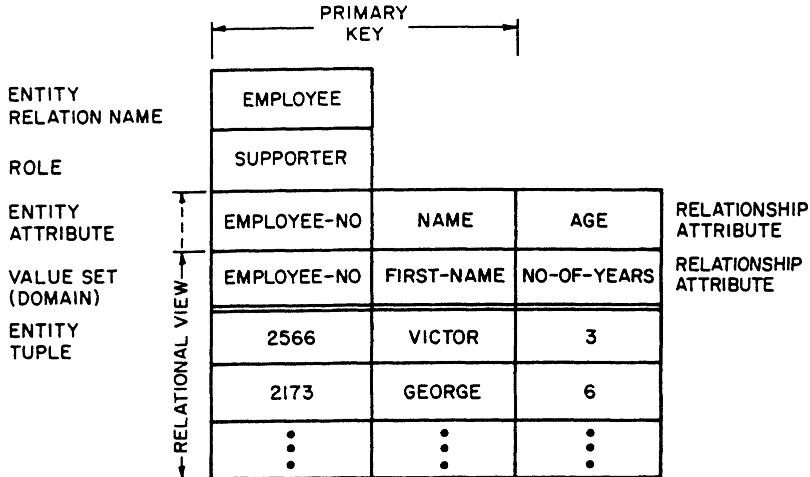


Fig. 9. A weak entity relation DEPENDENT

EMPLOYEE-NO is not an attribute of an entity in the set DEPENDENT but is the primary key of the employees who support dependents. Each row of values in Figure 9 is an entity tuple with EMPLOYEE-NO and NAME as its primary key. The whole table is an entity relation.

Theoretically, any kind of relationship may be used to identify entities. For simplicity, we shall restrict ourselves to the use of only one kind of relationship: the binary relationships with $1:n$ mapping in which the existence of the n entities on one side of the relationship depends on the existence of one entity on the other side of the relationship. For example, one employee may have n $(= 0, 1, 2, \ldots)$ dependents, and the existence of the dependents depends on the existence of the corresponding employee.

This method of identification of entities by relationships with other entities can be applied recursively until the entities which can be identified by their own attribute values are reached. For example, the primary key of a department in a company may consist of the department number and the primary key of the division, which in turn consists of the division number and the name of the company.

Therefore, we have two forms of entity relations. If relationships are used for identifying the entities, we shall call it a *weak entity relation* (Figure 9). If relationships are not used for identifying the entities, we shall call it a *regular entity relation* (Figure 7). Similarly, we also have two forms of relationship relations. If all entities in the relationship are identified by their own attribute values, we shall call it a *regular relationship relation* (Figure 8). If some entities in the relationship are identified by other relationships, we shall call it a *weak relationship relation*. For example, any relationships between DEPENDENT entities and other entities will result in weak relationship relations, since a DEPENDENT entity is identified by its name and its relationship with an EMPLOYEE entity. The distinction between regular (entity/relationship) relations and weak (entity/relationship) relations will be useful in maintaining data integrity.

المنارة للاستشارات

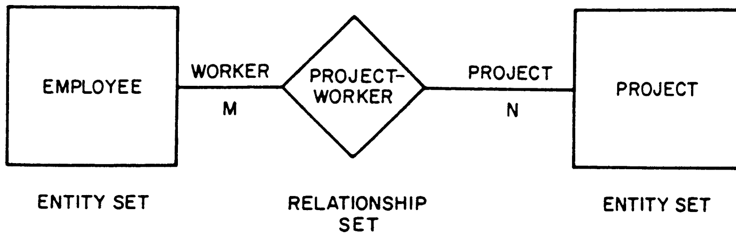


Fig. 10. A simple entity-relationship diagram

3. ENTITY-RELATIONSHIP DIAGRAM AND INCLUSION OF SEMANTICS IN DATA DESCRIPTION AND MANIPULATION

3.1 System Analysis Using the Entity-Relationship Diagram

In this section we introduce a diagrammatic technique for exhibiting entities and relationships: the entity-relationship diagram.

Figure 10 illustrates the relationship set PROJECT–WORKER and the entity sets EMPLOYEE and PROJECT using this diagrammatic technique. Each entity set is represented by a rectangular box, and each relationship set is represented by a diamond-shaped box. The fact that the relationship set PROJECT–WORKER is defined on the entity sets EMPLOYEE and PROJECT is represented by the lines connecting the rectangular boxes. The roles of the entities in the relationship are stated.

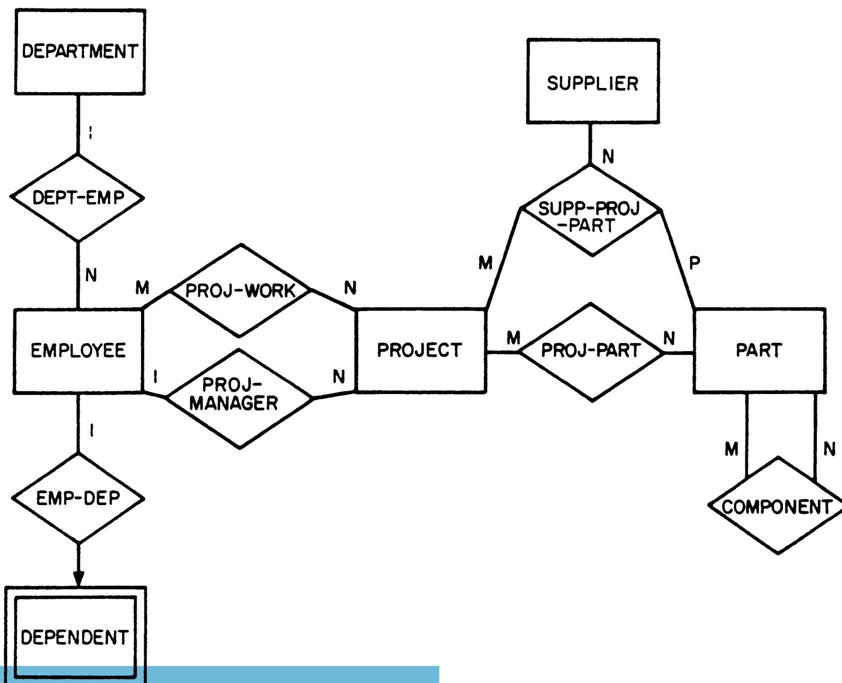


Fig. 11. An entity-relationship diagram for analysis of information in a manufacturing firm

المنارة للاستشارات

Figure 11 illustrates a more complete diagram of some entity sets and relationship sets which might be of interest to a manufacturing company. DEPARTMENT, EMPLOYEE, DEPENDENT, PROJECT, SUPPLIER, and PART are entity sets. DEPARTMENT–EMPLOYEE, EMPLOYEE–DEPENDENT, PROJECT–WORKER, PROJECT–MANAGER, SUPPLIER–PROJECT–PART, PROJECT–PART, and COMPONENT are relationship sets. The COMPONENT relationship describes what subparts (and quantities) are needed in making superparts. The meaning of the other relationship sets need not be explained.

Several important characteristics about relationships in general can be found in Figure 11:

(1) A relationship set may be defined on more than two entity sets. For example, the SUPPLIER–PROJECT–PART relationship set is defined on three entity sets: SUPPLIER, PROJECT, and PART.

(2) A relationship set may be defined on only one entity set. For example, the relationship set COMPONENT is defined on one entity set, PART.

(3) There may be more than one relationship set defined on given entity sets. For example, the relationship sets PROJECT–WORKER and PROJECT–MANAGER are defined on the entity sets PROJECT and EMPLOYEE.

(4) The diagram can distinguish between $1{:}n$, $m{:}n$, and $1{:}1$ mappings. The relationship set DEPARTMENT–EMPLOYEE is a $1{:}n$ mapping, that is, one department may have n ($n = 0, 1, 2, \ldots$) employees and each employee works for only one department. The relationship set PROJECT–WORKER is an $m{:}n$ mapping, that is, each project may have zero, one, or more employees assigned to it and each employee may be assigned to zero, one, or more projects. It is also possible to express 1:1 mappings such as the relationship set MARRIAGE. Information about the number of entities in each entity set which is allowed in a relationship set is indicated by specifying "1", "*m*", "*n*" in the diagram. The relational model and the entity set model[2] do not include this type of information; the network model cannot express a 1:1 mapping easily.

(5) The diagram can express the *existence dependency* of one entity type on another. For example, the arrow in the relationship set EMPLOYEE–DEPENDENT indicates that existence of an entity in the entity set DEPENDENT depends on the corresponding entity in the entity set EMPLOYEE. That is, if an employee leaves the company, his dependents may no longer be of interest.

Note that the entity set DEPENDENT is shown as a special rectangular box. This indicates that at level 2 the information about entities in this set is organized as a weak entity relation (using the primary key of EMPLOYEE as a part of its primary key).

3.2 An Example of a Database Design and Description

There are four steps in designing a database using the entity-relationship model: (1) identify the entity sets and the relationship sets of interest; (2) identify semantic information in the relationship sets such as whether a certain relationship

---

[2] This mapping information is included in DIAM II [24].

المنارة للاستشارات

set is an $1{:}n$ mapping; (3) define the value sets and attributes; (4) organize data into entity/relationship relations and decide primary keys.

Let us use the manufacturing company discussed in Section 3.1 as an example. The results of the first two steps of database design are expressed in an entity-relationship diagram as shown in Figure 11. The third step is to define value sets and attributes (see Figures 2 and 3). The fourth step is to decide the primary keys for the entities and the relationships and to organize data as entity/relationship relations. Note that each entity/relationship set in Figure 11 has a corresponding entity/relationship relation. We shall use the names of the entity sets (at level 1) as the names of the corresponding entity/relationship relations (at level 2) as long as no confusion will result.

At the end of the section, we illustrate a schema (data definition) for a small part of the database in the above manufacturing company example (the syntax of the data definition is not important). Note that value sets are defined with specifications of representations and allowable values. For example, values in EMPLOYEE-NO are represented as 4-digit integers and range from 0 to 2000. We then declare three entity relations: EMPLOYEE, PROJECT, and DEPENDENT. The attributes and value sets defined on the entity sets as well as the primary keys are stated. DEPENDENT is a weak entity relation since it uses EMPLOYEE.PK as part of its primary key. We also declare two relationship relations: PROJECT–WORKER and EMPLOYEE–DEPENDENT. The roles and involved entities in the relationships are specified. We use EMPLOYEE.PK to indicate the name of the entity relation (EMPLOYEE) and whatever attribute-value-set pairs are used as the primary keys in that entity relation. The maximum number of entities from an entity set in a relation is stated. For example, PROJECT–WORKER is an $m{:}n$ mapping. We may specify the values of m and n. We may also specify the minimum number of entities in addition to the maximum number. EMPLOYEE–DEPENDENT is a weak relationship relation since one of the related entity relations, DEPENDENT, is a weak entity relation. Note that the existence dependence of the dependents on the supporter is also stated.

```
DECLARE     VALUE-SETS       REPRESENTATION     ALLOWABLE-VALUES
            EMPLOYEE-NO      INTEGER (4)        (0,2000)
            FIRST-NAME       CHARACTER (8)      ALL
            LAST-NAME        CHARACTER (10)     ALL
            NO-OF-YEARS      INTEGER (3)        (0,100)
            PROJECT-NO       INTEGER (3)        (1,500)
            PERCENTAGE       FIXED (5.2)        (0,100.00)

DECLARE     REGULAR ENTITY RELATION EMPLOYEE
            ATTRIBUTE/VALUE-SET:
                EMPLOYEE-NO/EMPLOYEE-NO
                NAME/(FIRST-NAME, LAST-NAME)
                ALTERNATIVE-NAME/(FIRST-NAME,LAST-NAME)
                AGE/NO-OF-YEARS
            PRIMARY KEY:
                EMPLOYEE-NO
```

المنارة للاستشارات

```
DECLARE    REGULAR ENTITY RELATION PROJECT
           ATTRIBUTE/VALUE-SET:
                PROJECT-NO/PROJECT-NO
           PRIMARY KEY:
                PROJECT-NO

DECLARE    REGULAR RELATIONSHIP RELATION PROJECT-WORKER
           ROLE/ENTITY-RELATION.PK/MAX-NO-OF-ENTITIES
                WORKER/EMPLOYEE.PK/m
                PROJECT/PROJECT.PK/n      (m:n mapping)
           ATTRIBUTE/VALUE-SET:
                PERCENTAGE-OF-TIME/PERCENTAGE

DECLARE    WEAK RELATIONSHIP RELATION EMPLOYEE-DEPENDENT
           ROLE/ENTITY-RELATION.PK/MAX-NO-OF-ENTITIES
                SUPPORTER/EMPLOYEE.PK/1
                DEPENDENT/DEPENDENT.PK/n
           EXISTENCE OF DEPENDENT DEPENDS ON
           EXISTENCE OF SUPPORTER

DECLARE    WEAK ENTITY RELATION DEPENDENT
           ATTRIBUTE/VALUE-SET:
                NAME/FIRST-NAME
                AGE/NO-OF-YEARS
           PRIMARY KEY:
                NAME
                EMPLOYEE.PK THROUGH EMPLOYEE-DEPENDENT
```

3.3 Implications on Data Integrity

Some work has been done on data integrity for other models [8, 14, 16, 28]. With explicit concepts of entity and relationship, the entity-relationship model will be useful in understanding and specifying constraints for maintaining data integrity. For example, there are three major kinds of constraints on values:

(1) Constraints on *allowable values* for a value set. This point was discussed in defining the schema in Section 3.2.

(2) Constraints on *permitted* values for a certain attribute. In some cases, not all allowable values in a value set are permitted for some attributes. For example, we may have a restriction of ages of employees to between 20 and 65. That is,

$$\text{AGE}(e) \in (20,65), \text{ where } e \in \text{EMPLOYEE}.$$

Note that we use the level 1 notations to clarify the semantics. Since each entity/relationship set has a corresponding entity/relationship relation, the above expression can be easily translated into level 2 notations.

(3) Constraints on *existing values* in the database. There are two types of constraints:

(i) Constraints between sets of existing values. For example,

$$\{\text{NAME}(e) \mid e \in \text{MALE-PERSON}\} \subseteq \{\text{NAME}(e) \mid e \in \text{PERSON}\}.$$

المنارة للاستشارات

(ii) Constraints between particular values. For example,

$$\text{TAX}(e) \leq \text{SALARY}(e),\ e \in \text{EMPLOYEE}$$

or

$$\text{BUDGET}(e_i) = \sum \text{BUDGET}(e_j), \text{ where } e_i \in \text{COMPANY}, \; e_j \in \text{DEPARTMENT} \text{ and } [e_i, e_j] \in \text{COMPANY–DEPARTMENT}.$$

3.4 Semantics and Set Operations of Information Retrieval Requests

The semantics of information retrieval requests become very clear if the requests are based on the entity-relationship model of data. For clarity, we first discuss the situation at level 1. Conceptually, the information elements are organized as in Figures 4 and 5 (on Figures 2 and 3). Many information retrieval requests can be considered as a combination of the following basic types of operations:

(1) Selection of a subset of values from a value set.

(2) Selection of a subset of entities from an entity set (i.e. selection of certain rows in Figure 4). Entities are selected by stating the values of certain attributes (i.e. subsets of value sets) and/or their relationships with other entities.

(3) Selection of a subset of relationships from a relationship set (i.e. selection of certain rows in Figure 5). Relationships are selected by stating the values of certain attribute(s) and/or by identifying certain entities in the relationship.

(4) Selection of a subset of attributes (i.e. selection of columns in Figures 4 and 5).

An information retrieval request like "What are the ages of the employees whose weights are greater than 170 and who are assigned to the project with PROJECT-NO 254?" can be expressed as:

{AGE(e) | e ∈ EMPLOYEE, WEIGHT(e) > 170,
[e, e_j] ∈ PROJECT–WORKER, e_j ∈ PROJECT,
PROJECT–NO(e_j) = 254};

or

{AGE(EMPLOYEE) | WEIGHT(EMPLOYEE) > 170,
[EMPLOYEE,PROJECT] ∈ PROJECT–WORKER,
PROJECT–NO(EMPLOYEE) = 254}.

To retrieve information as organized in Figure 6 at level 2, "entities" and "relationships" in (2) and (3) should be replaced by "entity PK" and "relationship PK." The above information retrieval request can be expressed as:

{AGE(EMPLOYEE.PK) | WEIGHT(EMPLOYEE.PK) > 170
(WORKER/EMPLOYEE.PK,PROJECT/PROJECT.PK) ∈ {PROJECT–WORKER.PK},
PROJECT–NO (PROJECT.PK) = 254}.

To retrieve information as organized in entity/relationship relations (Figures 7, 8, and 9), we can express it in a SEQUEL-like language [6]:

```
SELECT    AGE
FROM      EMPLOYEE
WHERE     WEIGHT > 170
```

المنارة للاستشارات

Table I. Insertion

level 1	level 2
operation: insert an entity to an entity set	*operation:* create an entity tuple with a certain entity-PK *check:* whether PK already exists or is acceptable
operation: insert a relationship in a relationship set *check:* whether the entities exist	*operation:* create a relationship tuple with given entity PKs *check:* whether the entity PKs exist
operation: insert properties of an entity or a relationship *check:* whether the value is acceptable	*operation:* insert values in an entity tuple or a relationship tuple *check:* whether the values are acceptable

```
AND        EMPLOYEE.PK =
           SELECT    WORKER/EMPLOYEE.PK
           FROM      PROJECT-WORKER
           WHERE     PROJECT-NO = 254.
```

It is possible to retrieve information about entities in two different entity sets without specifying a relationship between them. For example, an information retrieval request like "List the names of employees and ships which have the same

Table II. Updating

level 1	level 2
operation: • change the value of an entity attribute	*operation:* • update a value *consequence:* • if it is not part of an entity PK, no consequence • if it is part of an entity PK, •• change the entity PKs in all related relationship relations •• change PKs of other entities which use this value as part of their PKs (for example, DEPENDENTS' PKs use EMPLOYEE'S PK)
operation: • change the value of a relationship attribute	*operation:* • update a value (note that a relationship attribute will not be a relationship PK)

المنارة للاستشارات

Table III. Deletion

level 1	level 2
operation: • delete an entity *consequences:* • delete any entity whose existence depends on this entity • delete relationships involving this entity • delete all related properties	*operation:* • delete an entity tuple *consequences* (applied recursively): • delete any entity tuple whose existence depends on this entity tuple • delete relationship tuples associated with this entity
operation: • delete a relationship *consequences:* • delete all related properties	*operation:* • delete a relationship tuple

age" can be expressed in the level 1 notation as:

$$\{(\text{NAME}(e_i),\text{NAME}(e_j)) \mid e_i \in \text{EMPLOYEE}, e_j \in \text{SHIP}, \text{AGE}(e_i) = \text{AGE}(e_j)\}.$$

We do not further discuss the language syntax here. What we wish to stress is that information requests may be expressed using set notions and set operations [17], and the request semantics are very clear in adopting this point of view.

3.5 Semantics and Rules for Insertion, Deletion, and Updating

It is always a difficult problem to maintain data consistency following insertion, deletion, and updating of data in the database. One of the major reasons is that the semantics and consequences of insertion, deletion, and updating operations usually are not clearly defined; thus it is difficult to find a set of rules which can enforce data consistency. We shall see that this data consistency problem becomes simpler using the entity-relationship model.

In Tables I–III, we discuss the semantics and rules[3] for insertion, deletion, and updating at both level 1 and level 2. Level 1 is used to clarify the semantics.

4. ANALYSIS OF OTHER DATA MODELS AND THEIR DERIVATION FROM THE ENTITY-RELATIONSHIP MODEL

4.1 The Relational Model

4.1.1 The Relational View of Data and Ambiguity in Semantics. In the relational model, *relation*, R, is a mathematical relation defined on sets $X_1, X_2, \ldots, X_n$:

$$R = \{(x_1, x_2, \ldots, x_n) \mid x_1 \in X_1, x_2 \in X_2, \ldots, x_n \in X_n\}.$$

The sets $X_1, X_2, \ldots, X_n$ are called *domains*, and $(x_1, x_2, \ldots, x_n)$ is called a *tuple*. Figure 12 illustrates a relation called EMPLOYEE. The domains in the relation

[3] Our main purpose is to illustrate the semantics of data manipulation operations. Therefore, these rules may not be complete. Note that the consequence of operations stated in the tables can be performed by the system instead of by the users.

المنارة للاستشارات

ROLE		LEGAL	LEGAL	ALTERNATIVE	ALTERNATIVE	
DOMAIN	EMPLOYEE-NO	FIRST-NAME	LAST-NAME	FIRST-NAME	LAST-NAME	NO-OF-YEARS
TUPLE	2566	PETER	JONES	SAM	JONES	25
	3378	MARY	CHEN	BARB	CHEN	23

Fig. 12. Relation EMPLOYEE

are EMPLOYEE-NO, FIRST-NAME, LAST-NAME, FIRST-NAME, LAST-NAME, NO-OF-YEAR. The ordering of rows and columns in the relation has no significance. To avoid ambiguity of columns with the same domain in a relation, domain names are qualified by *roles* (to distinguish the role of the domain in the relation). For example, in relation EMPLOYEE, domains FIRST-NAME and LAST-NAME may be qualified by roles LEGAL or ALTERNATIVE. An *attribute name* in the relational model is a domain name concatenated with a role name [10]. Comparing Figure 12 with Figure 7, we can see that "domains" are basically equivalent to value sets. Although "role" or "attribute" in the relational model seems to serve the same purpose as "attribute" in the entity-relationship model, the semantics of these terms are different. The "role" or "attribute" in the relational model is mainly used to distinguish domains with the same name in the same relation, while "attribute" in the entity-relationship model is a function which maps from an entity (or relationship) set into value set(s).

Using relational operators in the relational model may cause semantic ambiguities. For example, the join of the relation EMPLOYEE with the relation EMPLOYEE-PROJECT (Figure 13) on domain EMPLOYEE-NO produces the

PROJECT-NO	EMPLOYEE-NO
7	2566
3	2566
7	3378

Fig. 13. Relation EMPLOYEE-PROJECT

المنارة للاستشارات

		LEGAL	LEGAL	ALTERNATIVE	ALTERNATIVE	
PROJECT-NO	EMPLOYEE-NO	FIRST-NAME	LAST-NAME	FIRST-NAME	LAST-NAME	NO-OF-YEARS
7	2566	PETER	JONES	SAM	JONES	25
3	2566	PETER	JONES	SAM	JONES	25
7	3378	MARY	CHEN	BARB	CHEN	23

Fig. 14. Relation EMPLOYEE–PROJECT′ as a "join" of relations EMPLOYEE and EMPLOYEE–PROJECT

relation EMPLOYEE–PROJECT' (Figure 14). But what is the meaning of a join between the relation EMPLOYEE with the relation SHIP on the domain NO–OF–YEARS (Figure 15)? The problem is that the same domain name may have different semantics in different relations (note that a role is intended to distinguish domains in a given relation, not in all relations). If the domain NO–OF–YEAR of the relation EMPLOYEE is not allowed to be compared with the domain NO–OF–YEAR of the relation SHIP, different domain names have to be declared. But if such a comparison is acceptable, can the database system warn the user?

In the entity-relationship model, the semantics of data are much more apparent. For example, one column in the example stated above contains the values of AGE of EMPLOYEE and the other column contains the values of AGE of SHIP. If this semantic information is exposed to the user, he may operate more cautiously (refer to the sample information retrieval requests stated in Section 3.4). Since the database system contains the semantic information, it should be able to warn the user of the potential problems for a proposed "join-like" operation.

4.1.2 Semantics of Functional Dependencies Among Data. In the relational model, "attribute" B of a relation is *functionally dependent* on "attribute" A of the same relation if each value of A has no more than one value of B associated with it in the relation. Semantics of functional dependencies among data become clear

SHIP-NO	NAME	NO-OF-YEARS
037	MISSOURI	25
056	VIRGINIA	10

Fig. 15. Relation SHIP

المنارة للاستشارات

in the entity-relationship model. Basically, there are two major types of functional dependencies:

(1) Functional dependencies related to description of entities or relationships. Since an attribute is defined as a function, it maps an entity in an entity set to a single value in a value set (see Figure 2). At level 2, the values of the primary key are used to represent entities. Therefore, nonkey value sets (domains) are functionally dependent on primary-key value sets (for example, in Figures 6 and 7, NO-OF-YEARS is functionally dependent on EMPLOYEE-NO). Since a relation may have several keys, the nonkey value sets will functionally depend on any key value set. The key value sets will be mutually functionally dependent on each other. Similarly, in a relationship relation the nonkey value sets will be functionally dependent on the prime-key value sets (for example, in Figure 8, PERCENTAGE is functionally dependent on EMPLOYEE-NO and PROJECT-NO).

(2) Functional dependencies related to entities in a relationship. Note that in Figure 11 we identify the types of mappings (1:*n*, *m*:*n*, etc.) for relationship sets. For example, PROJECT-MANAGER is a 1:*n* mapping. Let us assume that PROJECT-NO is the primary key in the entity relation PROJECT. In the relationship relation PROJECT-MANAGER, the value set EMPLOYEE-NO will be functionally dependent on the value set PROJECT-NO (i.e. each project has only one manager).

The distinction between level 1 (Figure 2) and level 2 (Figures 6 and 7) and the separation of entity relation (Figure 7) from relationship relation (Figure 8) clarifies the semantics of functional dependencies among data.

4.1.3 3NF Relations Versus Entity-Relationship Relations. From the definition of "relation," any grouping of domains can be considered to be a relation. To avoid undesirable properties in maintaining relations, a normalization process is proposed to transform arbitrary relations into the first normal form, then into the second normal form, and finally into the third normal form (3NF) [9, 11]. We shall show that the entity and relationship relations in the entity-relationship model are similar to 3NF relations but with clearer semantics and without using the transformation operation.

Let us use a simplified version of an example of normalization described in [9]. The following three relations are in first normal form (that is, there is no domain whose elements are themselves relations):

EMPLOYEE (EMPLOYEE-NO)
PART (PART-NO, PART-DESCRIPTION, QUANTITY-ON-HAND)
PART-PROJECT (PART-NO, PROJECT-NO, PROJECT-DESCRIPTION,
PROJECT-MANAGER-NO, QUANTITY-COMMITTED).

Note that the domain PROJECT-MANAGER-NO actually contains the EMPLOYEE-NO of the project manager. In the relations above, primary keys are underlined.

Certain rules are applied to transform the relations above into third normal form:

EMPLOYEE' (EMPLOYEE-NO)
PART' (PART-NO, PART-DESCRIPTION, QUANTITY-ON-HAND)

المنارة للاستشارات

PROJECT' (PROJECT-NO, PROJECT-DESCRIPTION, PROJECT-MANAGER-NO)
PART-PROJECT' (PART-NO, PROJECT-NO, QUANTITY-COMMITTED).

Using the entity-relationship diagram in Figure 11, the following entity and relationship relations can be easily derived:

entity relations
PART'' (PART-NO, PART-DESCRIPTION, QUANTITY-ON-HAND)
PROJECT'' (PROJECT-NO, PROJECT-DESCRIPTION)
EMPLOYEE'' (EMPLOYEE-NO)

relationship relations
PART-PROJECT'' (PART/PART-NO, PROJECT/PROJECT-NO, QUANTITY-COMMITTED)
PROJECT-MANAGER'' (PROJECT/PROJECT-NO, MANAGER/EMPLOYEE-NO).

The role names of the entities in relationships (such as MANAGER) are indicated. The entity relation names associated with the PKs of entities in relationships and the value set names have been omitted.

Note that in the example above, entity/relationship relations are similar to the 3NF relations. In the 3NF approach, PROJECT-MANAGER-NO is included in the relation PROJECT' since PROJECT-MANAGER-NO is assumed to be functionally dependent on PROJECT-NO. In the entity-relationship model, PROJECT-MANAGER-NO (i.e. EMPLOYEE-NO of a project manager) is included in a relationship relation PROJECT-MANAGER since EMPLOYEE-NO is considered as an entity PK in this case.

Also note that in the 3NF approach, changes in functional dependencies of data may cause some relations not to be in 3NF. For example, if we make a new assumption that one project may have more than one manager, the relation PROJECT' is no longer a 3NF relation and has to be split into two relations as PROJECT'' and PROJECT-MANAGER''. Using the entity-relationship model, no such change is necessary. Therefore, we may say that by using the entity-relationship model we can arrange data in a form similar to 3NF relations but with clear semantic meaning.

It is interesting to note that the decomposition (or transformation) approach described above for normalization of relations may be viewed as a bottom-up approach in database design.[4] It starts with arbitrary relations (level 3 in Figure 1) and then uses some semantic information (functional dependencies of data) to transform them into 3NF relations (level 2 in Figure 1). The entity-relationship model adopts a top-down approach, utilizing the semantic information to organize data in entity/relationship relations.

4.2 The Network Model

4.2.1 Semantics of the Data-Structure Diagram. One of the best ways to explain the network model is by use of the *data-structure diagram* [3]. Figure 16(a) illustrates a data-structure diagram. Each rectangular box represents a record type.

[4] Although the decomposition approach was emphasized in the relational model literature, it is a procedure to obtain 3NF and may not be an intrinsic property of 3NF.

المنارة للاستشارات

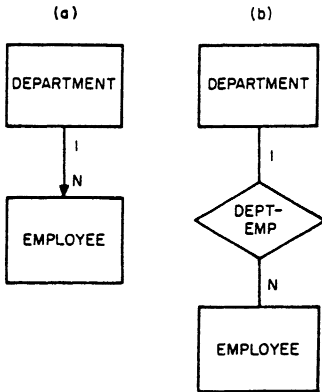


Fig. 16. Relationship DEPARTMENT–EMPLOYEE
(a) data structure diagram
(b) entity-relationship diagram

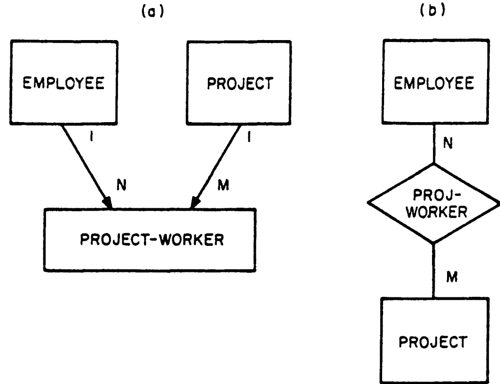


Fig. 17. Relationship PROJECT–WORKER
(a) data structure diagram
(b) entity–relationship diagram

The arrow represents a data-structure-set in which the DEPARTMENT record is the *owner-record,* and one owner-record may own n ($n = 0, 1, 2, \ldots$) *member-records.* Figure 16(b) illustrates the corresponding entity-relationship diagram. One might conclude that the arrow in the data-structure diagram represents a relationship between entities in two entity sets. This is not always true. Figures 17(a) and 17(b) are the data-structure diagram and the entity-relationship diagram expressing the relationship PROJECT–WORKER between two entity types EMPLOYEE and PROJECT. We can see in Figure 17(a) that the relationship PROJECT–WORKER becomes another record type and that the arrows no longer represent relationships between entities. What are the real meanings of the arrows in data-structure diagrams? The answer is that an arrow represents an $1{:}n$ relationship between two *record* (not entity) types and also implies the existence of an access path from the owner record to the member records. The data-structure diagram is a representation of the organization of records (level 4 in Figure 1) and is not an exact representation of entities and relationships.

4.2.2 Deriving the Data-Structure Diagram. Under what conditions does an arrow in a data-structure diagram correspond to a relationship of entities? A close comparison of the data-structure diagrams with the corresponding entity-relationship diagrams reveals the following rules:

1. For $1{:}n$ binary relationships an arrow is used to represent the relationship (see Figure 16(a)).

2. For $m{:}n$ binary relationships a "relationship record" type is created to represent the relationship and arrows are drawn from the "entity record" type to the "relationship record" type (see Figure 17(a)).

3. For k-ary ($k \geq 3$) relationships, the same rule as (2) applies (i.e. creating a "relationship record" type).

Since DBTG [7] does not allow a data-structure-set to be defined on a single record type (i.e. Figure 18 is not allowed although it has been implemented in [13]), a "relationship record" is needed to implement such relationships (see

المنارة للاستشارات

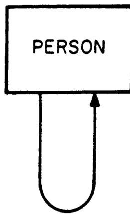
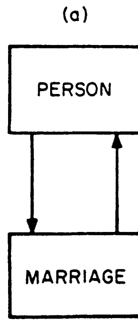


Fig. 18. Data-structure-set defined on the same record type

(a)



(b)

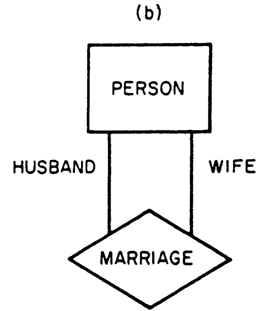


Fig. 19. Relationship MARRIAGE (a) data structure diagram (b) entity–relationship diagram

Figure 19(a)) [20]. The corresponding entity-relationship diagram is shown in Figure 19(b).

It is clear now that arrows in a data-structure diagram do not always represent relationships of entities. Even in the case that an arrow represents a $1{:}n$ relationship, the arrow only represents a unidirectional relationship [20] (although it is possible to find the owner-record from a member-record). In the entity-relationship model, both directions of the relationship are represented (the roles of both entities are specified). Besides the semantic ambiguity in its arrows, the network model is awkward in handling changes in semantics. For example, if the relationship between DEPARTMENT and EMPLOYEE changes from a $1{:}n$ mapping to an $m{:}n$ mapping (i.e. one employee may belong to several departments), we must create a relationship record DEPARTMENT–EMPLOYEE in the network model.

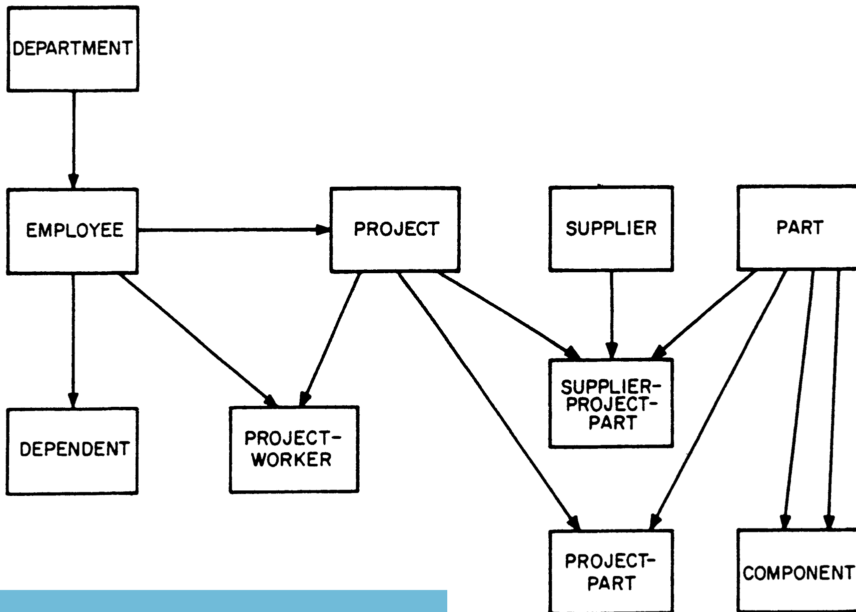


Fig. 20. The data structure diagram derived from the entity–relationship diagram in Fig. 11

المنارة للاستشارات

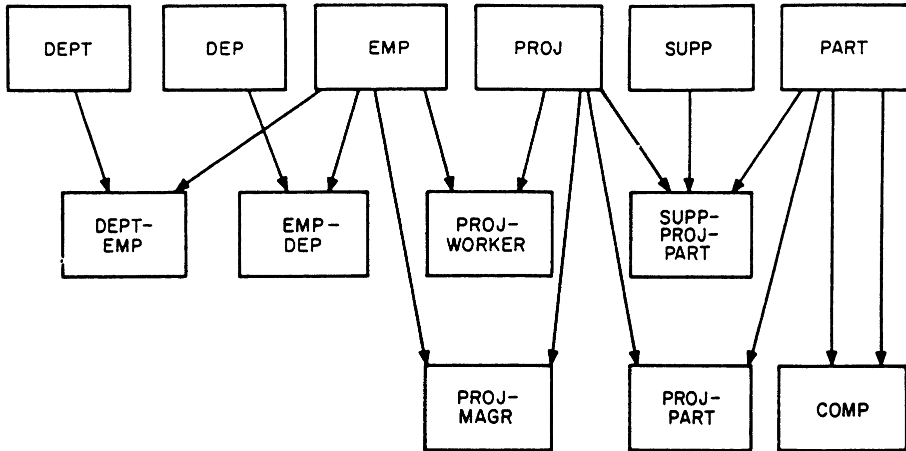


Fig. 21. The "disciplined" data structure diagram derived from the entity-relationship diagram in Fig. 11

In the entity-relationship model, all kinds of mappings in relationships are handled uniformly.

The entity-relationship model can be used as a tool in the structured design of databases using the network model. The user first draws an entity-relationship diagram (Figure 11). He may simply translate it into a data-structure diagram (Figure 20) using the rules specified above. He may also follow a discipline that every entity or relationship must be mapped onto a record (that is, "relationship records" are created for all types of relationships no matter that they are $1:n$ or $m:n$ mappings). Thus, in Figure 11, all one needs to do is to change the diamonds to boxes and to add arrowheads on the appropriate lines. Using this approach three more boxes—DEPARTMENT–EMPLOYEE, EMPLOYEE–DEPENDENT, and PROJECT–MANAGER—will be added to Figure 20 (see Figure 21). The validity constraints discussed in Sections 3.3–3.5 will also be useful.

4.3 The Entity Set Model

4.3.1 The Entity Set View. The basic element of the entity set model is the entity. Entities have names (*entity names*) such as "Peter Jones", "blue", or "22". Entity names having some properties in common are collected into an *entity-name-set*, which is referenced by the *entity-name-set-name* such as "NAME", "COLOR", and "QUANTITY".

An entity is represented by the entity-name-set-name/entity-name pair such as NAME/Peter Jones, EMPLOYEE–NO/2566, and NO–OF–YEARS/20. An entity is described by its association with other entities. Figure 22 illustrates the entity set view of data. The "DEPARTMENT" of entity EMPLOYEE–NO/2566 is the entity DEPARTMENT–NO/405. In other words, "DEPARTMENT" is the role that the entity DEPARTMENT–NO/405 plays to describe the entity EMPLOYEE–NO/2566. Similarly, the "NAME", "ALTERNATIVE–NAME", or "AGE" of EMPLOYEE–NO/2566 is "NAME/Peter Jones", "NAME/Sam Jones", or "NO–OF–YEARS/20", respectively. The description of the entity EMPLOYEE–

المنارة للاستشارات

NO/2566 is a collection of the related entities and their roles (the entities and roles circled by the dotted line). An example of the *entity description* of "EMPLOYEE-NO/2566" (in its full-blown, unfactored form) is illustrated by the set of role-name/entity-name-set-name/entity-name triplets shown in Figure 23. Conceptually, the entity set model differs from the entity-relationship model in the following ways:

(1) In the entity set model, everything is treated as an entity. For example, "COLOR/BLACK" and "NO-OF-YEARS/45" are entities. In the entity-relationship model, "blue" and "36" are usually treated as values. Note treating values as entities may cause semantic problems. For example, in Figure 22, what is the difference between "EMPLOYEE-NO/2566", "NAME/Peter Jones", and "NAME/Sam Jones"? Do they represent different entities?

(2) Only binary relationships are used in the entity set model,[5] while n-ary relationships may be used in the entity-relationship model.

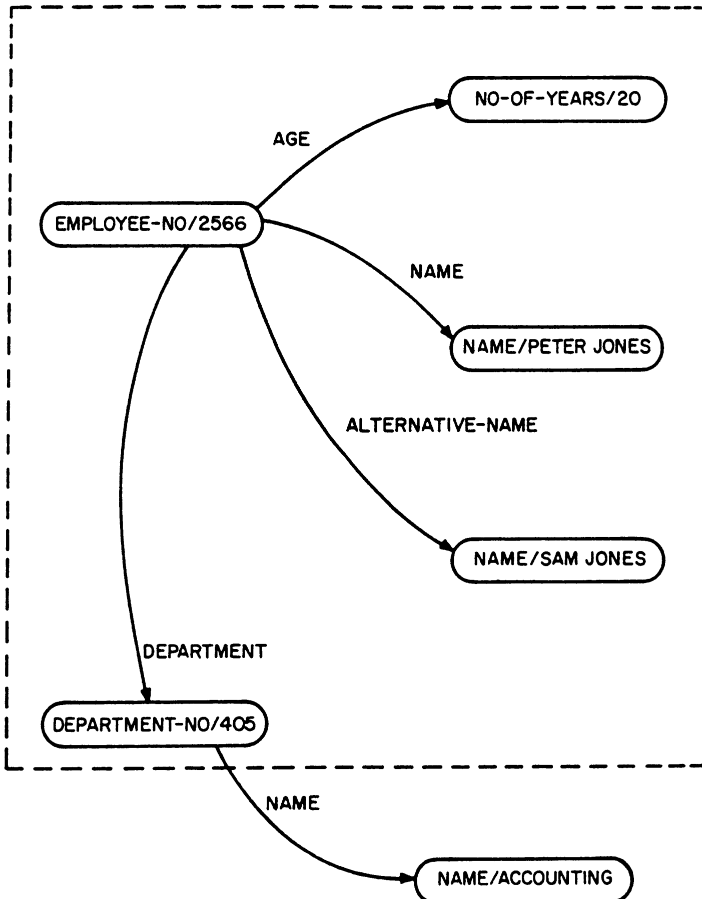


Fig. 22. The entity-set view

[5] In DIAM II [24], n-ary relationships may be treated as special cases of identifiers.

المنارة للاستشارات

THE ENTITY-RELATIONSHIP MODEL TERMINOLOGY	ATTRIBUTE OR ROLE	VALUE SET	VALUE
THE ENTITY SET MODEL TERMINOLOGY	"ROLE-NAME"	"ENTITY-NAME-SET-NAME"	"ENTITY-NAME"
	IDENTIFIER	EMPLOYEE-NO	2566
	NAME	NAME	PETER JONES
	NAME	NAME	SAM JONES
	AGE	NO-OF-YEARS	25
	DEPARTMENT	DEPARTMENT-NO	405

Fig. 23. An "entity description" in the entity-set model

4.3.2 Deriving the Entity Set View. One of the main difficulties in understanding the entity set model is due to its world view (i.e. identifying values with entities). The entity-relationship model proposed in this paper is useful in understanding and deriving the entity set view of data. Consider Figures 2 and 6. In Figure 2, entities are represented by e_i's (which exist in our minds or are pointed at with fingers). In Figure 6, entities are represented by values. The entity set model works both at level 1 and level 2, but we shall explain its view at level 2 (Figure 6). The entity set model treats all value sets such as NO-OF-YEARS as "entity-name-sets" and all values as "entity-names." The attributes become role names in the entity set model. For binary relationships, the translation is simple: the role of an entity in a relationship (for example, the role of "DEPARTMENT" in the relationship DEPARTMENT-EMPLOYEE) becomes the role name of the entity in describing the other entity in the relationship (see Figure 22). For n-ary ($n > 2$) relationships, we must create artificial entities for relationships in order to handle them in a binary relationship world.

ACKNOWLEDGMENTS

The author wishes to express his thanks to George Mealy, Stuart Madnick, Murray Edelberg, Susan Brewer, Stephen Todd, and the referees for their valuable sug-

المنارة للاستشارات

gestions (Figure 21 was suggested by one of the referees). This paper was motivated by a series of discussions with Charles Bachman. The author is also indebted to E.F. Codd and M.E. Senko for their valuable comments and discussions in revising this paper.

REFERENCES

1. Abrial, J.R. Data semantics. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 1–60.
2. Bachman, C.W. Software for random access processing. *Datamation 11* (April 1965), 36–41.
3. Bachman, C.W. Data structure diagrams. *Data Base 1*, 2 (Summer 1969), 4–10.
4. Bachman, C.W. Trends in database management—1975. Proc., AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 569–576.
5. Birkhoff, G., and Bartee, T.C. *Modern Applied Algebra*. McGraw-Hill, New York, 1970.
6. Chamberlin, D.D., and Raymond, F.B. SEQUEL: A structured English query language. Proc. ACM-SIGMOD 1974, Workshop, Ann Arbor, Michigan, May, 1974.
7. CODASYL. Data base task group report. ACM, New York, 1971.
8. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
9. Codd, E.F. Normalized data base structure: A brief tutorial. Proc. ACM-SIGFIDET 1971, Workshop, San Diego, Calif., Nov. 1971, pp. 1–18.
10. Codd, E.F. A data base sublanguage founded on the relational calculus. Proc. ACM-SIGFIDET 1971, Workshop, San Diego, Calif., Nov. 1971, pp. 35–68.
11. Codd, E.F. Recent investigations in relational data base systems. Proc. IFIP Congress 1974, North-Holland Pub. Co., Amsterdam, pp. 1017–1021.
12. Deheneffe, C., Hennebert, H., and Paulus, W. Relational model for data base. Proc. IFIP Congress 1974, North-Holland Pub. Co., Amsterdam, pp. 1022–1025.
13. Dodd, G.G. APL—a language for associate data handling in PL/I. Proc. AFIPS 1966 FJCC, Vol. 29, Spartan Books, New York, pp. 677–684.
14. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for data base integrity. Proc. Very Large Data Base Conf., Framingham, Mass., Sept. 1975, pp. 48–68.
15. Hainaut, J.L., and Lecharlier, B. An extensible semantic model of data base and its data language. Proc. IFIP Congress 1974, North-Holland Pub. Co., Amsterdam, pp. 1026–1030.
16. Hammer, M.M., and McLeod, D.J. Semantic integrity in a relation data base system. Proc. Very Large Data Base Conf., Framingham, Mass., Sept. 1975, pp. 25–47.
17. Lindgreen, P. Basic operations on information as a basis for data base design. Proc. IFIP Congress 1974, North-Holland Pub. Co., Amsterdam, pp. 993–997.
18. Mealy, G.H. Another look at data base. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 525–534.
19. Nijssen, G.M. Data structuring in the DDL and the relational model. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 363–379.
20. Olle, T.W. Current and future trends in data base management systems. Proc. IFIP Congress 1974, North-Holland Pub. Co., Amsterdam, pp. 998–1006.
21. Roussopoulos, N., and Mylopoulos, J. Using semantic networks for data base management. Proc. Very Large Data Base Conf., Framingham, Mass., Sept. 1975, pp. 144–172.
22. Rustin, R. (Ed.). Proc. ACM-SOGMOD 1974—debate on data models. Ann Arbor, Mich., May 1974.
23. Schmid, H.A., and Swenson, J.R. On the semantics of the relational model. Proc. ACM-SIGMOD 1975, Conference, San Jose, Calif., May 1975, pp. 211–233.
24. Senko, M.E. Data description language in the concept of multilevel structured description: DIAM II with FORAL. In *Data Base Description*, B.C.M. Dougue, and G.M. Nijssen, Eds., North-Holland Pub. Co., Amsterdam, pp. 239–258.

المنارة للاستشارات

25. SENKO, M.E., ALTMAN, E.B., ASTRAHAN, M.M., AND FEHDER, P.L. Data structures and accessing in data-base systems. *IBM Syst. J. 12*, 1 (1973), 30–93.
26. SIBLEY, E.H. On the equivalence of data base systems. Proc. ACM-SIGMOD 1974 debate on data models, Ann Arbor, Mich., May 1974, pp. 43–76.
27. STEEL, T.B. Data base standardization—a status report. Proc. ACM-SIGMOD 1975, Conference, San Jose, Calif., May 1975, pp. 65–78.
28. STONEBRAKER, M. Implementation of integrity constraints and views by query modification. Proc. ACM-SIGMOD 1975, Conference, San Jose, Calif., May 1975, pp. 65–78.
29. SUNDGREN, B. Conceptual foundation of the infological approach to data bases. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 61–95.
30. TAYLOR, R.W. Observations on the attributes of database sets. In *Data Base Description*, B.C.M. Dougue and G.M. Nijssen, Eds., North-Holland Pub. Co., Amsterdam, pp. 73–84.
31. TSICHRITZIS, D. A network framework for relation implementation. In *Data Base Description*, B.C.M. Douge and G.M. Nijssen, Eds., North-Holland Pub. Co., Amsterdam, pp. 269–282.

المنارة للاستشارات

Ole-Johan Dahl

Ole-Johan Dahl, Kristen Nygaard

Class and Subclass Declarations

Simulation Programming Languages, ed. By J.N. Buxton,
North Holland, Amsterdam, 1967
pp. 158–174

المنارة للاستشارات

CLASS AND SUBCLASS DECLARATIONS

OLE-JOHAN DAHL and KRISTEN NYGAARD
Norwegian Computing Center, Oslo, Norway

1. INTRODUCTION

A central idea of some programming languages [28,57,58] is to provide protection for the user against (inadvertantly) making meaningless data references. The effects of such errors are implementation dependent and can not be determined by reasoning within the programming language itself. This makes debugging difficult and impractical.

Security in this sense is particularly important in a list processing environment, where data are dynamically allocated and de-allocated, and the user has explicit access to data addresses (pointers, reference values, element values). To provide security it is necessary to have an automatic de-allocation mechanism (reference count, garbage collection). It is convenient to restrict operations on pointers to storage and retrieval. New pointer values are generated by allocation of storage space, pointing to the allocated space. The problem remains of correct interpretation of data referenced relative to user specified pointers, or checking the validity of assumptions inherent in such referencing. E.g. to speak of "*A* of *X*" is meaningful, only if there is an *A* among the data pointed to by *X*.

The record concept proposed by Hoare and Wirth [58] provides full security combined with good runtime efficiency. Most of the necessary checking can be performed at compile time. There is, however, a considerable expense in flexibility. The values of reference variables and procedures must be restricted by declaration to range over records belonging to a stated class. This is highly impractical.

The connection mechanism of SIMULA combines full security with greater flexibility at a certain expense in convenience and run time efficiency. The user is forced, by the syntax of the connection statement, to determine at run time the class of a referenced data structure (process) before access to the data is possible.

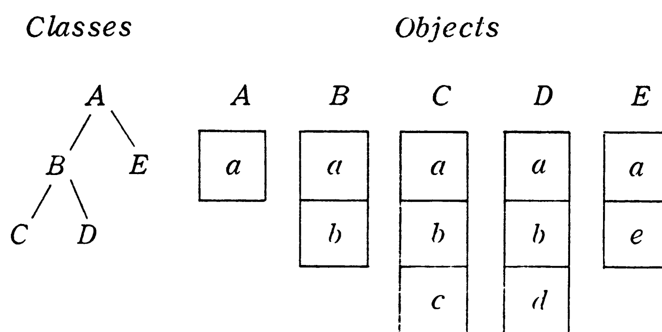
The subclass concept of Hoare [59] is an attempt to overcome the difficulties mentioned above, and to facilitate the manipulation of data structures, which are partly similar, partly distinct. This paper presents another approach to subclasses, and some applications of this approach.

المنارة للاستشارات

2. CLASSES

The class concept introduced is a remodelling of the record class concept proposed by Hoare. The notation is an extension of the ALGOL 60 syntax. A prefix notation is introduced to define subclasses organized in a hierarchical tree structure. The members of a class are called objects. Objects belonging to the same class have similar data structures. The members of a subclass are compound objects, which have a prefix part and a main part. The prefix part of a compound object has a structure similar to objects belonging to some higher level class. It can itself be a compound object.

The figure below indicates the structure of a class hierarchy and of the corresponding objects. A capital letter denotes a class. The corresponding lower case letter denotes the data comprising the main part of an object belonging to that class.



B, C, D, E are subclasses of *A*; *C* and *D* are sublasses of *B*.

2.1. *Syntax*

⟨class id.⟩ :: = ⟨identifier⟩
⟨prefix⟩ :: = ⟨class id.⟩
⟨class body⟩ :: = ⟨statement⟩
⟨main part⟩ :: = class ⟨class id.⟩ ⟨formal parameter part⟩;
⟨specification part⟩ ⟨class body⟩
⟨class declaration⟩ :: = ⟨main part⟩ | ⟨prefix⟩ ⟨main part⟩

2.2. *Semantics*

An object is an instance of a class declaration. Different instances of the same declaration are said to belong to class C, where C is the class identifier. If the class body does not take the form of an unlabelled block, it acts as if enclosed in an implicit block. The parameters and the quantities declared local to the outermost block

المنـارة للاستشارات

of the class body are called the attributes of an object. The attributes can be referenced locally from within the class body, or non-locally by a mechanism called remote acessing (5).

The parameters are transmitted by value. One possible use of the statements of the class body may be to initialize attribute values.

A prefixed class declaration represents the result of concatenating the declaration referenced by the prefix and the main part. The concatenation is recursively defined by the following rules.

1) The formal parameter lists of the former and the latter are concatenated to form one parameter list.

2) The specification parts are juxtaposed.

3) A combined class body is formed, which is a block, whose block head contains the attribute declarations of the prefix body and the main body. The block tail contains the statements of the prefix body followed by those of the main body.

The attributes of the main part are not accessible from within the prefix body, except by remote accessing. The attributes of the prefix are acessible as ordinary local quantities from within the body of the main part.

The object class represented by a prefixed class declaration is a subclass of the class denoted by the prefix. Subclasses can be nested to any depth by using prefixed class identifiers as prefixes to other class declarations.

Let A_0 be any class. If A_0 is prefixed, we will denote this prefix by A_1. The prefix of A_1 (if any) will be denoted by A_2 etc. The sequence

$$A_1, A_2, \ldots$$

will be called the "prefix sequence" of A_0. It follows from the syntax that if A_i and A_j both have A_k as prefix, they have identical prefix sequences.

It will be required that all prefix sequences are finite. (This excludes multiple occurrence of any class A_i in a prefix sequence.) Let

$$A_1, A_2 \ldots, A_n$$

be the prefix sequence of A_0. We shall say that the class A_i is "included in A_j" if $0 \leqslant i \leqslant j \leqslant n$.

المنارة للاستشارات

3. OBJECT REFERENCES

Reference values in the sense of [59] are introduced, in a slightly modified form.

3.1. *Reference types*

3.1.1. *Syntax*

⟨type⟩ :: = ⟨ALGOL type⟩ | ref | ref ⟨qualification⟩
⟨qualification⟩ :: = (⟨class id.⟩)

3.1.2. *Semantics*

Associated with each object is a unique value of type ref, which is said to reference or point to the object. A reference value may, by qualifying a declaration or specification by a class identifier, be required to refer to objects belonging to either this class or any of its subclasses. In addition the value of any item of type reference is restricted to objects belonging to classes whose declarations are statically visible from the declaration or specification of the item.

The reference value none is a permissible value for any reference item, regardless of its qualification.

3.2. *Reference Expressions*

3.2.1. *Syntax*

⟨simple ref. expr.⟩ :: = none | ⟨variable⟩ | ⟨function designator⟩ | ⟨object designator⟩ | ⟨local reference⟩
⟨ref. expr.⟩ :: = ⟨simple ref. expr.⟩ | if ⟨Boolean expr.⟩ then ⟨simple ref. expr.⟩ else ⟨ref. expr.⟩
⟨object designator⟩ :: = ⟨class id.⟩ ⟨actual parameter part⟩
⟨local reference⟩ :: = this ⟨class id.⟩

3.2.2. *Semantics*

A reference expression is a rule for computing a reference value. Thereby reference is made to an object, except if the value is none, which is a reference to "no object".

i) *Qualification.* A variable or function designator is qualified according to its declaration or specification. An object designator or local reference is qualified by the stated class identifier. The expression none is not qualified.

No qualification will be regarded as qualification by a universal class, which includes all declared classes.

المنارة للاستشارات

ii) *Object generation.* As the result of evaluating an object designator an object of the stated class is generated. The class body is executed. The value of the object designator is a reference to the generated object. The life span of the object is limited by that of its reference value.

iii) *Local reference.* A local reference "this *C*" is a meaningful expression within the class body of the class *C* or of any subclass of *C*. Its value is a reference to the current instance of the class declaration (object).

Within a connection block (5.2) connecting an object of class *C* or a subclass of *C* the expression "this *C*" is a reference to the connected object.

The general rule is that a local reference refers to the object, whose attributes are local to the smallest enclosing block, and which belongs to a class included in the one specified. If there is no such object, the expression is illegal.

4. REFERENCE OPERATIONS

4.1. *Assignment*

4.1.1. *Syntax*

⟨reference assignment⟩ :: = ⟨variable⟩: = ⟨reference expr.⟩ |
⟨variable⟩: = ⟨reference assignment⟩

4.1.2. *Semantics*

Let the left and right hand sides be qualified by *Cl* and *Cr*, respectively, and let the value of the right hand side be a reference to an object of class *Cv*. The legality and effect of the statement depends on the relations that hold between these classes.

Case 1. *Cl* includes *Cr*: The statement is legal, and the assignment is carried out.

Case 2. *Cl* is a subclass of *Cr*: The statement is legal, and the assignment is carried out if *Cl* includes *Cv*, or if the value is none. If *Cl* does not include *Cv*, the effect of the statement is undefined (cf. 6.1).

Case 3. *Cl* and *Cr* satisfy neither of the above relations: The statement is illegal.

The following additional rule is considered: The statement is legal only if the declaration of the left hand item (variable, array or ⟨type⟩ procedure) is within the scope of the class identifier *Cr* and all its subclasses. (The scope is in this case defined after having effected all concatenations implied by prefixes.)

المنارة للاستشارات

This rule would have the following consequences.

1) Accessible reference values are limited to pointers to objects, whose attributes are accessible by remote referencing (5).

2) Classes represented by declarations local to different instances of the same block are kept separate.

3) Certain security problems are simplified.

4.2. *Relations*

4.2.1. *Syntax*

⟨relation⟩:: = ⟨ALGOL relation⟩ |
⟨reference expr.⟩ = ⟨reference expr.⟩ |
⟨reference expr.⟩ ≠ ⟨reference expr.⟩ |
⟨reference expr.⟩ is ⟨class id.⟩

4.2.2. *Semantics*

Two reference values are said to be equal if the point to the same object, or if both are none. A relation "X is C" is true if the object referenced by X belongs to the class C or to any of its subclasses.

4.3. *For statements*

4.3.1. *Syntax*

⟨for list element⟩:: = ⟨ALGOL for list element⟩ | ⟨reference expr.⟩ |
⟨reference expr.⟩ while ⟨Boolean expr.⟩

4.3.2. *Semantics*

The extended for statement will facilitate the scanning of list structures.

5. ATTRIBUTE REFERENCING

An attribute of an object is identified completely by the following items of information:

1) the value of a ⟨reference expr.⟩ identifying an object,

2) a ⟨class id.⟩ specifying a class, which includes that of the object, and

3) the ⟨identifier⟩ of an attribute declared for objects of the stated class.

The class identification, item 2, is implicit at run time in a reference value, however, in order to obtain runtime efficiency

المنارة للاستشارات

it is necessary that this information is available to the compiler.

For a local reference to an attribute, i.e. a reference from within the class body, items 1 and 2 are defined implicitly. Item 1 is a reference to the current instance (i.e. object), and item 2 is the class identifier of the class declaration.

Non-local (remote) referencing is either through remote identifiers or through connection. The former is an adaptation of the technique proposed in [57], the latter corresponds to the connection mechanism of SIMULA [28].

5.1. *Remote Identifiers*

5.1.1. *Syntax*

```
⟨remote identifier⟩ :: = ⟨reference expr.⟩ . ⟨identifier⟩
⟨identifier 1⟩ :: = ⟨identifier⟩ | ⟨remote identifier⟩
```

Replace the meta-variable ⟨identifier⟩ by ⟨identifier 1⟩ at appropriate places of the ALGOL syntax.

5.1.2. *Semantics*

A remote identifier identifies an attribute of an individual object. Item 2 above is defined by the qualification of the reference expression. If the latter has the value none, the meaning of the remote identifier is undefined (cf. 6.2).

5.2. *Connection*

5.2.1. *Syntax*

```
⟨connection block 1⟩ :: = ⟨statement⟩
⟨connection block 2⟩ :: = ⟨statement⟩
⟨connection clause⟩ :: = when ⟨class id.⟩ do ⟨connection block 1⟩
⟨otherwise clause⟩ :: = ⟨empty⟩ | otherwise ⟨connection block 2⟩
⟨connection part⟩ :: = ⟨connection clause⟩ |
                        ⟨connection part⟩ ⟨connection clause⟩
⟨connection statement⟩ :: = inspect ⟨reference expr.⟩ do
                             ⟨connection block 2⟩ |
                             inspect ⟨reference expr.⟩
                             ⟨connection part⟩ ⟨otherwise clause⟩
```

5.2.2. *Semantics*

The connection mechanism serves a double purpose:

1) To define item 1 above implicitly for attribute references within connection blocks. The reference expression of a connection statement is evaluated once and its value is stored. Within a connection block this value is said to reference the connected object.

المنارة للاستشارات

It can itself be accessed through a ⟨local reference⟩ (see section 3.2.2).

2) To discriminate on class membership at run time, thereby defining item 2 implicitly for attribute references within alternative connection blocks. Within a ⟨connection block 1⟩ item 2 is defined by the class identifier of the connection clause. Within a ⟨connection block 2⟩ it is defined by the qualification of the reference expression of the connection statement.

Attributes of a connected object are thus immediately accessible through their respective identifiers, as declared in the class declaration corresponding to item 2. These identifiers act as if they were declared local to the connection block. The meaning of such an identifier is undefined, if the corresponding ⟨local reference⟩ has the value none. This can only happen within a ⟨connection block 2⟩.

6. UNDEFINED CASES

In defining the semantics of a programming language the term "undefined" is a convenient stratagem for postponing difficult decisions concerning special cases for which no obvious interpretation exists. The most difficult ones are concerned with cases, which can only be recognized by runtime checking.

One choice is to forbid offending special cases. The user must arrange his program in such a way that they do not occur, if necessary by explicit checking. For security the compiled program must contain implicit checks, which to some extent will duplicate the former. Failure of a check results in program termination and an error message. The implicit checking thus represents a useful debugging aid, and, subject to the implementor's foresight, it can be turned off for a "bugfree" program (if such a thing exists).

Another choice is to define ad hoc, but "reasonable" standard behaviours in difficult special cases. This can make the language much more easy to use. The programmer need not test explicitly for special cases, provided that the given ad hoc rule is appropriate in each situation. However, the language then has no implicit debugging aid for locating unforeseen special cases (for which the standard rules are not appropriate).

In the preceding sections the term undefined has been used three times in connection with two essentially different special cases.

6.1. *Conflicting reference assignment*

Section 4.1.2, case 2, *C*l does not include *C*v: The suggested standard behaviour is to assign the value none.

المنارة للاستشارات

6.2. *Non-existing attributes*

Sections 5.1.2 and 5.2.2: The evaluation of an attribute reference, whose item 1 is equal to none, should cause an error printout and program termination. Notice that this trap will ultimately catch most unforeseen instances of case 6.1.

7. EXAMPLES

The class and subclass concepts are intended to be general aids to data structuring and referencing. However, certain widely used classes might well be included as specialized features of the programming language.

As an example the classes defined below may serve to manipulate circular lists of objects by standard procedures. The objects of a list may have different data structures. The "element" and "set" concepts of SIMULA will be available as special cases in a slightly modified form.

```
class linkage; begin ref (linkage) suc, pred; end linkage;
linkage class link; begin
   procedure out; if suc ≠ none then
      begin pred. suc: = suc; suc. pred: = pred;
               suc: = pred: = none end out;
   procedure into (L); ref (list) L;
      begin if suc ≠ none then out;
               suc: = L; pred: = suc. pred;
               suc. pred: = pred. suc: = this linkage end into;
   end link;
linkage class list;
   begin suc: = pred: = this linkage end list;
```

Any object prefixed by "link" can go in and out of circular lists. If X is a reference expression qualified by link or a subclass of link, whose value is different from none, the statements

$$X.\ \text{into}\ (L) \qquad \text{and} \qquad X.\ \text{out}$$

are meaningful, where L is a reference to a list.

Examples of user defined subclasses are:

```
link class car (license number, weight);
   integer license number; real weight; . . . ;
car class truck (load); ref (list) load; . . . ;
car class bus (capacity); integer capacity;
   begin ref (person) array passenger [1 : capacity] . . . end;
list class bridge; begin real load; . . . end;
```

المنارة للاستشارات

Multiple list memberships may be implemented by means of auxiliary objects.

```
link class element (X); ref X;;
```

A circular list of element objects is analogous to a "set" in SIMULA. The declaration "set S" of SIMULA is imitated by "ref (list) S" followed by the statement "S: = list".

The following are examples of procedures closely similar to the corresponding ones of SIMULA.

```
procedure include (X, S); value X; ref X; ref (list) S;
if X ≠ none then element (X). into (S);
ref (linkage) procedure suc (X); value X; ref (linkage) X;
   suc: = if X ≠ none then X. suc else none;
ref (link) procedure first (S); ref (list) S;
   first: = S. suc;
Boolean procedure empty (S); value S; ref (list) S;
   empty: = S. suc = S;
```

Notice that for an empty list S "suc (S)" is equal to S, whereas "first (S)" is equal to none. This is a result of rule 6.1 and the fact that the two functions have different qualifications.

8. EXTENSIONS

8.1. *Prefixed Blocks*

8.1.1. *Syntax*

```
⟨prefixed block⟩:: = ⟨block prefix⟩ ⟨main block⟩
⟨block prefix⟩:: = ⟨object designator⟩
⟨main block :: = ⟨unlabelled block⟩
⟨block⟩ :: = ⟨ALGOL block⟩ | ⟨prefixed block⟩
             ⟨label⟩:⟨prefixed block⟩
```

8.1.2. *Semantics*

A prefixed block is the result of concatenating (2.2) an instance of a class declaration and the main block. The formal parameters of the former are given initial values as specified by the actual parameters of the block prefix. The latter are evaluated at entry into the prefixed block.

8.2. *Concatenation*

The following extensions of the concepts of class body and concatenation give increased flexibility.

المنارة للاستشارات

8.2.1. *Syntax*

⟨class body⟩:: = ⟨statement⟩|⟨split body⟩
⟨split body⟩:: = ⟨block head⟩;⟨part 1⟩ inner; ⟨part 2⟩
⟨part 1⟩ :: = ⟨empty⟩|⟨statement⟩;⟨part 1⟩
⟨part 2⟩:: = ⟨compound tail⟩

8.2.2. *Semantics*

If the class body of a prefix is a split body, concatenation is defined as follows: the compound tail of the resulting class body consists of part 1 of the prefix body, followed by the statements of the main body, followed by part 2 of the prefix body. If the main body is a split body, the result of the concatenation is itself a split body.

For an object, whose class body is a split body, the symbol inner represents a dummy statement. A class body must not be a prefixed block.

8.3. *Virtual quantities*

The parameters to a class declaration are called by value. Call by name is difficult to implement with full security and good efficiency. The main difficulty is concerned with the definition of the dynamic scope of the actual parameter corresponding to the formal name parameter. It is felt that the cost of an unrestricted call by name mechanism would in general be out of proportion to its gain.

The virtual quantities described below represent another approach to call by name in class declarations. The mechanism provides access at one prefix level of the prefix sequence of an object to quantities declared local to the object at lower prefix levels.

8.3.1. *Syntax*

⟨class declaration⟩:: = ⟨prefix⟩⟨class declarator⟩⟨class id.⟩
⟨formal parameter part⟩;
⟨specification part⟩⟨virtual part⟩
⟨class body⟩
⟨virtual part⟩:: = ⟨empty⟩|virtual: ⟨specification part⟩

8.3.2. *Semantics*

The identifiers of a ⟨virtual part⟩ should not otherwise occur in the heading or in the block head of the class body. Let $A_1, \ldots, A_n$ be the prefix sequence of A_0 and let X be an identifier occurring in the ⟨virtual part⟩ of A_i. If X identifies a parameter of A_j or a quantity declared local to the body of A_j, $j < i$, then for an object of class A_0 identity is established between the virtual quantity X and the quantity X local to A_j.

المنارة للاستشارات

If there is no A_j, $j < i$, for which X is local, a reference to the virtual quantity X of the object constitutes a run time error (in analogy with 6.2).

8.3.3. *Example*

```
class A; virtual: real X, Y,Z;. . . ;
A class B(X,Y); real X,Y;. . . ;
A class C(Y, Z); real Y,Z;. . . ;
A class D(Z, X); real Z,X;. . . ;
ref (A) Q;
```

The attribute reference $Q.X$ is meaningful if Q refers to an object of class B or D. Notice that all three subclasses contain objects with only two attributes.

8.4. *Example*

As an example on the use of the extended class concept we shall define some aspects of the SIMULA concepts "process", "main program", and "SIMULA block".

Quasi-parallel sequencing is defined in terms of three basic procedures, which operate on a system variable SV. SV is an implied and hidden attribute of every object, and may informally be characterized as a variable of "type label". Its value is either null or a program point [5]. SV of a class object initially contains the "exit" information which refers back to the object designator. SV of a prefixed block has the initial value null. The three basic procedures are:

1) detach. The value of SV is recorded, and a new value, called a reactivation point, is assigned referring to the next statement in sequence. Control proceeds to the point referenced by the old value of SV. The effect is undefined if the latter is null.

2) resume(X); ref X. A new value is assigned to SV referring to the next statement in sequence. Control proceeds to the point referenced by SV of the object X. The effect is undefined if $X.SV$ is null or if X is none. null is assigned to $X.SV$.

3) goto(X); ref X. Control proceeds to the point referenced by SV of the object X. The effect is undefined if $X.SV$ is null or if X is none. null is assigned to $X.SV$.

```
class SIMULA; begin
   ref(process)current;
   class process; begin ref(process)nextev; real evtime;
      detach; inner; current: =nextev; goto(nextev)end;
      procedure schedule(X,T); ref(process)X; real T;
            begin X.evtime: = T; ------------ end;
```

المنارة للاستشارات

```
process class main program; begin
                   L: resume(this SIMULA); go to L end;
schedule(main program, 0)end SIMULA;
```

The "sequencing set" of SIMULA is here represented by a simple chain of processes, starting at "current", and linked by the attribute "nextev". The "schedule" procedure will insert the referenced process at the correct position in the chain, according to the assigned time value. The details have been omitted here.

The "main program" object is used to represent the SIMULA object within its own sequencing set.

Most of the sequencing mechanisms of SIMULA can, except for the special syntax, be declared as procedures local to the SIMULA class body.

Examples:

```
procedure passivate; begin current: =current. nextev;
                                       resume(current)end;
procedure activate(X); ref X; inspect X when process do
    if nextev = none then
    begin nextev: =current; evtime: =current. evtime;
          current: =this process; resume(current)end;
procedure hold(T); real T; inspect current do
    begin current: =nextev; schedule(this process, evtime+T);
          resume(current)end;
```

Notice that the construction "process class" can be regarded as a definition of the symbol "activity" of SIMULA. This definition is not entirely satisfactory, because one would like to apply the prefix mechanism to the activity declarations themselves.

9. CONCLUSION

The authors have for some time been working on a new version of the SIMULA language, tentatively named SIMULA 67. A compiler for this language is now being programmed and others are planned. The first compiler should be working by the end of this year.

As a part of this work the class concept and the prefix mechanism have been developed and explored. The original purpose was to create classes and subclasses of data structures and processes. Another useful possibility is to use the class concept to protect whole families of data, procedures, and subordinate classes. Such families can be called in by prefixes. Thereby language "dialects" oriented towards special problem areas can be defined in a convenient way. The administrative problems in making user defined classes generally available are important and should not be overlooked.

المنارة للاستشارات

Some areas of application of the class concept have been illustrated in the preceding sections, others have not yet been explored. An interesting area is input/output. In ALGOL the procedure is the only means for handling I/0. However, a procedure instance is generated by the call, and does not survive this call. Continued existence, and existence in parallel versions is wanted for buffers and data defining external layout, etc. System classes, which include the declarations of local I/0 procedures, may prove useful.

The SIMULA 67 will be frozen in June this year, and the current plan is to include the class and reference mechanisms described in sections 2-6. Class prefixes should be permitted for activity declarations. The "element" and "set" concepts of SIMULA will be replaced by appropriate system defined classes. Additional standard classes may be included.

SIMULA is a true extension of ALGOL 60. This property will very probably be preserved in SIMULA 67.

DISCUSSION

Garwick:

This language has been designed with a very specific line of thought just as GPL has been designed with a very specific line. Dahl's line is different from mine. His overriding consideration has been security. My effort has always been security but not to the same degree. I think that Dahl has gone too far in this respect and thereby lost quite a number of facilities, especially a thing like the "call by name". He can of course use a reference to a variable; this corresponds very closely to the FORTRAN type of "call by address", as opposed to the call by name in ALGOL and so for instance he can not use Jensens device. As you know in GPL, I use pointers. A pointer is not the same as a reference; it is a more general concept. So I think the loss of facilities here is a little too much to take for the sake of security.

The "virtuals" seem to be very closely corresponding to the "externals" in FORTRAN or assembly languages. But you see first of all you can only access things which belong to the same complex structure and secondly it seems to me that it is pretty hard to get type declarations for these procedures. You have to have declared the type of the value of the procedure and the type of parameters. In the example given the procedures seem to be parameterless and they do not deliver any value for the function. So I would like to know how Dahl would take care of that.

المنارة للاستشارات

Dahl:

We think of SIMULA as an extension of ALGOL 60. We therefore provide exactly the same kind of specification for a virtual quantity as you would do for a formal parameter. You can write procedure P; real procedure Q; array A; and so forth.

I would much have preferred to specify the formal parameters of P within the virtual specification of P itself. Then, of course, alternative actual declarations in subclasses could have been simplified by omitting much of the procedure heading. This would have made it possible to check at compile time the actual parameters of a call for a virtual procedure. But in order to be consistent with ALGOL 60, we decided not to do it in this way.

The virtual quantities are in many ways similar to ALGOL's name parameters, but not quite as powerful. It turns out that there is no analogy to Jensen's device. This, I feel, is a good thing, because I hate to implement Jensen's device. It is awful.

If you specify a virtual real X, then you have the option to provide an actual declaration real X in a subclass. But you cannot declare a real expression for X. So, if you specify a quantity which looks like a variable, you can only provide an actual quantity which is a variable. This concept seems more clean to me than the call by name of ALGOL.

To begin with, the whole concept of virtual variables seemed to be superfluous because there was nothing more to say about a virtual variable than what had already been said in the specification. But there is: you can say whether or not it actually exists. A virtual variable X takes no space in the data record of an object if there is no actual declaration of X at any subclass level of the object. Therefore you can use the device for saving space, or for increasing the flexibility in attribute referencing without wasting space. If you access any virtual quantity out of turn, the implementation can catch you and give a run time error message. It is a problem similar to the "null" problem.

Strachey:

Supposing you had classes C and D, could you then define procedures P in both and if so, if you defined one in C and one in D, both being called P, which one would win? Do the scopes go the reverse way from the ordinary scopes or do they go the same way?

Dahl:

Thank you for reminding me of the problem which exists here. The concatenation rule states that declarations given at different

المنارة للاستشارات

prefix levels are brought together into a single block head. Name conflicts in a concatenated block head are regarded as errors of the same kind as redeclarations in an ordinary ALGOL block head. However, if there is a "name conflict" between a declared quantity and a virtual one, identity is established between the two, if the declaration and specification "match".

Strachey:

The other thing I was going to ask about is whether you have thought about the question of achieving security, not by making it impossible to refer to any thing which has gone away but by making it impossible to cause anything which is referred to, to go away. That is to say, by keeping an account of the number of pointers or references to each record, which is one of the methods of garbage collection and only letting it go away when this count reaches zero. The curious thing is this is generally faster than garbage collection.

Dahl:

We have made some experiments on that recently which suggest that it may not be faster.

Strachey:

Anyway, have you thought of this as an alternative method for providing security?

Dahl:

Evidently an actual parameter called by name is represented at run-time by a pointer of some kind, and you could achieve security by instructing the garbage collector to follow such pointers in addition to stored reference values. But then the price you pay for the call by name is much higher than for instance in ALGOL, where data referenced by any parameter has to be retained for other reasons. In my view, a call by name mechanism for classes would be a convenient device which would invite a programmer to entirely misuse the computer - by writing programs where no data can ever be de-allocated and without realizing it.

Petrone:

My first question was covered by Strachey but I now have another question which has arisen from his question. I am asking you whether the call by name mechanism was already present in the old SIMULA in the array case. And did you use it in garbage collection on arrays?

المنارة للاستشارات

Dahl:

That is quite correct. There is a pointer from the object to the array, and the garbage collector will follow it. The reason why we did that is that an array is usually a big thing, which it is reasonable to regard as a separate object.

It is not reasonable to give a small thing like a real variable an independent existence, because that may cause very severe fragmentation of the store. Fragmentation is a disaster if you do not have a compacting scheme, and if you have one the fragmentation will tend to increase the time for each garbage collection and also the frequency of calling for it.

Petrone:

Your concatenation mechanism expresses the possibility of generating families of activity declarations - I am speaking now in terms of your old SIMULA - and the virtual mechanism seems to be a restricted call by name of quantities declared within such a family. Maybe it would be better to restrict the call by name to within an activity block, so that an activity block is equivalent to an ALGOL program with the full call by name mechanism available for procedures.

Dahl:

SIMULA in new and old versions has the complete call by name mechanism for parameters to procedures. You could also have name parameters to classes at no extra cost if you restricted any actual parameter called by name to be computable within the block enclosing the referenced class declaration. That is, it must only reference quantities which are local to that block or to outer blocks. But this is a rather unpleasant restriction considering that an actual parameter may be part of a generating expression occurring deep down in a block hierarchy.

المنارة للاستشارات

Tom DeMarco

Structure Analysis and System Specification

Yourdon, New York 1978
pp. 1–17 and 37–44

المنارة للاستشارات

STRUCTURED ANALYSIS

AND

SYSTEM SPECIFICATION

by

Tom DeMarco

Foreword by

P.J. Plauger

YOURDON inc.
1133 Avenue of the Americas
New York, New York 10036

المنارة للاستشارات

First Printing, March 1978

Second Printing, June 1978

Revised, December 1978

Copyright © 1978, 1979 by YOURDON inc., New York, N.Y. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, record, or otherwise, without the prior written permission of the publisher. *Library of Congress Catalog Card Number 78-51285.*

ISBN 978-3-540-42290-7 ISBN 978-3-642-48354-7 (eBook)
DOI 10.1007/978-3-642-48354-7

المنارة للاستشارات

CONTENTS

PAGE

PART 1: BASIC CONCEPTS

1. The Meaning of Structured Analysis 3

1.1 What is analysis? 4
1.2 Problems of analysis 9
1.3 The user-analyst relationship 14
1.4 What is Structured Analysis? 15

2. Conduct of the analysis phase 19

2.1 The classical project life cycle 19
2.2 The modern life cycle 22
2.3 The effect of Structured Analysis on the life cycle 25
2.4 Procedures of Structured Analysis 27
2.5 Characteristics of the Structured Specification 31
2.6 Political effects of Structured Analysis 32
2.7 Questions and answers 35

3. The Tools of Structured Analysis 37

3.1 A sample situation 37
3.2 A Data Flow Diagram example 38
3.3 A Data Dictionary example 42
3.4 A Structured English example 43
3.5 A Decision Table example 44
3.6 A Decision Tree example 44

PART 2: FUNCTIONAL DECOMPOSITION

4. Data Flow Diagrams 47

4.1 What is a Data Flow Diagram? 47
4.2 A Data Flow Diagram by any other name . . . 48
4.3 DFD characteristics — inversion of viewpoint 48

5. Data Flow Diagram conventions 51

5.1 Data Flow Diagram elements 51
5.2 Procedural annotation of DFD's 61
5.3 The Lump Law 62

المنارة للاستشارات

6. Guidelines for drawing Data Flow Diagrams 63

6.1 Identifying net inputs and outputs 63
6.2 Filling in the DFD body 64
6.3 Labeling data flows 66
6.4 Labeling processes 66
6.5 Documenting the steady state 68
6.6 Omitting trivial error-handling details 68
6.7 Portraying data flow and not control flow 68
6.8 Starting over 69

7. Leveled Data Flow Diagrams 71

7.1 Top-down analysis — the concept of leveling 72
7.2 Elements of a leveled DFD set 75
7.3 Leveling conventions 77
7.4 Bottom-level considerations 83
7.5 Advantages of leveled Data Flow Diagrams 87
7.6 Answers to the leveled DFD Guessing Game 87

8. A Case Study in Structured Analysis 89

8.1 Background for the case study 89
8.2 Welcome to the project — context of analysis 90
8.3 The top level 91
8.4 Intermezzo: What's going on here? 94
8.5 The lower levels 96
8.6 Summary 104
8.7 Postscript 104

9. Evaluation and Refinement of Data Flow Diagrams 105

9.1 Tests for correctness 105
9.2 Tests for usefulness 112
9.3 Starting over 114

10. Data Flow Diagrams for System Specification 117

10.1 The man-machine dialogue 117
10.2 The integrated top-level approach 118
10.3 Problems and potential problems 120

PART 3: DATA DICTIONARY

11. The analysis phase Data Dictionary 125

11.1 The uses of Data Dictionary 126
11.2 Correlating Data Dictionary to the DFD's 127
11.3 Implementation considerations 127

المنارة للاستشارات

12. Definitions in the Data Dictionary 129

12.1 Characteristics of a definition 129
12.2 Definition conventions 133
12.3 Redundancy in DD definitions 137
12.4 Self-defining terms 139
12.5 Treatment of aliases 142
12.6 What's in a name? 143
12.7 Sample entries by class 144

13. Logical Data Structures 149

13.1 Data base considerations 150
13.2 Data Structure Diagrams (DSD's) 152
13.3 Uses of the Data Structure Diagram 155

14. Data Dictionary Implementation 157

14.1 Automated Data Dictionary 157
14.2 Manual Data Dictionary 162
14.3 Hybrid Data Dictionary 162
14.4 Librarian's role in Data Dictionary 163
14.5 Questions about Data Dictionary 164

PART 4: PROCESS SPECIFICATION

15. Description of Primitives 169

15.1 Specification goals 169
15.2 Classical specification writing methods 177
15.3 Alternative means of specification 177

16. Structured English 179

16.1 Definition of Structured English 179
16.2 An example 180
16.3 The logical constructs of Structured English 184
16.4 The vocabulary of Structured English 202
16.5 Structured English styles 203
16.6 The balance sheet on Structured English 210
16.7 Gaining user acceptance 212

17. Alternatives for Process Specification 215

17.1 When to use a Decision Table 215
17.2 Getting started 217
17.3 Deriving the condition matrix 219
17.4 Combining Decision Tables and Structured English 221
17.5 Selling Decision Tables to the user 221
17.6 Decision Trees 222

المنارة للاستشارات

17.7 A procedural note 225
17.8 Questions and answers 225

PART 5: SYSTEM MODELING

18. Use of System Models 229

18.1 Logical and physical DFD characteristics 230
18.2 Charter for Change 231
18.3 Deriving the Target Document 232

19. Building a Logical Model of the Current System 233

19.1 Use of expanded Data Flow Diagrams 235
19.2 Deriving logical file equivalents 238
19.3 Brute-force logical replacement 254
19.4 Logical DFD walkthroughs 256

20. Building a Logical Model of a Future System 257

20.1 The Domain of Change 258
20.2 Partitioning the Domain of Change 260
20.3 Testing the new logical specification 263

21. Physical Models 265

21.1 Establishing options 265
21.2 Adding configuration-dependent features 269
21.3 Selecting an option 269

22. Packaging the Structured Specification 273

22.1 Filling in deferred details 273
22.2 Presentation of key interfaces 275
22.3 A guide to the Structured Specification 275
22.4 Supplementary and supporting material 278

PART 6: STRUCTURED ANALYSIS FOR A FUTURE SYSTEM

23. Looking Ahead to the Later Project Phases 283

23.1 Analyst roles during design and implementation 283
23.2 Bridging the gap from analysis to design 284
23.3 User roles during the later phases 285

24. Maintaining the Structured Specification 287

24.1 Goals for specification maintenance 287
24.2 The concept of the specification increment 289

المنارة للاستشارات

24.3 Specification maintenance procedures 292
24.4 The myth of Change Control 294

25. Transition into the Design Phase 297

25.1 Goals for design 297
25.2 Structured Design 302
25.3 Implementing Structured Designs 323

26. Acceptance Testing 325

26.1 Derivation of normal path tests 326
26.2 Derivation of exception path tests 328
26.3 Transient state tests 330
26.4 Performance tests 330
26.5 Special tests 331
26.6 Test packaging 331

27. Heuristics for Estimating 333

27.1 The empirically derived estimate 334
27.2 Empirical productivity data 335
27.3 Estimating rules 336

GLOSSARY 341

BIBLIOGRAPHY 347

INDEX 349

المنارة للاستشارات

PART 1

BASIC CONCEPTS

المنارة للاستشارات

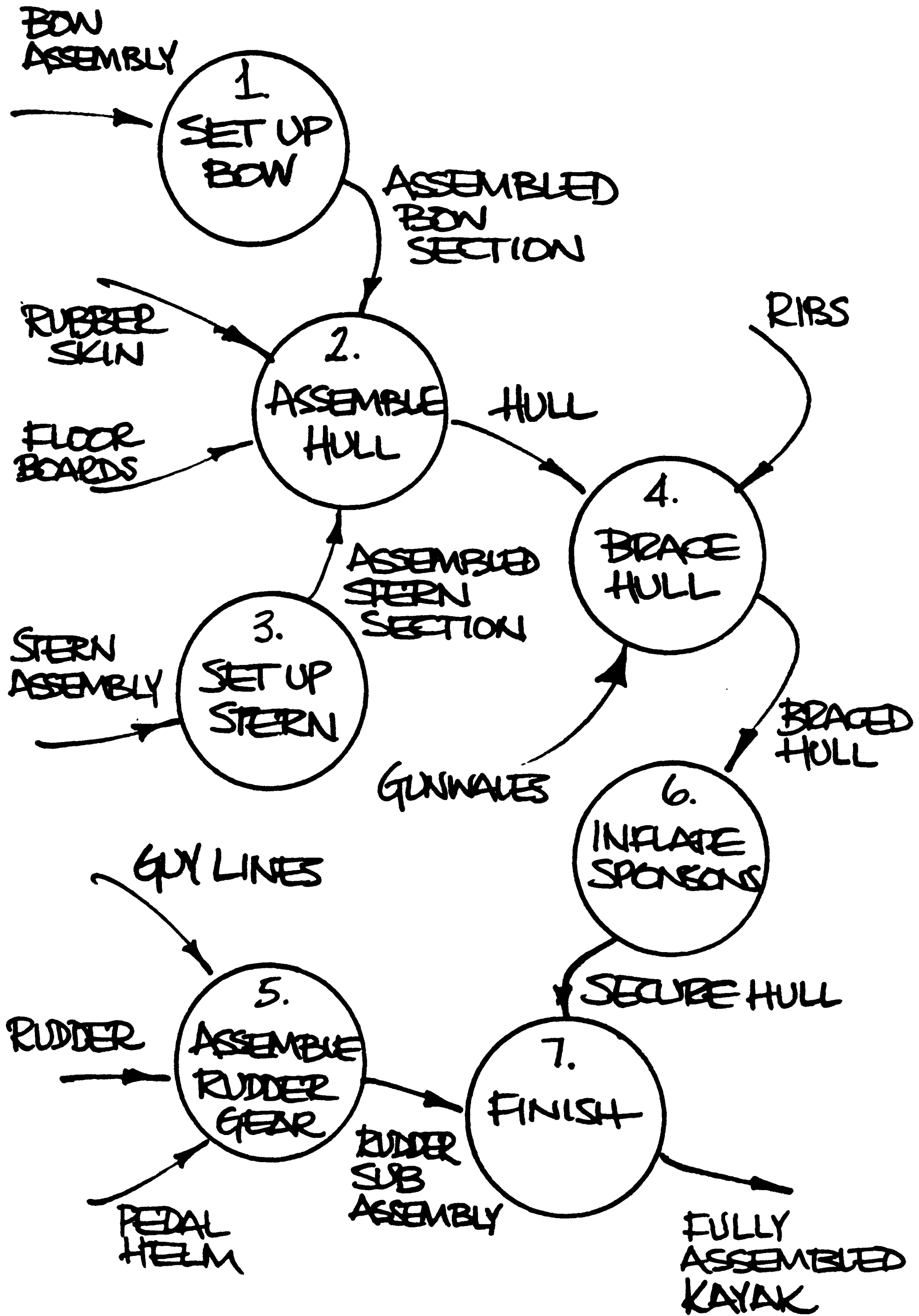


Figure 1

المنارة للاستشارات

1 THE MEANING OF STRUCTURED ANALYSIS

Let's get right to the point. This book is about Structured Analysis, and Structured Analysis is primarily concerned with a new kind of Functional Specification, the Structured Specification. Fig. 1 shows part of a Structured Specification.

The example I have chosen is a system of sorts, but obviously not a computer system. It is, in fact, a manual assembly procedure. Procedures like the one described in Fig. 1 are usually documented by a text narrative. Such descriptions have many of the characteristics of the classical Functional Specifications that we analysts have been producing for the last 20 years. (The Functional Specification describes *automated* procedures — that is the main difference between the two.) Take a look at a portion of the text that prompted me to draw Fig. 1.

Assembly Instructions for KLEPPER Folding Boats

1. Lay out hull in grass (or on carpet). Select a clean, level spot.
2. Take folded bow section (with red dot), lay it in grass, unfold 4 hinged gunwale boards. Kneel down, spread structure lightly with left hand near bow, place right hand on pullplate at *bottom* of hinged rib, and set up rib gently by pulling towards center of boat. Deckbar has a tongue-like fitting underneath which will connect with fitting on top of rib if you lift deckbar lightly, guide tongue to rib, press down on deckbar near bow to lock securely. Now lift whole bowsection using both arms wrap-around style (to keep gunwales from flopping down) and slide into front of hull. Center seam of blue deck should rest on top of deckbar.
3. Take folded stern section (blue dot, 4 "horseshoes" attached), unfold 4 gunwales, set up rib by pulling on pullplate at *bottom* of rib. Deckbar locks to top of rib *from the side* by slipping a snaplock over a tongue attached to top of rib . . .

   And so forth.

The differences are fairly evident: The text plunges immediately into the details of the early assembly steps, while the structured variant tries to present the big picture first, with the intention of working smoothly from abstract to detailed. The Structured Specification is graphic and the text is not. The old-fashioned approach is one-dimensional (written narrative is always one-dimensional), and the structured variant is multidimensional. There are other differences as well; we'll get to those later. My intention here is only to give you an initial glimpse at a Structured Specification.

Now let's go back and define some terms.

المنارة للاستشارات

1.1 What is analysis?

Analysis is the study of a problem, prior to taking some action. In the specific domain of computer systems development, analysis refers to the study of some business area or application, usually leading to the specification of a new system. The action we're going to be taking later on is the implementation of that system.

The most important product of systems analysis — of the analysis phase of the life cycle — is the specification document. Different organizations have different terms for this document: Functional Specification, External Specification, Design Specification, Memo of Rationale, Requirements Document. In order to avoid the slightly different connotations that these names carry, I would like to introduce a new term here: the Target Document. The Target Document establishes the goals for the rest of the project. It says what the project will have to deliver in order to be considered a success. The Target Document is the principal product of analysis.

Successful completion of the analysis phase involves all of the following:

1. selecting an optimal target
2. producing detailed documentation of that target in such a manner that subsequent implementation can be evaluated to see whether or not the target has been attained
3. producing accurate predictions of the important parameters associated with the target, including costs, benefits, schedules, and performance characteristics
4. obtaining concurrence on each of the items above from each of the affected parties

In carrying out this work, the analyst undertakes an incredibly large and diverse set of tasks. At the very minimum, analysts are responsible for: user liaison, specification, cost-benefit study, feasibility analysis, and estimating. We'll cover each of these in turn, but first an observation about some characteristics that are common to all the analyst's activities.

1.1.1 Characteristics of Analysis

Most of us come to analysis by way of the implementation disciplines — design, programming, and debugging. The reason for this is largely historical. In the past, the business areas being automated were the simpler ones, and the users were rather unsophisticated; it was more realistic to train computer people to understand the application than to train users to understand EDP technology. As we come to automate more and more complex areas, and as our users (as a result of prevalent computer training at the high school and college level) come to be more literate in automation technologies, this trend is reversing.

المنارة للاستشارات

But for the moment, I'm sure you'll agree with me that most computer systems analysts are first of all computer people. That being the case, consider this observation: Whatever analysis is, it certainly is not very similar to the work of designing, programming, and debugging computer systems. Those kinds of activities have the following characteristics:

- The work is reasonably straightforward. Software sciences are relatively new and therefore not as highly specialized as more developed fields like medicine and physics.
- The interpersonal relationships are not very complicated nor are there very many of them. I consider the business of building computer systems and getting them to run a rather friendly activity, known for easy relationships.
- The work is very definite. A piece of code, for instance, is either right or wrong. When it's wrong, it lets you know in no uncertain terms by kicking and screaming and holding its breath, acting in obviously abnormal ways.
- The work is satisfying. A positive glow emanates from the programmer who has just found and routed out a bug. A friend of mine who is a doctor told me, after observing programmers in the debugging phase of a project, that most of them seemed "high as kites" much of the time. I think he was talking about the obvious satisfaction programmers take in their work.

The implementation disciplines are straightforward, friendly, definite, and satisfying. Analysis is none of these things:

- It certainly isn't easy. Negotiating a complex Target Document with a whole community of heterogeneous and conflicting users and getting them all to agree is a gargantuan task. In the largest systems for the most convoluted organizations, the diplomatic skills that the analyst must bring to bear are comparable to the skills of a Kissinger negotiating for peace in the Middle East.
- The interpersonal relationships of analysis, particularly those involving users, are complicated, sometimes even hostile.
- There is nothing definite about analysis. It is not even obvious when the analysis phase is done. For want of better termination criteria, the analysis phase is often considered to be over when the time allocated for it is up!
- Largely because it is so indefinite, analysis is not very satisfying. In the most complicated systems, there are so many compromises to be made that no one is ever completely happy with the result. Frequently, the various parties involved in the

المنارة للاستشارات

> negotiation of a Target Document are so rankled by their own concessions, they lose track of what a spectacular feat the analyst has achieved by getting them to agree at all.

So analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating. Once you're hooked, the old easy pleasures of system building are never again enough to satisfy you.

1.1.2 The User Liaison

During the 1960's, our business community saw a rash of conglomerations in which huge corporate monoliths swallowed up smaller companies and tried to digest them. As part of this movement, many centralized computer systems were installed with an aim toward gathering up the reins of management, and thus allowing the conglomerate's directors to run the whole show. If you were an analyst on one of these large Management Information System (MIS) projects, you got to see the user-analyst relationship at its very worst. Users were dead set against their functions being conglomerated, and of course that's just what the MIS systems were trying to do. The philosophy of the 60's was that an adversary relationship between the analyst and the user could be very productive, that analysts could go in, as the representatives of upper management, and force the users to participate and comply.

Of course the record of such projects was absolutely dismal. I know of no conglomerate that made significant gains in centralization through a large Management Information System. The projects were often complete routs. Many conglomerates are now spinning off their acquisitions and finding it rather simple to do so because so little true conglomeration was ever achieved. Due to the experience of the 60's, the term Management Information System, even today, is likely to provoke stifled giggles in a group of computer people.

The lesson of the 60's is that no system is going to succeed without the active and willing participation of its users. Users have to be made aware of how the system will work and how they will make use of it. They have to be sold on the system. Their expertise in the business area must be made a key ingredient to system development. They must be kept aware of progress, and channels must be kept open for them to correct and tune system goals during development. All of this is the responsibility of the analyst. He is the users' teacher, translator, and advisor. This intermediary function is the most essential of all the analyst's activities.

1.1.3 Specification

The analyst is the middleman between the user, who decides what has to be done, and the development team, which does it. He bridges this gap with a Target Document. The business of putting this document together and getting it accepted by all parties is specification. Since the Target Document is the analyst's principal output, specification is the most visible of his activities.

المنارة للاستشارات

If you visit the Royal Naval Museum at Greenwich, England, you will see the results of some of the world's most successful specification efforts, the admiralty models. Before any ship of the line was constructed, a perfect scale model had to be built and approved. The long hours of detail work were more than repaid by the clear understandings that come from studying and handling the models.

The success of the specification process depends on the product, the Target Document in our case, being able to serve as a model of the new system. To the extent that it helps you visualize the new system, the Target Document is the system model.

1.1.4 Cost-Benefit Analysis

The study of relative cost and benefits of potential systems is the feedback mechanism used by an analyst to select an optimal target. While Structured Analysis does not entail new methods for conduct of this study, it nonetheless has an important effect. An accurate and meaningful system model helps the user and the analyst perfect their vision of the new system and refine their estimates of its costs and benefits.

1.1.5 Feasibility Analysis

It is pointless to specify a system which defies successful implementation. Feasibility analysis refers to the continual testing process the analyst must go through to be sure that the system he is specifying can be implemented within a set of given constraints. Feasibility analysis is more akin to design than to the other analysis phase activities, since it involves building tentative physical models of the system and evaluating them for ease of implementation. Again, Structured Analysis does not prescribe new procedures for this activity. But its modeling tools will have some positive effect.

1.1.6 Estimating

Since analysis deals so heavily with a system which exists only on paper, it involves a large amount of estimating. The analyst is forever being called upon to estimate cost or duration of future activities, CPU load factors, core and disk extents, manpower allocation . . . almost anything. I have never heard of a project's success being credited to the fine estimates an analyst made; but the converse is frequently true — poor estimates often lead to a project's downfall, and in such cases, the analyst usually receives full credit.

Estimating is rather different from the other required analysis skills:

- *Nobody is an expert estimator.* You can't even take a course in estimating, because nobody is willing to set himself up as enough of an authority on the subject to teach it.
- *We don't build our estimating skills, because we don't collect any data about our past results.* At the end of a project we rarely go

للاستشارات

back and carry out a thorough postmortem to see how the project proceeded. How many times have you seen project performance statistics published and compared to the original estimates? In my experience, this is done only in the very rare instance of a project that finishes precisely on time and on budget. In most cases, the original schedule has long since vanished from the record and will never be seen again.

- *None of this matters as much as it ought to anyway,* since most things we call "estimates" in computer system projects are not estimates at all. When your manager asks you to come up with a schedule showing project completion no later than June 1 and using no more than six people, you're not doing any real estimating. You are simply dividing up the time as best you can among the phases. And he probably didn't estimate either; chances are his dates and manpower loading were derived from budgetary figures, which were themselves based upon nothing more than Wishful Thinking.

All these factors aside, estimating plays a key part in analysis. There are some estimating heuristics that are a by-product of Structured Analysis; these will be discussed in a subsequent chapter. The key word here is *heuristic.* A heuristic is a cheap trick that often works well but makes no guarantee. It is not an algorithm, a process that leads to a guaranteed result.

1.1.7 The Defensive Nature of Analysis

In addition to the analysis phase activities presented above, there are many others; the analyst is often a project utility infielder, called upon to perform any number of odd jobs. As the project wears on, his roles may change. But the major activities, and the ones that will concern us most in this book, are: user liaison, specification, cost-benefit and feasibility analysis, and estimating.

In setting about these activities, the analyst should be guided by a rule which seems to apply almost universally: *The overriding concern of analysis is not to achieve success, but to avoid failure.* Analysis is essentially a defensive business.

This melancholy observation stems from the fact that the great flaming failures of the past have inevitably been attributable to analysis phase flaws. When a system goes disastrously wrong, it is the analyst's fault. When a system succeeds, the credit must be apportioned among many participants, but failure (at least the most dramatic kind) belongs completely to the analyst. If you think of a system project that was a true rout — years late, or orders of magnitude over budget, or totally unacceptable to the user, or utterly impossible to maintain — it almost certainly was an analysis phase problem that did the system in.

Computer system analysis is like child-rearing; you can do grievous damage, but you cannot ensure success.

المنارة للاستشارات

My reason for presenting this concept here is to establish the following context for the rest of the book: The principal goal of Structured Analysis is to minimize the probability of critical analysis phase error. The tools of Structured Analysis are defensive means to cope with the most critical risk areas of analysis.

1.2 Problems of analysis

Projects can go wrong at many different points: The fact that we spend so much time, energy, and money on maintenance is an indication of our failures as designers; the fact that we spend so much on debugging is an indictment of our module design and coding and testing methods. But analysis failures fall into an entirely different class. When the analysis goes wrong, we don't just spend more money to come up with a desired result — we spend *much* more money, and often don't come up with any result.

That being the case, you might expect management to be super-conservative about the analysis phase of a project, to invest much more in doing the job correctly and thus avoid whole hosts of headaches downstream. Unfortunately, it is not as simple as that. Analysis is plagued with problems that are not going to be solved simply by throwing money at them. You may have experienced this yourself if you ever participated in a project where too much time was allocated to the analysis phase. What tends to happen in such cases is that work proceeds in a normal fashion until the major products of analysis are completed. In the remaining time, the project team spins its wheels, agonizing over what more it could do to avoid later difficulties. When the time is finally up, the team breathes a great sigh of relief and hurries on to design. Somehow the extra time is just wasted — the main result of slowing down the analysis phase and doing everything with exaggerated care is that you just get terribly bored. Such projects are usually every bit as subject to failures of analysis as others.

I offer this list of the major problems of analysis:

1. communication problems
2. the changing nature of computer system requirements
3. the lack of tools
4. problems of the Target Document
5. work allocation problems
6. politics

Before looking at these problems in more detail, we should note that none of them will be *solved* by Structured Analysis or by any other approach to analysis. The best we can hope for is some better means to grapple with them.

المنارة للاستشارات

1.2.1 Communication Problems

A long-unsolved problem of choreography is the development of a rigorous notation to describe dance. Merce Cunningham, commenting on past failures to come up with a useful notation, has observed that the motor centers of the brain are separated from the reading and writing centers. This physical separation in the brain causes communication difficulties.

Computer systems analysis is faced with this kind of difficulty. The business of specification is, for the most part, involved in describing procedure. Procedure, like dance, resists description. (It is far easier to demonstrate procedure than to describe it, but that won't do for our purposes.) Structured Analysis attempts to overcome this difficulty through the use of graphics. When you use a picture instead of text to communicate, you switch mental gears. Instead of using one of the brain's serial processors, its reading facility, you use a parallel processor.

All of this is a highfalutin way to present a "lowfalutin" and very old idea: A picture is worth a thousand words. The reason I present it at all is that analysts seem to need some remedial work on this concept. When given a choice (in writing a Target Document, for instance) between a picture and a thousand words, most analysts opt unfailingly for the thousand words.

Communication problems are exacerbated in our case by the lack of a common language between user and analyst. The things we analysts work with — specifications, data format descriptions, flowcharts, code, disk and core maps — are totally inappropriate for most users. The one aspect of the system the user is most comfortable talking about is the set of human procedures that are his interface to the system, typically something we don't get around to discussing in great detail with him until well after analysis, when the user manuals are being written.

Finally, our communication problem is complicated by the fact that what we're describing is usually a system that exists only in our minds. There is no model for it. In our attempts to flesh out a picture of the system, we are inclined to fill in the physical details (CRT screens, report formats, and so forth) much too early.

To sum it up, the factors contributing to the communication problems of analysis are

1. the natural difficulty of describing procedure
2. the inappropriateness of our method (narrative text)
3. the lack of a common language between analyst and user
4. the lack of any usable early model for the system

المنارة للاستشارات

1.2.2 The Changing Nature of Requirements

I sometimes think managers are sent to a special school where they are taught to talk about "freezing the specification" at least once a day during the analysis phase. The idea of freezing the specification is a sublime fiction. Changes won't go away and they can't be ignored. If a project lasts two years, you ought to expect as many legitimate changes (occasioned by changes in the way business is done) to occur during the project as would occur in the first two years after cutover. In addition to changes of this kind, an equal number of changes may arise from the user's increased understanding of the system. This type of change results from early, inevitable communication failures, failures which have since been corrected.

When we freeze a Target Document, we try to hold off or ignore change. But the Target Document is only an approximation of the true project target; therefore, by holding off and ignoring change, we are trying to proceed toward a target *without benefit of any feedback.*

There are two reasons why managers want to freeze the Target Document. First, they want to have a stable target to work toward, and second, an enormous amount of effort is involved in updating a specification. The first reason is understandable, but the second is ridiculous. *It is unacceptable to write specifications in such a way that they can't be modified.* Ease of modification has to be a requirement of the Target Document.

This represents a change of ground rules for analysis. In the past, it was expected that the Target Document would be frozen. It was a positive advantage that the document was impossible to change since that helped overcome resistance to the freeze. It was considered normal for an analyst to hold off a change by explaining that implementing it in the Target Document would require retyping every page. I even had one analyst tell me that the system, once built, was going to be highly flexible, so that it would be easier to put the requested change into the system itself rather than to put it into the specification!

Figures collected by GTE, IBM, and TRW over a large sample of system changes, some of them incorporated immediately and others deferred, indicate that the difference in cost can be staggering. It can cost two orders of magnitude more to implement a change after cutover than it would have cost to implement it during the analysis phase. As a rule of thumb, you should count on a 2:1 cost differential to result from deferring change until a subsequent project phase.[1]

My conclusion from all of this is that we must change our methods; we must begin building Target Documents that are highly maintainable. In fact, maintainability of the Target Document is every bit as essential as maintainability of the eventual system.

[1]See Barry Boehm's article, "Software Engineering," published in the *IEEE Transactions on Computers,* December 1976, for a further discussion of this topic.

المنارة للاستشارات

1.2.3 The Lack of Tools

Analysts work with their wits plus paper and pencil. That's about it. The fact that you are reading this book implies that you are looking for some tools to work with. For the moment, my point is that most analysts don't have any.

As an indication of this, consider your ability to evaluate the products of each project phase. You would have little difficulty evaluating a piece of code: If it were highly readable, well submodularized, well commented, conformed to generally accepted programming practice, had no GOTO's, ALTER's, or other forms of pathology — you would probably be willing to call it a good piece of code. Evaluating a design is more difficult, and you would be somewhat less sure of your judgment. But suppose you were asked to evaluate a Target Document. Far from being able to judge its quality, you would probably be hard pressed to say whether it qualified as a Target Document at all. Our inability to evaluate any but the most incompetent efforts is a sign of the lack of analysis phase tools.

1.2.4 Problems of the Target Document

Obviously the larger the system, the more complex the analysis. There is little we can do to limit the size of a system; there are, however, intelligent and unintelligent ways to deal with size. An intelligent way to deal with size is to *partition.* That is exactly what designers do with a system that is too big to deal with conveniently — they break it down into component pieces (modules). Exactly the same approach is called for in analysis.

The main thing we have to partition is the Target Document. We have to stop writing Victorian novel specifications, enormous documents that can only be read from start to finish. Instead, we have to learn to develop dozens or even hundreds of "mini-specifications." And we have to organize them in such a way that the pieces can be dealt with selectively.

Besides its unwieldy size, the classical Target Document is subject to further problems:

- It is excessively redundant.
- It is excessively wordy.
- It is excessively physical.
- It is tedious to read and unbearable to write.

1.2.5 Work Allocation

Adding manpower to an analysis team is even more complicated than beefing up the implementation team. The more successful classical analyses are done by very small teams, often only one person. On rush projects, the analysis phase is sometimes shortchanged since people assume it will take forever, and there is no convenient way to divide it up.

المنارة للاستشارات

I think it obvious that this, again, is a partitioning problem. Our failure to come up with an early partitioning of the subject matter (system or business area) means that we have no way to divide up the rest of the work.

1.2.6 Politics

Of course, analysis is an intensely political subject. Sometimes the analyst's political situation is complicated by communication failures or inadequacies of his methods. That kind of problem can be dealt with positively — the tools of Structured Analysis, in particular, will help.

But most political problems do not lend themselves to simple solutions. The underlying cause of political difficulty is usually the changing distribution of power and autonomy that accompanies the introduction of a new system. No new analysis procedures are going to make such an impending change less frightening.

Political problems aren't going to go away and they won't be "solved." The most we can hope for is to limit the effect of disruption due to politics. Structured Analysis approaches this objective by making analysis procedures more formal. To the extent that each of the analyst's tasks is clearly (and publicly) defined, and has clearly stated deliverables, the analyst can expect less political impact from them. Users understand the limited nature of his investigations and are less inclined to overreact. The analyst becomes less of a threat.

1.3 The user-analyst relationship

Since Structured Analysis introduces some changes into the user-analyst relationship, I think it is important to begin by examining this relationship in the classical environment. We need to look at the user's role, the analyst's role, and the division of responsibility between them.

1.3.1 What Is a User?

First of all, there is rarely just one user. In fact, the term "user" refers to at least three rather different roles:

- *The hands-on user*, the operator of the system. Taking an on-line banking system as an example, the hands-on users might include tellers and platform officers.
- *The responsible user*, the one who has direct business responsibility for the procedures being automated by the system. In the banking example, this might be the branch manager.
- *The system owner*, usually upper management. In the banking example, this might be the Vice President of Banking Operations.

المنارة للاستشارات

Sometimes these roles are combined, but most often they involve distinctly different people. When multiple organizations are involved, you can expect the total number of users to be as much as three times the number of organizations.

The analyst must be responsible for communication with *all* of the users. I am continually amazed at how many development teams jeopardize their chances of success by failing to talk to one or more of their users. Often this takes the form of some person or organization being appointed "User Representative." This is done to spare the user the bother of the early system negotiations, and to spare the development team the bother of dealing with users. User Representatives would be fine if they also had authority to accept the system. Usually they do not. When it comes to acceptance, they step aside and let the real user come forward. When this happens, nobody has been spared any bother.

1.3.2 What Is an Analyst?

The analyst is the principal link between the user area and the implementation effort. He has to communicate the requirements to the implementors, and the details of how requirements are being satisfied back to the users. He may participate in the actual determination of what gets done: It is often the analyst who supplies the act of imagination that melds together applications and present-day technology. And, he may participate in the implementation. In doing this, he is assuming the role that an architect takes in guiding the construction of his building.

While the details may vary from one organization to the next, most analysts are required to be

- at ease with EDP concepts
- at ease with concepts particular to the business area
- able to communicate such concepts

1.3.3 Division of Responsibility Between Analyst and User

There is something terribly wrong with a user-analyst relationship in which the user specifies such physical details as hardware vendor, software vendor, programming language, and standards. Equally upsetting is the user who relies upon the analyst to decide how business ought to be conducted. What is the line that separates analyst functions from user functions?

I believe the analyst and the user ought to try to communicate across a "logical-physical" boundary that exists in any computer system project. Logical considerations include answers to the question, *What needs to be accomplished?* These fall naturally into the domain of the user. Physical considerations include answers to the question, *How shall we accomplish these things?* These are in the domain of the analyst.

المنارة للاستشارات

1.4 What is Structured Analysis?

So far, most of what we have been discussing has been the classical analysis phase, its problems and failings. How is Structured Analysis different? To answer that question, we must consider

- New goals for analysis. While we're changing our methods, what new analysis phase requirements shall we consider?
- Structured tools for analysis. What is available and what can be adapted?

1.4.1 New Goals for Analysis

Looking back over the recognized problems and failings of the analysis phase, I suggest we need to make the following additions to our set of analysis phase goals:

- The products of analysis must be highly maintainable. This applies particularly to the Target Document.
- Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out.
- Graphics have to be used wherever possible.
- We have to differentiate between logical and physical considerations, and allocate responsibility, based on this differentiation, between the analyst and the user.
- We have to build a logical system model so the user can gain familiarity with system characteristics before implementation.

1.4.2 Structured Tools for Analysis

At the very least, we require three types of new analysis phase tools:

- Something to help us partition our requirement and document that partitioning before specification. For this I propose we use a *Data Flow Diagram,* a network of interrelated processes. Data Flow Diagrams are discussed in Chapters 4 through 10.
- Some means of keeping track of and evaluating interfaces without becoming unduly physical. Whatever method we select, it has to be able to deal with an enormous flood of detail — the more we partition, the more interfaces we have to expect. For our interface tool I propose that we adopt a set of *Data Dictionary* conventions, tailored to the analysis phase. Data Dictionary is discussed in Chapters 11 through 14.

المنارة للاستشارات

- New tools to describe logic and policy, something better than narrative text. For this I propose three possibilities: *Structured English, Decision Tables,* and *Decision Trees.* These topics are discussed in Chapters 15 through 17.

1.4.3 Structured Analysis — A Definition

Now that we have laid all the groundwork, it is easy to give a working definition of Structured Analysis:

Structured Analysis is the use of these tools:

Data Flow Diagrams
Data Dictionary
Structured English
Decision Tables
Decision Trees

to build a new kind of Target Document, the Structured Specification.

Although the building of the Structured Specification is the most important aspect of Structured Analysis, there are some minor extras:

- estimating heuristics
- methods to facilitate the transition from analysis to design
- aids for acceptance test generation
- walkthrough techniques

1.4.4 What Structured Analysis Is Not

Structured Analysis deals mostly with a subset of analysis. There are many legitimate aspects of analysis to which Structured Analysis does not directly apply. For the record, I have listed items of this type below:

- cost-benefit analysis
- feasibility analysis
- project management
- performance analysis
- conceptual thinking (Structured Analysis might help you communicate better with the user; but if the user is just plain wrong, that might not be of much long-term benefit.)
- equipment selection
- personnel considerations
- politics

My treatment of these subjects is limited to showing how they fit in with the modified methods of Structured Analysis.

المنارة للاستشارات

3 THE TOOLS OF STRUCTURED ANALYSIS

The purpose of this chapter is to give you a look at each one of the tools of Structured Analysis at work. Once you have a good idea of what they are and how they fit together, we can go back and discuss the details.

3.1 A sample situation

The first example I have chosen is a real one, involving the workings of our own company, Yourdon inc. To enhance your understanding of what follows, you ought to be aware of these facts:

1. Yourdon is a medium-sized computer consulting and training company that teaches public and inhouse sessions in major cities in North America and occasionally elsewhere.
2. People register for seminars by mail and by phone. Each registration results in a confirmation letter and invoice being sent back to the registrant.
3. Payments come in by mail. Each payment has to be matched up to its associated invoice to credit accounts receivable.
4. There is a mechanism for people to cancel their registrations if they should have to.
5. Once you have taken one of the company's courses, or even expressed interest in one, your name is added to a data base of people to be pursued relentlessly forever after. This data base contains entries for tens of thousands of people in nearly as many organizations.
6. In addition to the normal sales prompting usage of the data base, it has to support inquiries such as
    - When is the next Structured Design Programming Workshop in the state of California?
    - Who else from my organization has attended the Structured Analysis seminar? How did they rate it?
    - Which instructor is giving the Houston Structured Design and Programming Workshop next month?

In early 1976, Yourdon began a project to install a set of automated management and operational aids on a PDP-11/45, running under the UNIX operating system. Development of the system — which is now operational — first called for a study of sales and accounting functions. The study made use of the tools and techniques of Structured Analysis. The following subsections present some partial and interim products of our analysis.

3.2 A Data Flow Diagram example

An early model of the operations of the company is presented in Fig. 9. It is in the form of a Logical Data Flow Diagram. Refer to that figure now, and we'll walk through one of its paths. The rest should be clear by inference.

Input to the portrayed area comes in the form of Transactions ("Trans" in the figure). These are of five types: Cancellations, Enrollments, Payments, Inquiries, plus those that do not qualify as any of these, and are thus considered Rejects. Although there are no people or locations or departments shown on this figure (it is logical, not physical), I will fill some of these in for you, just as I would for a user to help him relate back to the physical situation that he knows. The receptionist (a physical consideration) handles all incoming transactions, whether they come by phone or by mail. He performs the initial edit, shown as Process 1 in the figure. People who want to take a course in Unmitigated Freelance Speleology, for example, are told to look elsewhere. Incomplete or improperly specified enrollment requests and inquiries, etc., are sent back to the originator with a note. Only clean transactions that fall into the four acceptable categories are passed on.

Enrollments go next to the registrar. His function (Process 2) is to use the information on the enrollment form to update three files: the People File, the Seminar File, and the Payments File. He then fills out an enrollment chit and passes it on to the accounting department. In our figure, the enrollment chit is called "E-Data," and the accounting process that receives it is Process 6.

Information on the chit is now transformed into an invoice. This process is partially automated, by the way — a ledger machine is used — but that information is not shown on a logical Data Flow Diagram.

The invoice passes on to the confirmation process (which happens to be done by the receptionist in this case). This task (Process 7) involves combining the invoice with a customized form letter, to be sent out together as a confirmation. The confirmation goes back to the customer.

3.2.1 Some Data Flow Diagram Conventions

If you have followed the narrative so far, you have already picked up the major Data Flow Diagram conventions:

- *The Data Flow Diagram shows flow of data, not of control.* This is the difference between Data Flow Diagrams and flowcharts. The Data Flow Diagram portrays a situation from the point of view of the data, while a flowchart portrays it from the point of view of those who act upon the data. For this reason, you almost never see a loop in a Data Flow Diagram. A loop is something that the data are unaware of; each datum typically goes through it once, and so from its point of view it is not a loop at all. Loops and decisions are control considerations and do not appear in Data Flow Diagrams.

المنارة للاستشارات

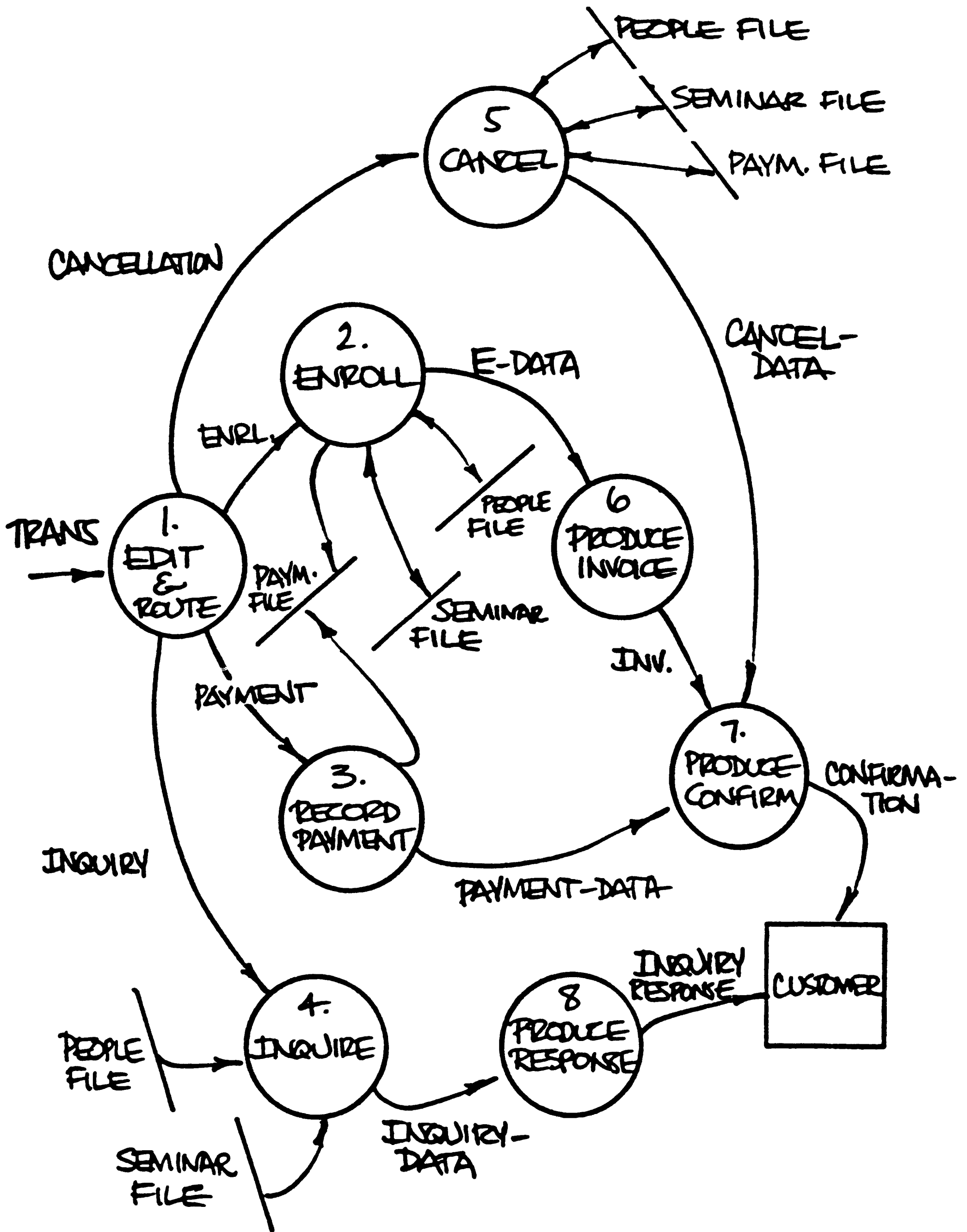


Figure 9

المنارة للاستشارات

- *Four notational symbols are used.* These are:
  - The named vector (called a data flow), which portrays a data path.
  - The bubble (called a process), which portrays transformation of data.
  - The straight line, which portrays a file or data base.
  - The box (called a source or sink), which portrays a net originator or receiver of data — typically a person or an organization outside the domain of our study.

Since no control is shown, you can't tell from looking at a Data Flow Diagram which path will be followed. The Data Flow Diagram shows only the set of possible paths. Similarly, you can't tell what initiates a given process. You cannot assume, for instance, that Process 6 is started by the arrival of an E-Data — in fact, that's not how it works at all. E-Data's accumulate until a certain day of the week arrives, and then invoices all go out in a group. So the data flow E-Data indicates the data path, but not the prompt. The prompting information does not appear on a Data Flow Diagram.

3.2.2 An Important Advantage of the Data Flow Diagram

Suppose you were walking through Fig. 9 with your user and he made the comment: "That's all very fine, but in addition to seminars, this company also sells books. I don't see the books operation anywhere."

"Don't worry, Mr. User," you reply, "the book operation is fully covered here," (now you are thinking furiously where to stick it) "here in Process . . . um . . . Process Number 3. Yes, definitely 3. It's part of recording payments, only you have to look into the details to see that."

Analysts are always good at thinking on their feet, but in this case, the effort is futile. The book operation has quite simply been *left out* of Fig. 9 — it's wrong. No amount of thinking on your feet can cover up this failing. No books flow in or out, no inventory information is available, no reorder data flows are shown. Process 3 simply doesn't have access to the information it needs to carry out books functions. Neither do any of the others.

Your only option at this point is to admit the figure is wrong and fix it. While this might be galling when it happens, in the long run you are way ahead — making a similar change later on to the hard code would cost you considerably more grief.

I have seen this happen so many times: an analyst caught flat-footed with an incorrect Data Flow Diagram, trying to weasel his way out, but eventually being forced to admit that it is wrong and having to fix it. I conclude that it is a natural characteristic of the tool:

> *When a Data Flow Diagram is wrong, it is glaringly, demonstrably, indefensibly wrong.*

المنارة للاستشارات

This seems to me to be an enormous advantage of using Data Flow Diagrams.

3.2.3 What Have We Accomplished With a Data Flow Diagram?

The Data Flow Diagram is documentation of a situation from the point of view of the data. This turns out to be a more useful viewpoint than that of any of the people or systems that process the data, because the data itself sees the big picture. So the first thing we have accomplished with the Data Flow Diagram is to come up with a meaningful portrayal of a system or a part of a system.

The Data Flow Diagram can also be used as a model of a real situation. You can try things out on it conveniently and get a good idea of how the real system will react when it is finally built.

Both the conceptual documentation and the modeling are valuable results of our Data Flow Diagramming effort. But something else, perhaps more important, has come about as a virtually free by-product of the effort: The Data Flow Diagram gives us a highly useful *partitioning* of a system. Fig. 9 shows an unhandily large operation conveniently broken down into eight pieces. It also shows all the interfaces among those eight pieces. (If any interface is left out, the diagram is simply wrong and has to be fixed.)

Notice that the use of a Data Flow Diagram causes us to go about our partitioning in a rather oblique way. If what we wanted to do was break things down, why didn't we just do that? Why didn't we concentrate on functions and subfunctions and just accomplish a brute-force partitioning? The reason for this is that a brute-force partitioning is too difficult. It is too difficult to say with any assurance that some task or group of tasks constitutes a "function." In fact, I'll bet you can't even define the word function except in a purely mathematical sense. Your dictionary won't do much better — it will give a long-winded definition that boils down to saying a function is a bunch of stuff to be done. The concept of function is just too imprecise for our purposes.

The oblique approach of partitioning by Data Flow Diagram gives us a "functional" partitioning, where this very special-purpose definition of the word functional applies:

> A partitioning may be considered *functional* when the interfaces among the pieces are minimized.

This kind of partitioning is ideal for our purposes.

3.3 A Data Dictionary example

Refer back to Fig. 9 for a moment. What is the interface between Process 3 and Process 7? As long as all that specifies the interface is the weak name "Payment-Data," we don't have a specification at all. "Payment-Data" could mean anything. We must state precisely what me mean by the data flow bearing that name in order for our Structured Specification to be anything more than a hazy sketch of the system. It is in the Data Dictionary that we state precisely what each of our data flows is made up of.

المنارة للاستشارات

An entry from the sample project Data Dictionary might look like this:

Payment-Data = Customer-Name +
Customer-Address +
Invoice-Number +
Amount-of-Payment

In other words, the data flow called "Payment-Data" consists precisely of the items Customer-Name, Customer-Address, Invoice-Number, and Amount-of-Payment, concatenated together. They must appear in that order, and they must all be present. No other kind of data flow could qualify as a Payment-Data, even though the name might be applicable.

You may have to make several queries to the Data Dictionary in order to understand a term completely enough for your needs. (This also happens with conventional dictionaries — you might look up the term perspicacious, and find that it means sagacious; then you have to look up sagacious.) In the case of the example above, you may have to look further in the Data Dictionary to see exactly what an Invoice-Number is:

Invoice-Number = State-Code +
Customer-Account-Number +
Salesman-ID +
Sequential-Invoice-Count

Just as the Data Flow Diagram effects a partitioning of the area of our study, the Data Dictionary effects a top-down partitioning of our data. At the highest levels, data flows are defined as being made up of subordinate elements. Then the subordinate elements (also data flows) are themselves defined in terms of still more detailed subordinates.

Before our Structured Specification is complete, there will have to be a Data Dictionary entry for every single data flow on our Data Flow Diagram, and for all the subordinates used to define them. In the same fashion, we can use Data Dictionary entries to define our files.

3.4 A Structured English example

Partitioning is a great aid to specification, but you can't specify by partitioning alone. At some point you have to stop breaking things down into finer and finer pieces, and actually document the makeup of the pieces. In the terms of our Structured Specification, we have to state what it takes to do each of the data transformations indicated by a bubble on our Data Flow Diagram.

There are many ways we could go about this. Narrative text is certainly the most familiar of these. To the extent that we have partitioned sufficiently before beginning to specify, we may be spared the major difficulties of narrative description. However, we can do even better.

المنارة للاستشارات

A tool that is becoming more and more common for process description is Structured English. Presented below is a Structured English example of a user's invoice handling policy from the sample analysis. It appears without clarification; if clarification is needed, it has failed in its intended purpose.

==================================

POLICY FOR INVOICE PROCESSING

```
If the amount of the invoice exceeds $500,
    If the account has any invoice more than 60 days overdue,
        hold the confirmation pending resolution of the debt.
    Else (account is in good standing),
        issue confirmation and invoice.
Else (invoice $500 or less),
    If the account has any invoice more than 60 days overdue,
        issue confirmation, invoice and write message on the
        credit action report.
    Else (account is in good standing),
        issue confirmation and invoice.
```

==================================

3.5 A Decision Table example

The same policy might be described as well by a Decision Table:

	RULES			
CONDITIONS	**1**	**2**	**3**	**4**
1. Invoice > $500	Y	N	Y	N
2. Account overdue by 60+ days	Y	Y	N	N
ACTIONS				
1. Issue Confirmation	N	Y	Y	Y
2. Issue Invoice	N	Y	Y	Y
3. Msg to C.A.R.	N	Y	N	N

3.6 A Decision Tree example

As a third alternative, you might describe the same policy with a Decision Tree. I have included the equivalent Decision Tree as Fig. 10.

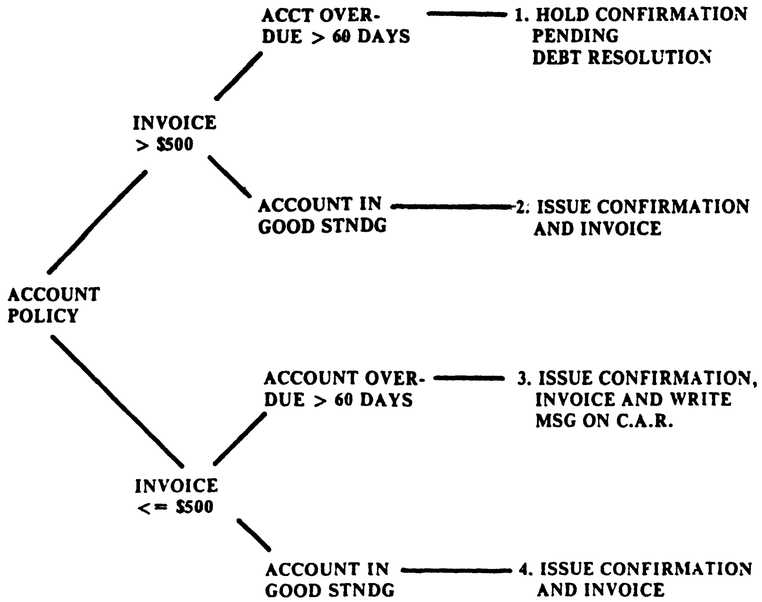


Figure 10

المنارة للاستشارات

Edsger Dijkstra

Solution of a Problem in Concurrent Programming Control

Communications of the ACM, Vol. 8 (9), 1965
pp. 569

Go To Statement Considered Harmful

Communications of the ACM, Vol. 11 (2), 1968
pp. 147–148

المنارة للاستشارات

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more

المنارة للاستشارات

computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** $b, c[1:N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer $(1 \leq i \leq N)$ is:

```
"integer j;
Li0:  b[i] := false;
Li1:  if k ≠ i then
Li2:  begin c[i] := true;
Li3:  if b[k] then k := i;
      go to Li1
      end
        else
Li4:  begin c[i] := false;
        for j := 1 step 1 until N do
          if j ≠ i and not c[j] then go to Li1
      end;
      critical section;
      c[i] := true;  b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0"
```

The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement *Li*4 without jumping back to *Li*1, i.e., finding all other *c*'s **true** after having set its own *c* to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to *Li*1) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the *k*th computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in *Li*3 the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to

المنارة للاستشارات

k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement $Li4$) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.

المنارة للاستشارات

المنارة للاستشارات

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the

المنارة للاستشارات

program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy $(B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En)$, the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommenda-

المنارة للاستشارات

tion to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

المنارة للاستشارات

Michael Fagan

Design and Code Inspections to Reduce Errors in Program Development

IBM Systems Journal, Vol. 15 (3), 1976
pp. 182–211

المنارة للاستشارات

Substantial net improvements in programming quality and productivity have been obtained through the use of formal inspections of design and of code. Improvements are made possible by a systematic and efficient design and code verification process, with well-defined roles for inspection participants. The manner in which inspection data is categorized and made suitable for process analysis is an important factor in attaining the improvements. It is shown that by using inspection results, a mechanism for initial error reduction followed by ever-improving error rates can be achieved.

Design and code inspections to reduce errors in program development

by M. E. Fagan

Successful management of any process requires planning, measurement, and control. In programming development, these requirements translate into defining the programming process in terms of a series of operations, each operation having its own exit criteria. Next there must be some means of measuring completeness of the product at any point of its development by inspections or testing. And finally, the measured data must be used for controlling the process. This approach is not only conceptually interesting, but has been applied successfully in several programming projects embracing systems and applications programming, both large and small. It has not been found to "get in the way" of programming, but has instead enabled higher predictability than other means, and the use of inspections has improved productivity and product quality. The purpose of this paper is to explain the planning, measurement, and control functions as they are affected by inspections in programming terms.

An ingredient that gives maximum play to the planning, measurement, and control elements is consistent and vigorous *discipline*. Variable rules and conventions are the usual indicators of a lack of discipline. An iron-clad discipline on all rules, which can stifle programming work, is not required but instead there should be a clear understanding of the flexibility (or nonflexibility) of each of the rules applied to various aspects of the pro-

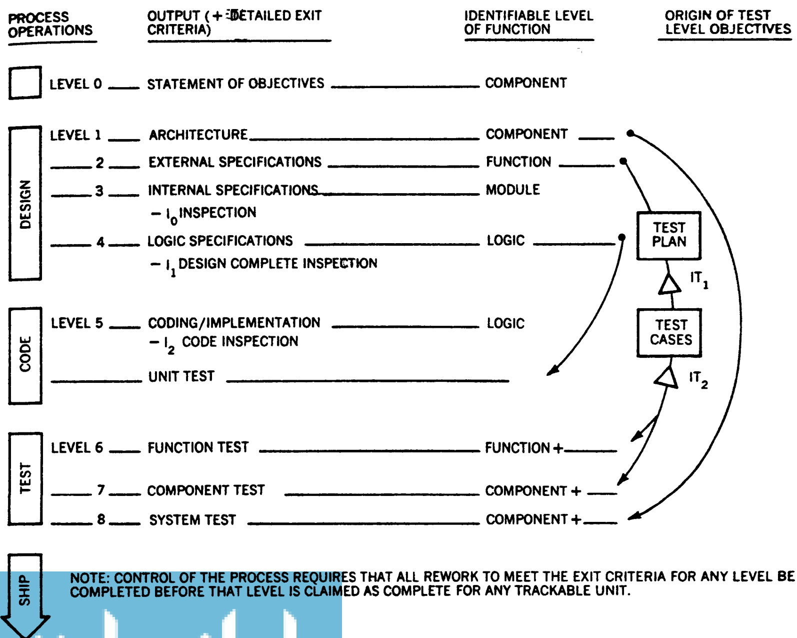
المنارة للاستشارات

ject. An example of flexibility may be waiving the rule that all main paths will be tested for the case where repeated testing of a given path will logically do no more than add expense. An example of necessary inflexibility would be that *all* code must be inspected. A clear statement of the project rules and changes to these rules along with faithful adherence to the rules go a long way toward practicing the required project discipline.

A prerequisite of process management is a clearly defined series of operations in the process (Figure 1). The miniprocess within each operation must also be clearly described for closer management. A clear statement of the criteria that must be satisfied to exit each operation is mandatory. This statement and accurate data collection, with the data clearly tied to trackable units of known size and collected from specific points in the process, are some essential constituents of the information required for process management.

In order to move the form of process management from qualitative to more quantitative, process terms must be more specific, data collected must be appropriate, and the limits of accuracy of the data must be known. The effect is to provide more precise

Figure 1 Programming process



المنارة للاستشارات

information in the correct process context for decision making by the process manager.

In this paper, we first describe the programming process and places at which inspections are important. Then we discuss factors that affect productivity and the operations involved with inspections. Finally, we compare inspections and walk-throughs on process control.

a manageable process

A process may be described as a set of operations occurring in a definite sequence that operates on a given input and converts it to some desired output. A general statement of this kind is sufficient to convey the notion of the process. In a practical application, however, it is necessary to describe the input, output, internal processing, and processing times of a process in very specific terms if the process is to be executed and practical output is to be obtained.

In the programming development process, explicit requirement statements are necessary as input. The series of processing operations that act on this input must be placed in the correct sequence with one another, the output of each operation satisfying the input needs of the next operation. The output of the final operation is, of course, the explicitly required output in the form of a verified program. Thus, the objective of each processing operation is to receive a defined input and to produce a definite output that satisfies a specific set of exit criteria. (It goes without saying that each operation can be considered as a miniprocess itself.) A well-formed process can be thought of as a continuum of processing during which sequential sets of exit criteria are satisfied, the last set in the entire series requiring a well-defined end product. Such a process is not amorphous. It can be measured and controlled.

exit criteria

Unambiguous, explicit, and universally accepted exit criteria would be perfect as process control checkpoints. It is frequently argued that universally agreed upon checkpoints are impossible in programming because all projects are different, etc. However, *all* projects do reach the point at which there is a project checkpoint. As it stands, any trackable unit of code achieving a clean compilation can be said to have satisfied a universal exit criterion or checkpoint in the process. Other checkpoints can also be selected, albeit on more arguable premises, but once the prem-

المنارة للاستشارات

ises are agreed upon, the checkpoints become visible in most, if not all, projects. For example, there is a point at which the design of a program is considered complete. This point may be described as the level of detail to which a unit of design is reduced so that one design statement will materialize in an estimated three to 10 source code instructions (or, if desired, five to 20, for that matter). Whichever particular ratio is selected across a project, it provides a checkpoint for the process control of that project. In this way, suitable checkpoints may be selected throughout the development process and used in process management. (For more specific exit criteria see Reference 1.)

The cost of reworking errors in programs becomes higher the later they are reworked in the process, so every attempt should be made to find and fix errors as early in the process as possible. This cost has led to the use of the inspections described later and to the description of exit criteria which include assuring that all errors known at the end of the inspection of the new "clean-compilation" code, for example, have been correctly fixed. So, rework of all known errors up to a particular point must be complete before the associated checkpoint can be claimed to be met for any piece of code.

Where inspections are not used and errors are found during development or testing, the cost of rework as a fraction of overall development cost can be suprisingly high. For this reason, errors should be found and fixed as close to their place of origin as possible.

Production studies have validated the expected quality and productivity improvements and have provided estimates of standard productivity rates, percentage improvements due to inspections, and percentage improvements in error rates which are applicable in the context of large-scale operating system program production. (The data related to operating system development contained herein reflect results achieved by IBM in applying the subject processes and methods to representative samples. Since the results depend on many factors, they cannot be considered representative of every situation. They are furnished merely for the purpose of illustrating what has been achieved in sample testing.)

The purpose of the test plan inspection IT_1, shown in Figure 1, is to find voids in the functional variation coverage and other discrepancies in the test plan. IT_2, test case inspection of the test cases, which are based on the test plan, finds errors in the

المنارة للاستشارات

test cases. The total effects of IT_1 and IT_2 are to increase the integrity of testing and, hence, the quality of the completed product. And, because there are less errors in the test cases to be debugged during the testing phase, the overall project schedule is also improved.

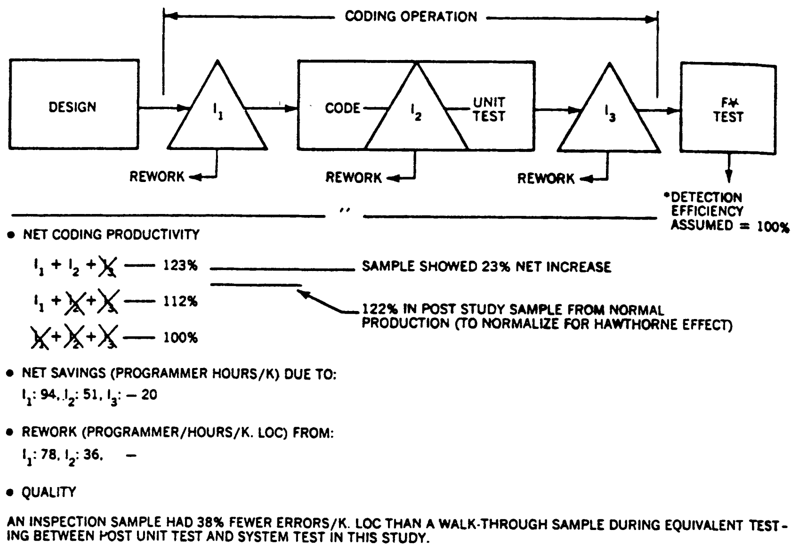
A process of the kind depicted in Figure 1 installs all the intrinsic programming properties in the product as required in the statement of objectives (Level 0) by the time the coding operation (Level 5) has been completed—except for packaging and publications requirements. With these exceptions, all later work is of a verification nature. This verification of the product provides no contribution to the product during the essential development (Levels 1 to 5); it only adds error detection and elimination (frequently at one half of the development cost). I_0, I_1, and I_2 inspections were developed to measure and influence intrinsic quality (error content) in the early levels, where error rework can be most economically accomplished. Naturally, the beneficial effect on quality is also felt in later operations of the development process and at the end user's site.

An improvement in productivity is the most immediate effect of purging errors from the product by the I_0, I_1, and I_2 inspections. This purging allows rework of these errors very near their origin, early in the process. Rework done at these levels is 10 to 100 times less expensive than if it is done in the last half of the process. Since rework detracts from productive effort, it reduces productivity in proportion to the time taken to accomplish the rework. It follows, then, that finding errors by inspection and reworking them earlier in the process reduces the overall rework time and increases productivity even within the early operations and even more over the total process. Since less errors ship with the product, the time taken for the user to install programs is less, and his productivity is also increased.

The quality of documentation that describes the program is of as much importance as the program itself for poor quality can mislead the user, causing him to make errors quite as important as errors in the program. For this reason, the quality of program documentation is verified by publications inspections (PI_0, PI_1, and PI_2). Through a reduction of user-encountered errors, these inspections also have the effect of improving user productivity by reducing his rework time.

المنارة للاستشارات

Figure 2 A study of coding productivity



A study of coding productivity

A piece of the design of a large operating system component (all done in structured programming) was selected as a study sample (Figure 2). The sample was judged to be of moderate complexity. When the piece of design had been reduced to a level of detail sufficient to meet the Design Level 4 exit criteria[2] (a level of detail of design at which one design statement would ultimately appear as three to 10 code instructions), it was submitted to a design-complete inspection (100 percent), I_1. On conclusion of I_1, all error rework resulting from the inspection was completed, and the design was submitted for coding in PL/S. The coding was then done, and when the code was brought to the level of the first clean compilation,[2] it was subjected to a code inspection (100 percent), I_2. The resultant rework was completed and the code was subjected to unit test. After unit test, a unit test inspection, I_3, was done to see that the unit test plan had been fully executed. Some rework was required and the necessary changes were made. This step completed the coding operation. The study sample was then passed on to later process operations consisting of building and testing.

inspection sample

The inspection sample was considered of sufficient size and nature to be representative for study purposes. Three programmers

المنارة للاستشارات

designed it, and it was coded by 13 programmers. The inspection sample was in modular form, was structured, and was judged to be of moderate complexity on average.

coding operation productivity

Because errors were identified and corrected in groups at I_1 and I_2, rather than found one-by-one during subsequent work and handled at the higher cost incumbent in later rework, the overall amount of error rework was minimized, even within the coding operation. Expressed differently, considering the inclusion of *all* I_1 time, I_2 time, and resulting error rework time (with the usual coding and unit test time in the total time to complete the operation), a *net* saving resulted when this figure was compared to the no-inspection case. This net saving translated into a 23 percent increase in the productivity of the coding operation alone. Productivity in later levels was also increased because there was less error rework in these levels due to the effect of inspections, but the increase was not measured directly.

An important aspect to consider in any production experiment involving human beings is the Hawthorne Effect.[3] If this effect is not adequately handled, it is never clear whether the effect observed is due to the human bias of the Hawthorne Effect or due to the newly implemented change in process. In this case a *control sample* was selected at random from many pieces of work *after the I_1 and I_2 inspections were accepted as commonplace.* (Previous experience without I_1 and I_2 approximated the net coding productivity rate of 100 percent datum in Figure 2.) The difference in coding productivity between the experimental sample (with I_1 and I_2 for the first time) and the control sample was 0.9 percent. This difference is not considered significant. Therefore, the measured increase in coding productivity of 23 percent is considered to validly accrue from the only change in the process: addition of I_1 and I_2 inspections.

control sample

The control sample was also considered to be of representative size and was from the same operating system component as the study sample. It was designed by four programmers and was coded by seven programmers. And it was considered to be of moderate complexity on average.

net savings

Within the coding operation only, the net savings (including inspection and rework time) in programmer hours per 1000 Non-Commentary Source Statements (K.NCSS)[4] were I_1: 94, I_2: 51, and I_3: −20. As a consequence, I_3 is no longer in effect.

المنارة للاستشارات

If personal fatigue and downtime of 15 percent are allowed in addition to the 145 programmer hours per K.NCSS, the saving approaches one programmer month per K.NCSS (assuming that our sample was truly representative of the rest of the work in the operating system component considered).

error rework

The error rework in programmer hours per K.NCSS found in this study due to I_1 was 78, and 36 for I_2 (24 hours for design errors and 12 for code errors). Time for error rework must be specifically scheduled. (For scheduling purposes it is best to develop rework hours per K.NCSS from history depending upon the particular project types and environments, but figures of 20 hours for I_1, and 16 hours for I_2 (*after the learning curve*) may be suitable to start with.)

quality

The only comparative measure of quality obtained was a comparison of the inspection study sample with a fully comparable piece of the operating system component that was produced similarly, except that walk-throughs were used in place of the I_1 and I_2 inspections. (Walk-throughs[5] were the practice before implementation of I_1 and I_2 inspections.) The process span in which the quality comparison was made was seven months of testing beyond unit test after which it was judged that both samples had been equally exercised. The results showed the inspection sample to contain 38 percent less errors than the walk-through sample.

Note that up to inspection I_2, no machine time has been used for debugging, and so machine time savings were not mentioned. Although substantial machine time is saved overall since there are less errors to test for in inspected code in later stages of the process, no actual measures were obtained.

Table 1 Error detection efficiency

Process Operations	*Errors Found per K.NCSS*	*Percent of Total Errors Found*
Design		
I_1 inspection }		
Coding }	38*	82
I_2 inspection }		
Unit test }		
Preparation for acceptance test }	8	18
Acceptance test	0	
Actual usage (6 mo.)	0	
Total	46	100

*51% were logic errors, most of which were missing rather than due to incorrect design.

المنارة للاستشارات

inspections in applications development

In the development of applications, inspections also make a significant impact. For example, an application program of eight modules was written in COBOL by Aetna Corporate Data Processing department, Aetna Life and Casualty, Hartford, Connecticut, in June 1975.[6] Two programmers developed the program. The number of inspection participants ranged between three and five. The only change introduced in the development process was the I_1 and I_2 inspections. The program size was 4,439 Non-Commentary Source Statements.

An automated estimating program, which is used to produce the normal program development time estimates for all the Corporate Data Processing department's projects, predicted that designing, coding, and unit testing this project would require 62 programmer days. In fact, the time actually taken was 46.5 programmer days including inspection meeting time. The resulting saving in programmer resources was 25 percent.

The inspections were obviously very thorough when judged by the inspection error detection efficiency of 82 percent and the later results during testing and usage as shown in Table 1.

The results achieved in Non-Commentary Source Statements per Elapsed Hour are shown in Table 2. These inspection rates are four to six times faster than for systems programming. If these rates are generally applicable, they would have the effect of making the inspection of applications programs much less expensive.

Table 2 Inspection rates in NCSS per hour

Operations	I_1	I_2
Preparation	898	709
Inspection	652	539

Inspections

Inspections are a *formal*, *efficient*, and *economical* method of finding errors in design and code. All instructions are addressed at least once in the conduct of inspections. Key aspects of inspections are exposed in the following text through describing the I_1 and I_2 inspection conduct and process. I_0, IT_1, IT_2, PI_0, PI_1, and PI_2 inspections retain the same essential properties as

المنارة للاستشارات

Table 3. Inspection process and rate of progress

Process operations	Rate of progress* (loc/hr) Design I_1	Code I_2	Objectives of the operation
1. Overview	500	not necessary	Communication education
2. Preparation	100	125	Education
3. Inspection	130	150	*Find errors*
4. Rework	20 hrs/K.NCSS	16 hrs/K.NCSS	Rework and resolve errors found by inspection
5. Follow-up	–	–	See that all errors, problems, and concerns have been resolved

*These notes apply to systems programming and are conservative. Comparable rates for applications programming are much higher. Initial schedules may be started with these numbers and as project history that is keyed to unique environments evolves, the historical data may be used for future scheduling algorithms.

the I_1 and I_2 inspections but differ in materials inspected, number of participants, and some other minor points.

the people involved

The inspection team is best served when its members play their particular roles, assuming the particular vantage point of those roles. These roles are described below:

1. *Moderator*—The *key person* in a successful inspection. He must be a competent programmer but need *not* be a technical expert on the program being inspected. To preserve objectivity and to increase the integrity of the inspection, it is usually advantageous to use a moderator from an unrelated project. The moderator must manage the inspection team and offer leadership. Hence, he must use personal sensitivity, tact, and drive in balanced measure. His use of the strengths of team members should produce a synergistic effect larger than their number; in other words, *he is the coach*. The duties of moderator also include scheduling suitable meeting places, reporting inspection results within one day, and follow-up on rework. *For best results the moderator should be specially trained.* (This training is brief but very advantageous.)
2. *Designer*—The programmer responsible for producing the program design.
3. *Coder/Implementor*—The programmer responsible for translating the design into code.
4. *Tester*—The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

المنارة للاستشارات

If the coder of a piece of code also designed it, he will function in the designer role for the inspection process; a coder from some related or similar program will perform the role of the coder. If the same person designs, codes, and tests the product code, the coder role should be filled as described above, and another coder—preferably with testing experience—should fill the role of tester.

Four people constitute a good-sized inspection team, although circumstances may dictate otherwise. The team size should not be artificially increased over four, but if the subject code is involved in a number of interfaces, the programmers of code related to these interfaces may profitably be involved in inspection. Table 3 indicates the inspection process and rate of progress.

scheduling inspections and rework

The total time to complete the inspection process from overview through follow-up for I_1 or I_2 inspections with four people involved takes about 90 to 100 people-hours for systems programming. Again, these figures may be considered conservative but they will serve as a starting point. Comparable figures for applications programming tend to be much lower, implying lower cost per K.NCSS.

Because the error detection efficiency of most inspection teams tends to dwindle after two hours of inspection but then picks up after a period of different activity, it is advisable to schedule inspection sessions of no more than two hours at a time. Two two-hour sessions per day are acceptable.

The time to do inspections and resulting rework must be scheduled and managed with the same attention as other important project activities. (After all, as is noted later, for one case at least, it is possible to find approximately two thirds of the errors reported during an inspection.) If this is not done, the immediate work pressure has a tendency to push the inspections and/or rework into the background, postponing them or avoiding them altogether. The result of this short-term respite will obviously have a much more dramatic long-term negative effect since the finding and fixing of errors is delayed until later in the process (and after turnover to the user). Usually, the result of postponing early error detection is a lengthening of the overall schedule and increased product cost.

Scheduling inspection time for modified code may be based on the algorithms in Table 3 *and on judgment*.

المنارة للاستشارات

I_1 inspection process

Keeping the objective of each operation in the forefront of team activity is of paramount importance. Here is presented an outline of the I_1 inspection process operations.

1. *Overview* (whole team) — The designer first describes the overall area being addressed and then the specific area he has designed in detail — logic, paths, dependencies, etc. Documentation of design is distributed to all inspection participants on conclusion of the overview. (For an I_2 inspection, no overview is necessary, but the participants should remain the same. Preparation, inspection, and follow-up proceed as for I_1 but, of course, using code listings *and* design specifications as inspection materials. Also, at I_2 the moderator should flag for special scrutiny those areas that were reworked since I_1 errors were found *and other design changes* made.)

2. *Preparation* (individual) — Participants, using the design documentation, literally do their homework to try to understand the design, its intent and logic. (Sometimes flagrant errors are found during this operation, but in general, the number of errors found is not nearly as high as in the inspection operation.) To increase their error detection in the inspection, the inspection team should first study the ranked distributions of error types found by recent inspections. This study will prompt them to concentrate on the most fruitful areas. (See examples in Figures 3 and 4.) Checklists of clues on finding these errors should also be studied. (See partial examples of these lists in Figures 5 and 6 and complete examples for I_0 in Reference 1 and for I_1 and I_2 in Reference 7.)

3. *Inspection* (whole team) — A "reader" chosen by the moderator (usually the coder) describes how he will implement the design. He is expected to paraphrase the design as expressed by the designer. Every piece of logic is covered at least once, and every branch is taken at least once. All higher-level documentation, high-level design specifications, logic specifications, etc., and macro and control block listings at I_2 must be available and present during the inspection.

   Now that the design is understood, *the objective is to find errors*. (Note that an error is defined as any condition that causes malfunction or that precludes the attainment of expected or previously specified results. Thus, deviations from specifications are clearly termed errors.) The finding of errors is actually done during the implementor/coder's dis-

المنارة للاستشارات

Figure 3 Summary of design inspections by error type

VP Individual Name	Inspection file Missing	Wrong	Extra	Errors	Error %
CD CB Definition	16	2		18	3.5 } 10.4
CU CB Usage	18	17	1	36	6.9 }
FS FPFS	1			1	.2
IC Interconnect Calls	18	9		27	5.2
IR Interconnect Reqts	4	5	2	11	2.1
LO Logic	126	57	24	207	39.8 ←
L3 Higher Lvl Docu	1		1	2	.4
MA Mod Attributes	1			1	.2
MD More Detail	24	6	2	32	6.2
MN Maintainability	8	5	3	16	3.1
OT Other	15	10	10	35	6.7
PD Pass Data Areas		1		1	.2
PE Performance	1	2	3	6	1.2
PR Prologue/Prose	44	38	7	89	17.1 ←
RM Return Code/Msg	5	7	2	14	2.7
RU Register Usage	1	2		3	.6
ST Standards					
TB Test & Branch	12	7	2	21	4.0
	295	168	57	520	100.0
	57%	32%	11%		

Figure 4 Summary of code inspections by error type

VP Individual Name	Inspection file Missing	Wrong	Extra	Errors	Error %
CC Code Comments	5	17	1	23	6.6
CU CB Usage	3	21	1	25	7.2
DE Design Error	31	32	14	77	22.1 ←
F1		8		8	2.3
IR Interconnect Calls	7	9	3	19	5.5
LO Logic	33	49	10	92	26.4 ←
MN Maintainability	5	7	2	14	4.0
OT Other					
PE Performance	3	2	5	10	2.9
PR Prologue/Prose	25	24	3	52	14.9 ←
PU PL/S or BAL Use	4	9	1	14	4.0
RU Register Usage	4	2		6	1.7
SU Storage Usage	1			1	.3
TB Test & Branch	2	5		7	2.0
	123	185	40	348	100.0

course. Questions raised are pursued only to the point at which an error is recognized. It is noted by the moderator; its type is classified; severity (major or minor) is identified, and the inspection is continued. Often the solution of a problem is obvious. If so, it is noted, but no specific solution hunting is

المنارة للاستشارات

Figure 5 Examples of what to examine when looking for errors at I_1

I_1 Logic

Missing

1. Are All Constants Defined?
2. Are All Unique Values Explicitly Tested on Input Parameters?
3. Are Values Stored after They Are Calculated?
4. Are All Defaults Checked Explicitly Tested on Input Parameters?
5. If Character Strings Are Created Are They Complete, Are All Delimiters Shown?
6. If a Keyword Has Many Unique Values, Are They All Checked?
7. If a Queue Is Being Manipulated, Can the Execution Be Interrupted; If So, Is Queue Protected by a Locking Structure; Can Queue Be Destroyed Over an Interrupt?
8. Are Registers Being Restored on Exits?
9. In Queuing/Dequeuing Should Any Value Be Decremented/Incremented?
10. Are All Keywords Tested in Macro?
11. Are All Keyword Related Parameters Tested in Service Routine?
12. Are Queues Being Held in Isolation So That Subsequent Interrupting Requestors Are Receiving Spurious Returns Regarding the Held Queue?
13. Should any Registers Be Saved on Entry?
14. Are All Increment Counts Properly Initialized (0 or 1)?

Wrong

1. Are Absolutes Shown Where There Should Be Symbolics?
2. On Comparison of Two Bytes, Should All Bits Be Compared?
3. On Built Data Strings, Should They Be Character or Hex?
4. Are Internal Variables Unique or Confusing If Concatenated?

Extra

1. Are All Blocks Shown in Design Necessary or Are They Extraneous?

to take place during inspection. (The inspection is *not* intended to redesign, evaluate alternate design solutions, or to find solutions to errors; it is intended just to find errors!) A team is most effective if it operates with only one objective at a time.

Within one day of conclusion of the inspection, the moderator should produce a written report of the inspection and its findings to ensure that all issues raised in the inspection will be addressed in the rework and follow-up operations. Examples of these reports are given as Figures 7A, 7B, and 7C.

4. *Rework*—All errors or problems noted in the inspection report are resolved by the designer or coder/implementor.

5. *Follow-Up*—It is imperative that every issue, concern, and error be entirely resolved at this level, or errors that result can be 10 to 100 times more expensive to fix if found later in

المنارة للاستشارات

Figure 6 Examples of what to examine when looking for errors at I_2

INSPECTION SPECIFICATION

I_2 *Test Branch*

Is Correct Condition Tested (If X = ON vs. IF X = OFF)?
Is (Are) Correct Variable(s) Used for Test
(If X = ON vs. If Y = ON)?
Are Null THENs/ELSEs Included as Appropriate?
Is Each Branch Target Correct?
Is the Most Frequently Exercised Test Leg the THEN Clause?

I_2 *Interconnection (or Linkage) Calls*

For Each Interconnection Call to Either a Macro, SVC or Another Module:
Are All Required Parameters Passed Set Correctly?
If Register Parameters Are Used, Is the Correct Register Number Specified?
If Interconnection Is a Macro,
Does the Inline Expansion Contain All Required Code?
No Register or Storage Conflicts between Macro and Calling Module?
If the Interconnection Returns, Do All Returned Parameters Get Processed Correctly?

the process (programmer time only, machine time not included). It is the responsibility of the moderator to see that all issues, problems, and concerns discovered in the inspection operation have been resolved by the designer in the case of I_1, or the coder/implementor for I_2 inspections. If more than five percent of the material has been reworked, the team should reconvene and carry out a 100 percent reinspection. Where less than five percent of the material has been reworked, the moderator at his discretion may verify the quality of the rework himself or reconvene the team to reinspect either the complete work or just the rework.

commencing inspections

In Operation 3 above, it is one thing to direct people to find errors in design or code. It is quite another problem for them to find errors. Numerous experiences have shown that people have to be taught or prompted to find errors effectively. Therefore, it is prudent to condition them to seek the high-occurrence, high-cost error types (see example in Figures 3 and 4), and then describe the clues that usually betray the presence of each error type (see examples in Figures 5 and 6).

One approach to getting started may be to make a preliminary inspection of a design or code that is felt to be representative of the program to be inspected. Obtain a suitable quantity of errors, and analyze them by type and origin, cause, and salient indicative clues. With this information, an inspection specification may be constructed. This specification can be amended and improved in

المنارة للاستشارات

Figure 7A Error list

1. PR/M/MIN	Line 3:	the statement of the prologue in the REMARKS section needs expansion.
2. DA/W/MAJ	Line 123:	ERR–RECORD–TYPE is out of sequence.
3. PU/W/MAJ	Line 147:	the wrong bytes of an 8-byte field (current–data) are moved into the 2-byte field (this year).
4. LO/W/MAJ	Line 169:	while counting the number of leading spaces in NAME, the wrong variable (I) is used to calculate "J".
5. LO/W/MAJ	Line 172:	NAME–CHECK is PERFORMED one time too few.
6. PU/E/MIN	Line 175:	In NAME–CHECK, the check for SPACE is redundant.
7. DE/W/MIN	Line 175:	the design should allow for the occurrence of a period in a last name.

Figure 7B Example of module detail report

DATE________________

CODE INSPECTION REPORT

MODULE DETAIL

MOD/MAC: CHECKER SUBCOMPONENT/APPLICATION________________

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LO: LOGIC		9			1	
TB: TEST AND BRANCH						
EL: EXTERNAL LINKAGES						
RU: REGISTER USAGE						
SU: STORAGE USAGE						
DA: DATA AREA USAGE		2				
PU: PROGRAM LANGUAGE		2				1
PE: PERFORMANCE						
MN: MAINTAINABILITY					1	
DE: DESIGN ERROR					1	
PR: PROLOGUE				1		
CC: CODE COMMENTS						
OT: OTHER						
TOTAL:	13			5		

REINSPECTION REQUIRED? Y

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION: A BUG. M = MISSING. W = WRONG. E = EXTRA.
NOTE FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER. 3(2). WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE

light of new experience and serve as an on-going directive to focus the attention and conduct of inspection teams. The objective of an inspection specification is to help maximize and make more consistent the error detection efficiency of inspections where

المنارة للاستشارات

Figure 7C Example of code inspection summary report

CODE INSPECTION REPORT
SUMMARY

Date 11/20/-

To: Design Manager KRAUSS Development Manager GIOTTI
Subject: Inspection Report for CHECKER Inspection date 11 19 -
System/Application ______ Release ______ Build ______
Component ______ Subcomponents(s) ______

					ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				
					Pre-insp			Est Post			Rework							
Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	A	M	D	A	M	D	A	M	D	Prep	Insp Meetg	Re-work	Follow-up	Sub-component
	N		McGINLEY	HALE	348			400			50			9.0	8.8	8.0	1.5	
				Totals														

Reinspection required? YES Length of inspection (clock hours and tenths) 2.2
Reinspection by (date) 11/25/- Additional modules/macros? NO
DCR #'s written C-2
Problem summary: Major 13 Minor 5 Total 18
Errors in changed code: Major ____ Minor ____ Errors in base code: Major ____ Minor ____

LARSON		McGINLEY		HALE	
Initial Desr	Detailed Dr	Programmer	Team Leader	Other	Moderator's Signature

Error detection efficiency

$$= \frac{\text{Errors found by an inspection}}{\text{Total errors in the product before inspection}} \times 100$$

reporting inspection results

The reporting forms and form completion instructions shown in the Appendix may be used for I_1 and I_2 inspections. Although these forms were constructed for use in systems programming development, they may be used for applications programming development with minor modification to suit particular environments.

The moderator will make hand-written notes recording errors found during inspection meetings. He will categorize the errors and then transcribe counts of the errors, by type, to the module detail form. By maintaining cumulative totals of the counts by error type, and dividing by the number of projected executable source lines of code inspected to date, he will be able to establish installation averages within a short time.

Figures 7A, 7B, and 7C are an example of a set of code inspection reports. Figure 7A is a partial list of errors found in code inspection. Notice that errors are described in detail and are classified by error type, whether due to something being missing, wrong, or extra as the cause, and according to major or minor severity. Figure 7B is a module level summary of the errors contained in the entire error list represented by Figure 7A. The code inspection summary report in Figure 7C is a summary of

المنارة للاستشارات

inspection results obtained on all modules inspected in a particular inspection session or in a subcomponent or application.

inspections and languages

Inspections have been successfully applied to designs that are specified in English prose, flowcharts, HIPO, (Hierarchy plus Input-Process-Output) and PIDGEON (an English prose-like meta language).

The first code inspections were conducted on PL/S and Assembler. Now, prompting checklists for inspections of Assembler, COBOL, FORTRAN, and PL/1 code are available.[7]

personnel considerations

One of the most significant benefits of inspections is the detailed feedback of results on a relatively real-time basis. The programmer finds out what error types he is most prone to make and their quantity and how to find them. This feedback takes place within a few days of writing the program. Because he gets early indications from the first few units of his work inspected, he is able to show improvement, and usually does, on later work even during the same project. In this way, feedback of results from inspections must be counted for the programmer's use and benefit: *they should not under any circumstances be used for programmer performance appraisal.*

Skeptics may argue that once inspection results are obtained, they will or even must count in performance appraisals, or at least cause strong bias in the appraisal process. The author can offer in response that inspections have been conducted over the past three years involving diverse projects and locations, hundreds of experienced programmers and tens of managers, and so far he has found no case in which inspection results have been used negatively against programmers. Evidently no manager has tried to "kill the goose that lays the golden eggs."

A preinspection opinion of some programmers is that they do not see the value of inspections because they have managed very well up to now, or because their projects are too small or somehow different. This opinion usually changes after a few inspections to a position of acceptance. The quality of acceptance is related to the success of the inspections they have experienced, the *conduct of the trained moderator*, and the *attitude demonstrated by management*. The acceptance of inspections by programmers and managers as a beneficial step in making programs is well-established amongst those who have tried them.

المنارة للاستشارات

Process control using inspection and testing results

Obviously, the range of analysis possible using inspection results is enormous. Therefore, only a few aspects will be treated here, and they are elementary expositions.

most error-prone modules

A listing of either I_1, I_2, or combined $I_1 + I_2$ data as in Figure 8 immediately highlights which modules contained the highest error density on inspection. If the error detection efficiency of each of the inspections was fairly constant, the ranking of error-prone modules holds. Thus if the error detection efficiency of inspection is 50 percent, and the inspection found 10 errors in a module, then it can be estimated that there are 10 errors remaining in the module. This information can prompt many actions to control the process. For instance, in Figure 8, it may be decided to reinspect module "Echo" or to redesign and recode it entirely. Or, less drastically, it may be decided to test it "harder" than other modules and look especially for errors of the type found in the inspections.

distribution of error types

If a ranked distribution of error types is obtained for a group of "error-prone modules" (Figure 9), which were produced from the same Process A, for example, it is a short step to comparing this distribution with a "Normal/Usual Percentage Distribution." Large disparities between the sample and "standard" will lead to questions on why Process A, say, yields nearly twice as many internal interconnection errors as the "standard" process. If this analysis is done promptly on the first five percent of production, it may be possible to remedy the problem (if it is a problem) on the remaining 95 percent of modules for a particular shipment. Provision can be made to test the first five percent of the modules to remove the unusually high incidence of internal interconnection problems.

inspecting error-prone code

Analysis of the testing results, commencing as soon as testing errors are evident, is a vital step in controlling the process since future testing can be guided by early results.

Where testing reveals excessively error-prone code, it may be more economical and saving of schedule to select the most error-prone code and inspect it before continuing testing. (The business case will likely differ from project to project and case to case, but in many instances inspection will be indicated). The selection of the most error-prone code may be made with two considerations uppermost:

المنارة للاستشارات

Figure 8 Example of most error-prone modules based on I_1 and I_2

Module name	*Number of errors*	*Lines of code*	*Error density, Errors/K. Loc*
Echo	4	128	31
Zulu	10	323	31
Foxtrot	3	71	28
Alpha	7	264	27←Average
Lima	2	106	19 Error
Delta	3	195	15 Rate
⋮	⋮	⋮	⋮
	67		

Figure 9 Example of distribution of error types

	Number of errors	*%*	*Normal/usual distribution, %*
Logic	23	35	44
Interconnection/Linkage (Internal)	21	31 ?	18
Control Blocks	6	9	13
—	·	8	10
—	·	7	7
—	·	6	6
—	·	4	2
		100%	100%

1. Which modules head a ranked list when the modules are rated by test errors per K.NCSS?
2. In the parts of the program in which test coverage is low, which modules or parts of modules are most suspect based on $(I_1 + I_2)$ errors per K.NCSS and programmer judgment?

From a condensed table of ranked "most error-prone" modules, a selection of modules to be inspected (or reinspected) may be made. Knowledge of the error types already found in these modules will better prepare an inspection team.

The reinspection itself should conform with the I_2 process, except that an overview may be necessary if the original overview was held too long ago or if new project members are involved.

المنارة للاستشارات

Inspections and walk-throughs

Walk-throughs (or walk-thrus) are practiced in many different ways in different places, with varying regularity and thoroughness. This inconsistency causes the results of walk-throughs to vary widely and to be nonrepeatable. Inspections, however, having an established process and a formal procedure, tend to vary less and produce more repeatable results. Because of the variation in walk-throughs, a comparison between them and inspections is not simple. However, from Reference 8 and the walk-through procedures witnessed by the author and described to him by walk-through participants, as well as the inspection process described previously and in References 1 and 9, the comparison in Tables 4 and 5 is drawn.

effects on development process

Figure 10A describes the process in which a walk-through is applied. Clearly, the purging of errors from the product as it passes through the walk-through between Operations 1 and 2 is very beneficial to the product. In Figure 10B, the inspection process (and its feedback, feed-forward, and self-improvement) replaces the walk-through. The notes on the figure are self-explanatory.

Inspections are also an excellent means of measuring completeness of work against the exit criteria which must be satisfied to complete project checkpoints. (Each checkpoint should have a clearly defined set of exit criteria. Without exit criteria, a checkpoint is too negotiable to be useful for process control).

Inspections and process management

The most marked effects of inspections on the development process is to change the old adage that, "design is not complete until testing is completed," to a position where a very great deal must be known about the design before even the coding is begun. Although great discretion is still required in code implementation, more predictability and improvements in schedule, cost, and quality accrue. The old adage still holds true if one regards inspection as much a means of verification as testing.

percent of errors found

Observations in one case in systems programming show that approximately two thirds of all errors reported during development are found by I_1 and I_2 inspections prior to machine testing.

المنارة للاستشارات

Table 4. Inspection and walk-through processes and objectives

Inspection		*Walk-through*	
Process Operations	*Objectives*	*Process Operations*	*Objectives*
1. Overview	Education (Group)	–	–
2. Preparation	Education (Individual)	1. Preparation	Education (Individual)
3. Inspection	Find errors! (Group)	2. Walk-through	Education (Group) Discuss design alternatives Find errors
4. Rework	Fix problems	–	
5. Follow-up	Ensure all fixes correctly installed	–	

Note the separation of objectives in the inspection process.

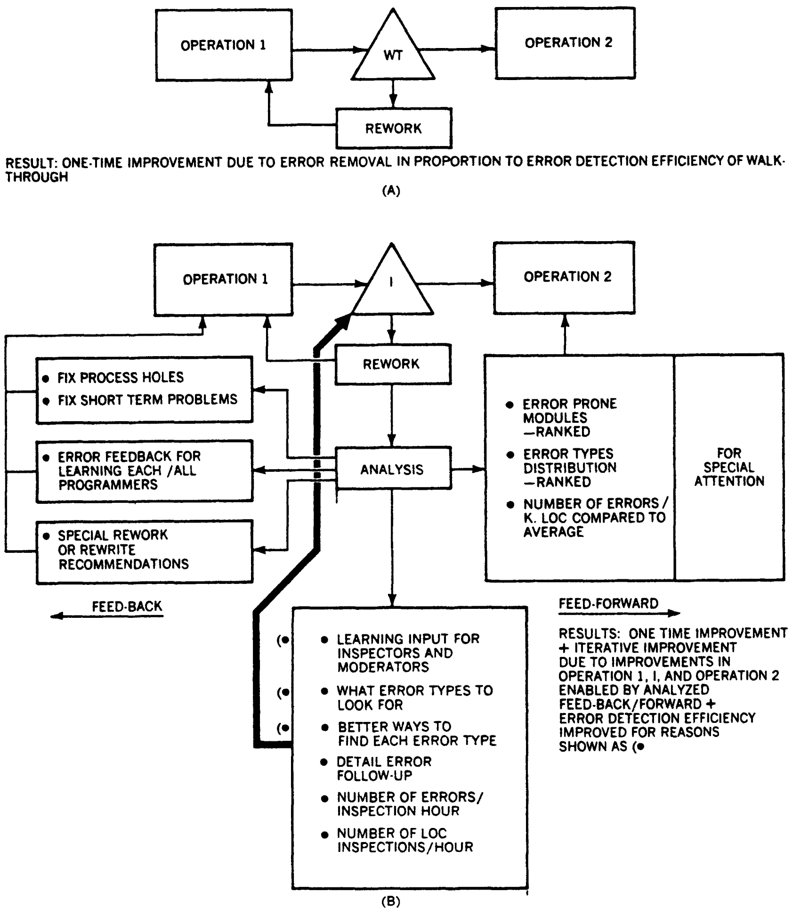
Table 5 Comparison of key properties of inspections and walk-throughs

Properties	*Inspection*	*Walk-Through*
1. Formal moderator training	Yes	No
2. Definite participant roles	Yes	No
3. Who "drives" the inspection or walk-through	Moderator	Owner of material (Designer or coder)
4. Use "How To Find Errors" checklists	Yes	No
5. Use distribution of error types to look for	Yes	No
6. Follow-up to reduce bad fixes	Yes	No
7. Less future errors because of detailed error feedback to individual programmer	Yes	Incidental
8. Improve inspection efficiency from analysis of results	Yes	No
9. Analysis of data → process problems → improvements	Yes	No

The error detection efficiencies of the I_1 and I_2 inspections separately are, of course, less than 66 percent. A similar observation of an application program development indicated an 82 percent find (Table 1). As more is learned and the error detection efficiency of inspection is increased, the burden of debugging on testing operations will be reduced, and testing will be more able to fulfill its prime objective of verifying quality.

المنارة للاستشارات

Figure 10 (A) Walk-through process, (B) Inspection process



effect on cost and schedule

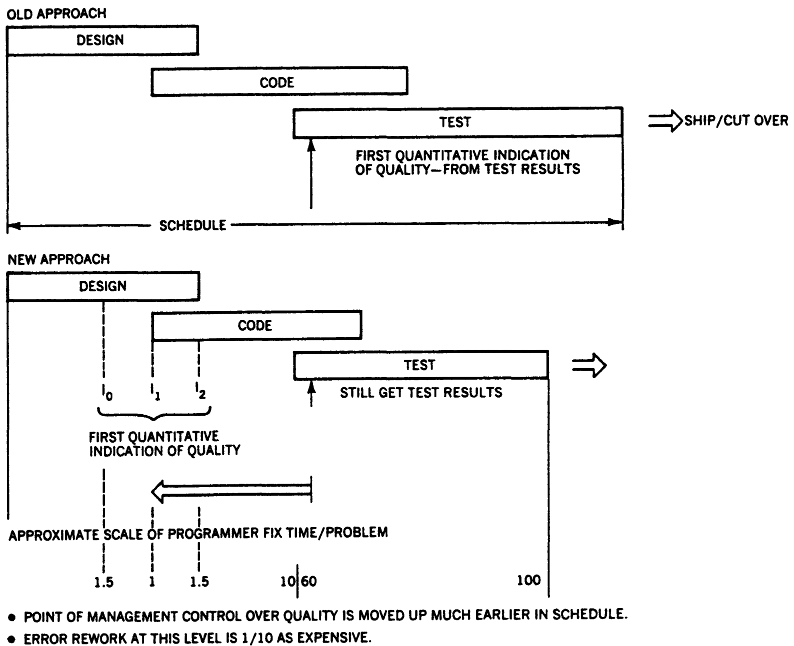
Comparing the "old" and "new" (with inspections) approaches to process management in Figure 11, we can see clearly that with the use of inspection results, error rework (which is a very significant variable in product cost) tends to be managed more during the first half of the schedule. This results in much lower cost than in the "old" approach, where the cost of error rework was 10 to 100 times higher and was accomplished in large part during the last half of the schedule.

process tracking

Inserting the I_1 and I_2 checkpoints in the development process enables assessment of project completeness and quality to be made early in the process (during the first half of the project instead of the latter half of the schedule, when recovery may be impossible without adjustments in schedule and cost). Since individually trackable modules of reasonably well-known size can

المنارة للاستشارات

Figure 11 Effect of inspection on process management



be counted as they pass through each of these checkpoints, the percentage completion of the project against schedule can be continuously and easily tracked.

effect on product knowledge

The overview, preparation, and inspection sequence of the operations of the inspection process give the inspection participants a high degree of product knowledge in a very short time. This important side benefit results in the participants being able to handle later development and testing with more certainty and less false starts. Naturally, this also contributes to productivity improvement.

An interesting sidelight is that because designers are asked at pre-I_1 inspection time for estimates of the number of lines of code (NCSS) that their designs will create, and they are present to count for themselves the actual lines of code at the I_2 inspection, the accuracy of design estimates has shown substantial improvement.

For this reason, an inspection is frequently a required event where responsibility for design or code is being transferred from

المنارة للاستشارات

one programmer to another. The complete inspection team is convened for such an inspection. (One-on-one reviews such as desk debugging are certainly worthwhile but do not approach the effectiveness of formal inspection.) Usually the side benefit of finding errors more than justifies the transfer inspection.

inspecting modified code

Code that is changed in, or inserted in, an existing module either in replacement of deleted code or simply inserted in the module is considered modified code. By this definition, a very large part of programming effort is devoted to modifying code. (The addition of entirely new modules to a system count as new, not modified, code.)

Some observations of errors per K.NCSS of modified code show its error rate to be considerably higher than is found in new code; (i.e., if 10.NCSS are replaced in a 100.NCSS module and errors against the 10.NCSS are counted, the error rate is described as number of errors per 10.NCSS, not number of errors per 100.NCSS). Obviously, if the number of errors in modified code are used to derive an error rate per K.NCSS for the whole module that was modified, this rate would be largely dependent upon the percentage of the module that is modified: this would provide a meaningless ratio. A useful measure is the number of errors per K.NCSS (modified) in which the higher error rates have been observed.

Since most modifications are small (e.g., 1 to 25 instructions), they are often erroneously regarded as trivially simple and are handled accordingly; the error rate goes up, and control is lost. In the author's experience, *all* modifications are well worth inspecting from an economic and a quality standpoint. A convenient method of handling changes is to group them to a module or set of modules and convene the inspection team to inspect as many changes as possible. But all changes must be inspected!

Inspections of modifications can range from inspecting the modified instructions and the surrounding instructions connecting it with its host module, to an inspection of the entire module. The choice of extent of inspection coverage is dependent upon the percentage of modification, pervasiveness of the modification, etc.

bad fixes

A very serious problem is the inclusion in the product of bad fixes. Human tendency is to consider the "fix," or correction, to a problem to be error-free itself. Unfortunately, this is all too frequently untrue in the case of fixes to errors found by inspections and by testing. The inspection process clearly has an oper-

المنارة للاستشارات

ation called Follow-Up to try and minimize the bad-fix problem, but the fix process of testing errors very rarely requires scrutiny of fix quality before the fix is inserted. Then, if the fix is bad, the whole elaborate process of going from source fix to link edit, to test the fix, to regression test must be repeated at needlessly high cost. The number of bad fixes can be economically reduced by some simple inspection after clean compilation of the fix.

Summary

We can summarize the discussion of design and code inspections and process control in developing programs as follows:

1. Describe the program development process in terms of operations, and define exit criteria which must be satisfied for completion of each operation.
2. Separate the objectives of the inspection process operations to keep the inspection team focused on one objective at a time:

Operation	*Objective*
Overview	Communications/education
Preparation	Education
Inspection	Find errors
Rework	Fix errors
Follow-up	Ensure all fixes are applied correctly

3. Classify errors by type, and rank frequency of occurrence of types. Identify *which types* to spend most time looking for in the inspection.
4. Describe *how* to look for presence of error types.
5. Analyze inspection results and use for constant process improvement (until process averages are reached and then use for process control).

Some applications of inspections include function level inspections I_0, design-complete inspections I_1, code inspections I_2, test plan inspections IT_1, test case inspections IT_2, interconnections inspections IF, inspection of fixes/changes, inspection of publications, etc., and post testing inspection. Inspections can be applied to the development of system control programs, applications programs, and microcode in hardware.

We can conclude from experience that inspections increase productivity and improve final program quality. Furthermore, improvements in process control and project management are enabled by inspections.

المنارة للاستشارات

ACKNOWLEDGMENTS

The author acknowledges, with thanks, the work of Mr. O. R. Kohli and Mr. R. A. Radice, who made considerable contributions in the development of inspection techniques applied to program design and code, and Mr. R. R. Larson, who adapted inspections to program testing.

CITED REFERENCES AND FOOTNOTES

1. O. R. Kohli, *High-Level Design Inspection Specification*, Technical Report TR 21.601, IBM Corporation, Kingston, New York (July 21, 1975).
2. It should be noted that the exit criteria for I_1 (design complete where one design statement is estimated to represent 3 to 10 code instructions) and I_2 (first clean code compilations) are checkpoints in the development process through which every programming project must pass.
3. The Hawthorne Effect is a psychological phenomenon usually experienced in human-involved productivity studies. The effect is manifested by participants producing above normal because they know they are being studied.
4. NCSS (Non-Commentary Source Statements), also referred to as "Lines of Code," are the sum of executable code instructions and declaratives. Instructions that invoke macros are counted once only. Expanded macroinstructions are also counted only once. Comments are not included.
5. Basically in a walk-through, program design or code is reviewed by a group of people gathered together at a structured meeting in which errors/issues pertaining to the material and proposed by the participants may be discussed in an effort to find errors. The group may consist of various participants but always includes the originator of the material being reviewed who usually plans the meeting and is responsible for correcting the errors. How it differs from an inspection is pointed out in Tables 2 and 3.

6. *Marketing Newsletter*, Cross Application Systems Marketing, "Program inspections at Aetna," MS-76-006, S2. IBM Corporation, Data Processing Division, White Plains, New York (March 29, 1976).
7. J. Ascoly, M. J. Cafferty, S. J. Gruen, and O. R. Kohli, *Code Inspection Specification*, Technical Report TR 21.630, IBM Corporation, Kingston, New York (1976).
8. N. S. Waldstein, *The Walk-Thru—A Method of Specification, Design and Review*, Technical Report TR 00.2536, IBM Corporation, Poughkeepsie, New York (June 4, 1974).
9. Independent study programs: *IBM Structured Programming Textbook*, SR20-7149-1, *IBM Structured Programming Workbook*, SR20-7150-0, IBM Corporation, Data Processing Division, White Plains, New York.

GENERAL REFERENCES

1. J. D. Aron, *The Program Development Process: Part 1: The Individual Programmer*, Structured Programs, 137–141, Addison-Wesley Publishing Co., Reading, Massachusetts (1974).

2. M. E. Fagan, *Design and Code Inspections and Process Control in the Development of Programs*, Technical Report TR 00.2763, IBM Corporation, Poughkeepsie, New York (June 10, 1976). This report is a revision of the author's *Design and Code Inspections and Process Control in the Development of Programs*, Technical Report TR 21.572, IBM Corporation, Kingston, New York (December 17, 1974).

المنارة للاستشارات

3. O. R. Kohli and R. A. Radice, *Low-Level Design Inspection Specification*, Technical Report TR 21.629. IBM Corporation, Kingston, New York (1976).

4. R. R. Larson, *Test Plan and Test Case Inspection Specifications*, Technical Report TR 21.586, IBM Corporation, Kingston, New York (April 4, 1975).

Appendix: Reporting forms and form completion instructions

Instructions for Completing Design Inspection Module Detail Form

This form (Figure 12) should be completed for each module/ macro that has valid problems against it. The problem-type information gathered in this report is important because a history of problem-type experience points out high-occurrence types. This knowledge can then be conveyed to inspectors so that they can concentrate on seeking the higher-occurrence types of problems.

Figure 12 Design inspection module detail form

DATE__________

DETAILED DESIGN INSPECTION REPORT

MODULE DETAIL

MOD/MAC:__________ SUBCOMPONENT/APPLICATION__________

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LO: LOGIC						
TB: TEST AND BRANCH						
DA: DATA AREA USAGE						
RM: RETURN CODES/MESSAGES						
RU: REGISTER USAGE						
MA: MODULE ATTRIBUTES						
EL: EXTERNAL LINKAGES						
MD: MORE DETAIL						
ST: STANDARDS						
PR: PROLOGUE OR PROSE						
HL: HIGHER LEVEL DESIGN DOC.						
US: USER SPEC.						
MN: MAINTAINABILITY						
PE: PERFORMANCE						
OT: OTHER						
TOTAL:						

REINSPECTION REQUIRED?__________

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION: A BUG. M = MISSING, W = WRONG, E = EXTRA.

NOTE. FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER. 3(2), WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE.

المنارة للاستشارات

1. MOD/MAC: The module or macro name.
2. SUBCOMPONENT: The associated subcomponent.
3. PROBLEM TYPE: Summarize the number of problems by type (logic, etc.), severity (major/minor), and by category (missing, wrong, or extra). For modified modules, detail the number of problems in the changed design versus the number in the base design. (Problem types were developed in a systems programming environment. Appropriate changes, if desired, could be made for application development.)
4. REINSPECTION REQUIRED?: Indicate whether the module/macro requires a reinspection.

All valid problems found in the inspection should be listed and attached to the report. A brief description of each problem, its error type, and the rework time to fix it should be given (see Figure 7A, which describes errors in similar detail to that required but is at a coding level).

Instructions for Completing Design Inspection Summary Form

Following are detailed instructions for completing the form in Figure 13.

1. TO: The report is addressed to the respective design and development managers.
2. SUBJECT: The unit being inspected is identified.
3. MOD/MAC NAME: The name of each module and macro as it resides on the source library.

Figure 13 Design inspection summary form

DESIGN INSPECTION REPORT
SUMMARY

Date________

To: Design Manager________ Development Manager________
Subject: Inspection Report for________ Inspection date________
System/Application________ Release________ Build________
Component________ Subcomponents(s)________

Mod/Mac Name	New or Mod	Full or Part Insp.	Detailed Designer	Programmer	ELOC Added, Modified. Deleted									Inspection People-hours (X.X.)				Subcomponent
					Est. Pre			Est. Post			Rework			Overview & Prep.	Insp Meetg	Re-work	Follow-up	
					A	M	D	A	M	D	A	M	D					
				Totals														

Reinspection required?________ Length of inspection (clock hours and tenths)________
Reinspection by (date)________ Additional modules/macros?________
DCR #'s written________
Problem summary: Major________ Minor________ Total________
Errors in changed code: Major____ Minor____ Errors in base code: Major____ Minor____

Initial Desr	Detailed Dr	Programmer	Team Leader	Other	Moderator's Signature

المنارة للاستشارات

4. NEW OR MOD: "N" if the module is new; "M" if the module is modified.
5. FULL OR PART INSP: If the module/macro is "modified," indicate "F" if the module/macro was fully inspected or "P" if partially inspected.
6. DETAILED DESIGNER: and PROGRAMMER: Identification of originators.
7. PRE-INSP EST ELOC: The estimated executable source lines of code (added, modified, deleted). Estimate made prior to the inspection by the designer.
8. POST-INSP EST ELOC: The estimated executable source lines of code. Estimate made after the inspection.
9. REWORK ELOC: The estimated executable source lines of code in rework as a result of the inspection.
10. OVERVIEW AND PREP: The number of people-hours (in tenths of hours) spent in preparing for the overview, in the overview meeting itself, and in preparing for the inspection meeting.
11. INSPECTION MEETING: The number of people-hours spent on the inspection meeting.
12. REWORK: The estimated number of people-hours spent to fix the problems found during the inspection.
13. FOLLOW-UP: The estimated number of people-hours spent by the moderator (and others if necessary) in verifying the correctness of changes made by the author as a result of the inspection.
14. SUBCOMPONENT: The subcomponent of which the module/macro is a part.
15. REINSPECTION REQUIRED?: Yes or no.
16. LENGTH OF INSPECTION: Clock hours spent in the inspection meeting.
17. REINSPECTION BY (DATE): Latest acceptable date for reinspection.
18. ADDITIONAL MODULES/MACROS?: For these subcomponents, are additional modules/macros yet to be inspected?
19. DCR #'S WRITTEN: The identification of Design Change Requests, DCR(s), written to cover problems in rework.
20. PROBLEM SUMMARY: Totals taken from Module Detail forms(s).
21. INITIAL DESIGNER, DETAILED DESIGNER, etc.: Identification of members of the inspection team.

المنارة للاستشارات

Instructions for Completing Code Inspection Module Detail Form

This form (Figure 14) should be completed according to the instructions for completing the design inspection module detail form.

Instructions for Completing Code Inspection Summary Form

This form (Figure 15) should be completed according to the instructions for the design inspection summary form except for the following items.

1. PROGRAMMER AND TESTER: Identifications of original participants involved with code.
2. PRE-INSP. ELOC: The noncommentary source lines of code (added, modified, deleted). Count made prior to the inspection by the programmer.
3. POST-INSP EST ELOC: The estimated noncommentary source lines of code. Estimate made after the inspection.

Figure 14 Code inspection module detail form

DATE__________

CODE INSPECTION REPORT

MODULE DETAIL

MOD/MAC:__________ SUBCOMPONENT/APPLICATION__________

SEE NOTE BELOW

	MAJOR*			MINOR		
PROBLEM TYPE:	M	W	E	M	W	E
LO: LOGIC						
TB: TEST AND BRANCH						
EL: EXTERNAL LINKAGES						
RU: REGISTER USAGE						
SU: STORAGE USAGE						
DA: DATA AREA USAGE						
PU: PROGRAM LANGUAGE						
PE: PERFORMANCE						
MN: MAINTAINABILITY						
DE: DESIGN ERROR						
PR: PROLOGUE						
CC: CODE COMMENTS						
OT: OTHER						
TOTAL:						

REINSPECTION REQUIRED?__________

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION· A BUG M = MISSING, W = WRONG, E = EXTRA.
NOTE: FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER· 3(2). WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE

Figure 15 Code inspection summary form

CODE INSPECTION REPORT
SUMMARY

Date________

To: Design Manager________ Development Manager________

Subject: Inspection Report for________ Inspection date________

System/Application________ Release________ Build________

Component________ Subcomponents(s)________

					ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				
					Pre-insp			Est Post			Rework							
Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	A	M	D	A	M	D	A	M	D	Prep	Insp Meetg	Re-work	Follow-up	Sub-component
				Totals														

Reinspection required?________ Length of inspection (clock hours and tenths)________

Reinspection by (date)________ Additional modules/macros?________

DCR #'s written________

Problem summary: Major________ Minor________ Total________

Errors in changed code: Major____ Minor____ Errors in base code: Major____ Minor____

Initial Desr | Detailed Dr | Programmer | Team Leader | Other | Moderator's Signature

4. REWORK ELOC: The estimated noncommentary source lines of code in rework as a result of the inspection.
5. PREP: The number of people hours (in tenths of hours) spent in preparing for the inspection meeting.

Reprint Order No. G321-5033.

المنارة للاستشارات

Michael Fagan
Advances in Software Inspections

IEEE Transactions on Software Engineering, Vol. SE-12 (7), 1986
pp. 744–751

المنارة للاستشارات

Advances in Software Inspections

MICHAEL E. FAGAN, MEMBER, IEEE

Manuscript received September 30, 1985.
The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
IEEE Log Number 8608192.

***Abstract*—This paper presents new studies and experiences that enhance the use of the inspection process and improve its contribution to development of defect-free software on time and at lower costs. Examples of benefits are cited followed by descriptions of the process and some methods of obtaining the enhanced results.**

Software inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product—and at lower cost—than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Excellent results have been obtained by small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.

***Index Terms*—Defect detection, inspection, project management, quality assurance, software development, software engineering, software quality, testing, walkthru.**

INTRODUCTION

THE software inspection process was created in 1972, in IBM Kingston, NY, for the dual purposes of improving software quality and increasing programmer productivity. Its accelerating rate of adoption throughout the software development and maintenance industry is an ac-

المنارة للاستشارات

knowledgment of its effectiveness in meeting its goals. Outlined in this paper are some enhancements to the inspection process, and the experiences of some of the many companies and organizations that have contributed to its evolution. The author is indebted to and thanks the many people who have given their help so liberally.

Because of the clear structure the inspection process has brought to the development process, it has enabled study of both itself and the conduct of development. The latter has enabled process control to be applied from the point at which the requirements are inspected—a much earlier point in the process than ever before—and throughout development. Inspections provide data on the performance of individual development operations, thus providing a unique opportunity to evaluate new tools and techniques. At the same time, studies of inspections have isolated and fostered improvement of its key characteristics such that very high defect detection efficiency inspections may now be conducted *routinely*. This simultaneous study of development and design and code inspections prompted the adaptation of the principles of the inspection process to inspections of requirements, user information, and documentation, and test plans and test cases. In each instance, the new uses of inspection were found to improve product quality and to be cost effective, i.e., it saved more than it cost. Thus, as the effectiveness of inspections are improving, they are being applied in many new and different ways to improve software quality and reduce costs.

Benefits: Defect Reduction, Defect Prevention, and Cost Improvement

In March 1984, while addressing the IBM SHARE User Group on software service, L. H. Fenton, IBM Director of VM Programming Systems, made an important statement on quality improvement due to inspections [1]:

المنارة للاستشارات

> "Our goal is to provide defect free products and product information, and we believe the best way to do this is by refining and enhancing our existing software development process.
>
> Since we introduced the inspection process in 1974, we have achieved significant improvements in quality. IBM has nearly doubled the number of lines of code shipped for System/370 software products since 1976, while the number of defects per thousand lines of code has been reduced by *two-thirds*. Feedback from early MVS/XA and VM/SP Release 3 users indicates these products met and, in many cases, exceeded our ever increasing quality expectations."

Observation of a small sample of programmers suggested that early experience gained from inspections caused programmers to reduce the number of defects that were injected in the design and code of programs created later during the same project [3]. Preliminary analysis of a much larger study of data from recent inspections is providing similar results.

It should be noted that the improvements reported by IBM were made while many of the enhancements to inspections that are mentioned here were being developed. As these improvements are incorporated into everyday practice, it is probable that inspections will help bring further reductions in defect injection and detection rates.

Additional reports showing that inspections improve quality *and* reduce costs follow. (In all these cases, the cost of inspections is included in project cost. Typically, all design and code inspection costs amount to 15 percent of project cost.)

المنارة للاستشارات

Project	Results
AETNA Life and Casualty. 4439 LOC [2]	—0 Defects in use. —25 percent reduction in development resource.
IBM RESPOND, U.K. 6271 LOC [3]	—0 Defects in use. —9 percent reduction in cost compared to walkthrus.
Standard Bank of South Africa. 143 000 LOC [4]	—0.15 Defects/KLOC in use. —95 percent reduction in corrective maintenance cost.
American Express, System code). 13 000 LOC	—0.3 Defects in use.

In the AETNA and IBM examples, inspections found 82 and 93 percent, respectively, of all defects (that would cause malfunction) detected over the life cycle of the products. The other two cases each found over 50 percent of all defects by inspection. While the Standard Bank of South Africa and American Express were unable to use trained inspection moderators, and the former conducted only code inspections, both obtained outstanding results. The tremendous reduction in corrective maintenance at the Standard Bank of South Africa would also bring impressive savings in life cycle costs.

Naturally, reduction in maintenance allows redirection of programmers to work off the application backlog, which is reputed to contain at least two years of work at most locations. Impressive cost savings and quality improvements have been realized by inspecting test plans and then the test cases that implement those test plans. For a product of about 20 000 LOC, R. Larson [5] reported that test inspections resulted in:

المنارة للاستشارات

• modification of approximately 30 percent of the functional matrices representing test coverage,

• detection of 176 major defects in the test plans and test cases (i.e., in 176 instances testing would have missed testing critical function or tested it incorrectly), and

• savings of more than 85 percent in programmer time by detecting the major defects by inspection as opposed to finding them during functional variation testing.

There are those who would use inspections whether or not they are cost justified for defect removal because of the nonquantifiable benefits the technique supplies toward improving the service provided to users and toward creating a more professional application development environment [6].

Experience has shown that inspections have the effect of slightly front-end loading the committment of people resources in development, adding to requirements and design, while greatly reducing the effort required during testing and for rework of design and code. The result is an overall *net* reduction in development resource, and usually in schedule too. Fig. 1 is a pictorial description of the familiar "snail" shaped curve of software development resource versus the time schedule including and without inspections.

The Software Quality Problem

The software quality problem is the result of defects in code and documentation causing failure to satisfy user requirements. It also impedes the growth of the information processing industry. Validity of this statement is attested to by three of the many pieces of supporting evidence:

• The SHARE User Group Software Service Task Force Report, 1983 [1], that recommended an order of magnitude improvement in software quality over the next

المنارة للاستشارات

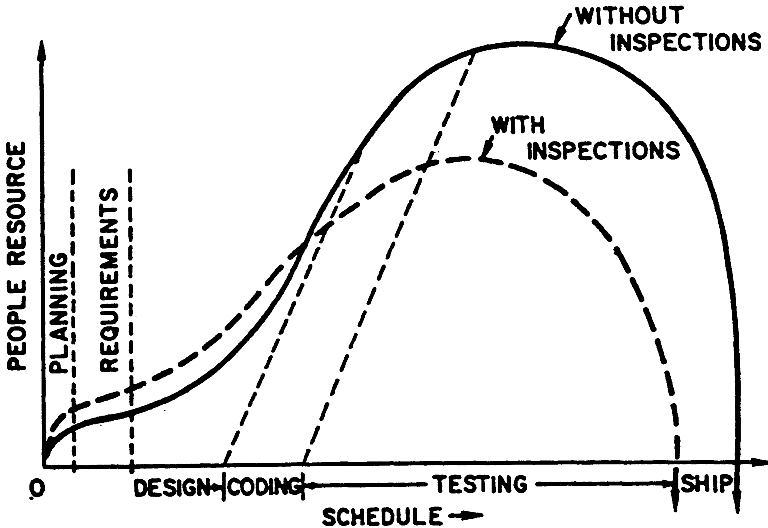


Fig. 1.

several years, with a like reduction in service. (Other manufacturers report similar recommendations from their users.)

- In 1979, 12 percent of programmer resource was consumed in post-shipment corrective maintenance alone and this figure was growing [8]. (Note that there is also a significant percentage of development and enhancement maintenance resource devoted to correcting defects. This is probably larger than the 12 percent expended in corrective maintenance, but there is no substantiating research.)
- The formal backlog of data processing tasks most quoted is three years [7].

At this point, a very important definition is in order:

A defect is an instance in which a requirement is not satisfied.

Here, it must be recognized that a requirement is any agreed upon commitment. It is not only the recognizable

المنارة للاستشارات

external product requirement, but can also include internal development requirements (e.g., the exit criteria of an operation) that must be met in order to satisfy the requirements of the end product. Examples of this would be the requirement that a test plan completely verifies that the product meets the agreed upon needs of the user, or that the code of a program must be complete before it is submitted to be tested.

While defects become manifest in the end product documentation or code, most of them are actually injected as the functional aspects of the product and its quality attributes are being created; during development of the requirements, the design and coding, or by insertion of changes. The author's research supports and supplements that of B. Boehm *et al.* [9] and indicates that there are eight attributes that must be considered when describing quality in a software product:

- intrinsic code quality,
- freedom from problems in operation,
- usability,
- installability,
- documentation for intended users,
- portability,
- maintainability and extendability, and "fitness for use"—that implicit conventional user needs are satisfied.

Inspections and the Software Quality Problem

Previously, each of these attributes of software quality were evaluated by testing and the end user. Now, some of them are being partly, and others entirely, verified against requirements by inspection. In fact, the product requirements themselves are often inspected to ascertain whether they meet user needs. In order to eliminate defects from the product it is necessary to address their prevention, or detection and resolution as soon as possible

للاستشارات

after their injection during development and maintenance. Prevention is the most desirable course to follow, and it is approached in many ways including the use of state machine representation of design, systematic programming, proof of correctness, process control, development standards, prototyping, and other methods. Defect detection, on the other hand, was once almost totally dependent upon testing during development and by the user. This has changed, and over the past decade walkthrus and inspections have assumed a large part of the defect detection burden; inspections finding from 60 to 90 percent defects. (See [2], [3], and other unpublished product experiences.) They are performed much nearer the point of injection of the defects than is testing, using less resource for rework and, thus, more than paying for themselves. In fact, inspections have been applied to most phases of development to verify that the key software attributes are present immediately after the point at which they should first be introduced into the product. They are also applied to test plans and test cases to improve the defect detection efficiency of testing. Thus, inspections have been instrumental in improving all aspects of software product quality, as well as the quality of logic design and code. In fact, inspections *supplement* defect prevention methods in improving quality.

Essential to the quality of inspection (or its defect detection efficiency) is proper definition of the development process. And, inspection quality is a direct contributor to product quality, as will be shown later.

Definition of the Development Process

The software development process is a series of operations so arranged that its execution will deliver the desired end product. Typically, these operations are: Requirements Definition, System Design, High Level

المنارة للاستشارات

Design, Low Level Design, Coding, Unit Testing, Component or Function Testing, System Testing, and then user support and Maintenance. In practice, some of these operations are repeated as the product is recycled through them to insert functional changes and fixes.

The attributes of software quality are invested along with the functional characteristics of the product during the early operations, when the cost to remedy defects is 10–100 times less than it would be during testing or maintenance [2]. Consequently, it is advantageous to find and correct defects as near to their point of origin as possible. This is accomplished by inspecting the output product of each operation to verify that it satisfies the output requirements or *exit criteria* of the operation. In most cases, these exit criteria are not specified with sufficient precision to allow go/no verification. Specification of exit criteria in unambiguous terms that are objective and preferably quantitative is an essential characteristic of any well defined process. Exit criteria are the standard against which inspections measure completion of the product at the end of an operation, and verify the presence or absence of quality attributes. (A deviation from exit criteria is a defect.)

Shown below are the essence of 4 key criteria taken from the full set of 15 exit criteria items for the Coding operation:

- The source code must be at the "first clean compilation" level. That means it must be properly compiled and be free of syntax errors.
- The code must accurately implement the low level design (which was the verified output of the preceding process operation).
- All design changes to date are included in the code.
- All rework resulting from the code inspection has been included and verified.

المنارة للاستشارات

The code inspection, I2, must verify that all 15 of these exit criteria have been satisfied before a module or other entity of the product is considered to have completed the Coding operation. Explicit exit criteria for several of the other inspection types in use will be contained in the author's book in software inspections. However, there is no reason why a particular project could not define its own sets of exit criteria. What is important is that exit criteria should be as objective as possible, so as to be repeatable; they should completely describe what is required to exit each operation; and, *they must be observed by all those involved*.

The objective of process control is to measure completion of the product during stages of its development, to compare the measurement against the project plan, and then to remedy any deviations from plan. In this context, the quality of both exit criteria and inspections are of vital importance. And, they must both be properly described in the manageable development process, for such a process must be controllable by definition.

Development is often considered a subset of the maintenance process. Therefore, the maintenance process must be treated in the same manner to make it equally manageable.

Software Inspection Overview

This paper will only give an overview description of the inspection process that is sufficient to enable discussion of updates and enhancements. The author's original paper on the software inspections process [2] gives a brief description of the inspection process and what goes on in an inspection, and is the base to which the enhancements are added. His forthcoming companion books on this subject and on building defect-free software will provide an implementation level description and will include all the points addressed in this paper and more.

المنارة للاستشارات

To convey the principles of software inspections, it is only really necessary to understand how they apply to design and code. A good grasp on this application allows tailoring of the process to enable inspection of virtually any operation in development or maintenance, and also allows inspection for any desired quality attribute. With this in mind, the main points of inspections will be exposed through discussing how they apply in design and code inspections.

There are three essential requirements for the implementation of inspections:

- definition of the DEVELOPMENT PROCESS in terms of operations and their EXIT CRITERIA,
- proper DESCRIPTION of the INSPECTION PROCESS, and
- CORRECT EXECUTION of the INSPECTION PROCESS. (Yes, correct execution of the process is vital.)

The Inspection Process

The inspection process follows any development operation whose product must be verified. As shown below, it consists of six operations, each with a specific objective:

Operation	Objectives
PLANNING	Materials to be inspected must meet inspection entry criteria.
	Arrange the availability of the right participants.
	Arrange suitable meeting place and time.
OVERVIEW	Group education of participants in what is to be inspected.
	Assign inspection roles to participants.

المنارة للاستشارات

PREPARATION	Participants learn the material and prepare to fulfill their assigned roles.
INSPECTION	*Find defects.* (Solution hunting and discussion of design alternatives is discouraged.)
REWORK	The author reworks all defects.
FOLLOW-UP	Verification by the inspection moderator or the entire inspection team to assure that all fixes are effective and that no secondary defects have been introduced.

Evaluation of hundreds of inspections involving thousands of programmers in which alternatives to the above steps have been tried has shown that all these operations are really necessary. Omitting or combining operations has led to degraded inspection efficiency that outweighs the apparent short-term benefits. OVERVIEW is the only operation that under certain conditions can be omitted with slight risk. Even FOLLOW-UP is justified as study has shown that approximately one of every six fixes are themselves incorrect, or create other defects.

From observing scores of inspections, it is evident that participation in inspection teams is extremely taxing and should be limited to periods of 2 hours. Continuing beyond 2 hours, the defect detection ability of the team seems to diminish, but is restored after a break of 2 hours or so during which other work may be done. Accordingly, no more than two 2 hour sessions of inspection per day are recommended.

To assist the inspectors in finding defects, for not all inspectors start off being good detectives, a checklist of defect types is created to help them identify defects appropriate to the exit criteria of each operation whose product is to be inspected. It also serves as a guide to classi-

المنارة للاستشارات

fication of defects found by inspection prior to their entry to the inspection and test defect data base of the project. (A database containing these and other data is necessary for quality control of development.)

People and Inspections

Inspection participants are usually programmers who are drawn from the project involved. The roles they play for design and code inspections are those of the *Author* (Designer or Coder), *Reader* (who paraphrases the design or code as if they will implement it), *Tester* (who views the product from the testing standpoint), and *Moderator*. These roles are described more fully in [2], but that level of detail is not required here. Some inspections types, for instance those of system structure, may require more participants, but it is advantageous to keep the number of people to a minimum. Involving the end users in those inspections in which they can truly participate is also very helpful.

The Inspection Moderator is a *key player* and *requires special training* to be able to conduct inspections that are optimally effective. Ideally, to preserve objectivity, the moderator should not be involved in development of the product that is to be inspected, but should come from another similar project. The moderator functions as a "player-coach" and is responsible for conducting the inspection so as to bring a peak of synergy from the group. This is a quickly learned ability by those with some interpersonal skill. In fact, when participants in the moderator training classes are questioned about their case studies, they invariably say that they sensed the presence of the "*Phantom Inspector*," who materialized as a feeling that there had been an additional presence contributed by the way the inspection team worked together. The moderator's task is to invite the Phantom Inspector.

المنارة للاستشارات

When they are properly approached by management, programmers respond well to inspections. In fact, after they become familiar with them, many programmers have been known to complain when they were not allowed enough time or appropriate help to conduct inspections correctly.

Three separate classes of education have been recognized as a necessity for proper long lasting implementation of inspections. First, *Management* requires a class of one day to familiarize them with inspections and their benefits to management, and *their role* in making them successful. Next, the *Moderators* need three days of education. And, finally, the other *Participants* should receive one half day of training on inspections, the benefits, and their roles. Some organizations have started inspections without proper education and have achieved some success, but less than others who prepared their participants fully. This has caused some amount of start-over, which was frustrating to everyone involved.

Management and Inspections

A definite philosophy and set of attitudes regarding inspections and their results is essential. The management education class on inspections is one of the best ways found to gain the knowledge that must be built into day-to-day management behavior that is required to get the most from inspections on a continuing basis. For example, management must show encouragement for proper inspections. Requiring inspections and then asking for shortcuts will not do. And, people must be motivated to find defects by inspection. *Inspection results must never be used for personnel performance appraisal*. However, the results of testing should be used for performance appraisal. This promotes finding and reworking defects at the lowest cost, and allows testing for verification instead

المنارة للاستشارات

of debugging. In most situations programmers come to depend upon inspections; they prefer defect-free product. And, at those installations where management has taken and maintained a leadership role with inspections, they have been well accepted and very successful.

Inspection Results and Their Uses

The defects found by inspection are immediately recorded and classified by the moderator before being entered into the project data base. Here is an example:

In module: *XXX*, Line: *YYY*, NAME-CHECK is performed one less time than required—LO/W/MAJ

The description of the defect is obvious. The classification on the right means that this is a defect in Logic, that the logic is Wrong (as opposed to Missing or Extra), and that it is a Major defect. A MAJOR defect is one that would cause a malfunction or unexpected result if left uncorrected. Inspections also find MINOR defects. They will not cause malfunction, but are more of the nature of poor workmanship, like misspellings that do not lead to erroneous product performance.

Major defects are of the same type as defects found by testing. (One unpublished study of defects found by system testing showed that more than 87 percent could have been detected by inspection.) Because Major defects are equivalent to test defects, inspection results can be used to identify *defect prone design and code*. This is enabled because empirical data indicates a directly proportional relationship between the inspection detected defect rate in a piece of code and the defect rate found in it by subsequent testing. Using inspection results in this way, it is possible to identify defect prone code and correct it, in effect, performing real-time quality control of the product as it is being developed, *before it is shipped or put into use*.

المنارة للاستشارات

There are, of course, many Process and Quality Control uses for inspection data including:

- Feedback to improve the development process by identification and correction of the root causes of systematic defects before more code is developed;
- *Feed-forward* to prepare the process ahead to handle problems *or to evaluate corrective action in advance* (e.g., handling defect prone code);
- Continuing improvement and control of inspections.

An outstanding benefit of feedback, as reported in [3] was that designers and coders through involvement in inspections of their own work learned to find defects they had created more easily. This enabled them to *avoid* causing these defects in future work, thus providing much higher quality product.

Various Applications of Inspections

The inspection process was originally applied to hardware logic, and then to software logic design and code. It was in the latter case that it first gained notice. Since then it has been very successfully applied to software test plans and test cases, user documentation, high level design, system structure design, design changes, requirements development, and microcode. It has also been employed for special purposes such as cleaning up defect prone code, and improving the quality of code that has already been tested. And, finally, it has been resurrected to produce defect-free hardware. It appears that virtually anything that is created by a development process and that can be made visible and readable can be inspected. All that is necessary for an inspection is to define the exit criteria of the process operation that will make the product to be inspected, tailor the inspection defect checklists to the particular product and exit criteria, and then to execute the inspection process.

المنارة للاستشارات

What's in a Name?

In contrast to inspections, walkthrus, which can range anywhere from cursory peer reviews to inspections, do not usually practice a process that is repeatable or collect data (as with inspections), and hence this process cannot be reasonably studied and improved. Consequently, their defect detection efficiencies are usually quite variable and, when studied, were found to be much lower than those of inspections [2], [3]. However, the name "walkthru" (or "walkthrough") has a place, for in some management and national cultures it is more desirable than the term "inspection" and, in fact, the walkthrus in *some* of these situations are identical to formal inspections. (In almost all instances, however, the author's experience has been that the term walkthru has been accurately applied to the less efficient method—which process is actually in use can be readily determined by examining whether a formally defined development process with exit criteria is in effect, and by applying the criteria in [2, Table 5] to the activity. In addition, initiating walkthrus as a migration path to inspections has led to a lot of frustration in many organizations because once they start with the informal, they seem to have much more difficulty moving to the formal process than do those that introduce inspections from the start. And, programmers involved in inspections are usually more pleased with the results. In fact, their major complaints are generally to do with things that detract from inspection quality.) What is important is that the same results should not be expected of walkthrus as is required of inspections, *unless a close scrutiny proves the process and conduct of the "walkthru" is identical to that required for inspections*. Therefore, although walkthrus do serve very useful though limited functions, they are not discussed further in this paper.

المنارة للاستشارات

Recognizing many of the abovementioned points, the IBM Information Systems Management Institute course on this subject is named: "Inspections: Formal Application Walkthroughs." They teach about inspection.

CONTRIBUTORS TO SOFTWARE INSPECTION QUALITY

Quality of inspection is defined as its ability to detect all instances in which the product does not meet its requirements. Studies, evaluations, and the observations of many people who have been involved in inspections over the past decade provide insights into the contributors to inspection quality. Listing contributors is of little value in trying to manage them as many have relationships with each other. These relationships must be understood in order to isolate and deal with initiating root causes of problems rather than to waste effort dealing with symptoms. The ISHIKAWA or FISHBONE CAUSE/EFFECT DIAGRAM [11], shown in Fig. 2, shows the contributors and their cause/effect relationships.

As depicted in Fig. 2, the main contributors, shown as main branches on the diagram, are: *PRODUCT INSPECTABILITY, INSPECTION PROCESS, MANAGERS,* and *PROGRAMMERS*. Subcontributors, like *INSPECTION MATERIALS* and *CONFORMS WITH STANDARDS*, which contribute to the *PRODUCT INSPECTABILITY*, are shown as twigs on these branches. Contributors to the subcontributors are handled similarly. Several of the relationships have been proven by objective statistical analysis, others are supported by empirical data, and some are evident from project experience. For example, one set of relationships very thoroughly established in a controlled study by F. O. Buck, in "Indicators of Quality Inspections" [10], are:

- excessive SIZE OF MATERIALS to be inspected leads to a PREPARATION RATE that is too high.

المنارة للاستشارات

• PREPARATION RATE that is too high contributes to an excessive RATE OF INSPECTION, and

• Excessive RATE OF INSPECTION *causes fewer defects to be found.*

This study indicated that the following rates should be used in planning the I2 code inspection:

OVERVIEW:	500 Noncommentary Source Statements per Hour.
PREPARATION:	125 Noncommentary Source Statements per Hour.
INSPECTION:	90 Noncommentary Source Statements per Hour.
Maximum Inspection Rate:	125 Noncommentary Source Statements per Hour.

The rate of inspection seems tied to the thoroughness of the inspection, and there is evidence that defect detection efficiency diminishes at rates above 125 NCSS/h. (Many projects require reinspection if this maximum rate is exceeded, and the reinspection usually finds more defects.) Separate from this study, project data show that inspections conducted by trained moderators are very much more likely to approximate the permissible inspection rates, and yield higher quality product than moderators who have not been trained. Meeting this rate is not a direct conscious purpose of the moderator, but rather is the result of proper conduct of the inspection. In any event, as the study shows, requiring too much material to be inspected will induce insufficient PREPARATION which, in turn, will cause the INSPECTION to be conducted too fast. Therefore, it is the responsibility of man-

المنارة للاستشارات

agement and the moderator to start off with a plan that will lead to successful inspection.

The planning rate for high level design inspection of *systems design* is approximately twice the rate for code inspection, and low level (Logic) design inspection is nearly the same (rates are based upon the designer's estimate of the number of source lines of code that will be needed to implement the design). Both these rates *may* depend upon the complexity of the material to be inspected and the manner in which it is prepared (e.g., unstructured code is more difficult to read and requires the inspection rate to be lowered. Faster inspection rates while retaining high defect detection efficiency *may* be feasible with highly structured, easy to understand material, *but further study is needed*). Inspections of requirements, test plans, and user documentation are governed by the same rules as for code inspection, although inspection rates are not as clear for them and are probably more product and project dependent than is the case of code.

With a good knowledge of and attention to the contributors to inspection quality, management can profoundly influence the quality, and the development and maintenance costs of the products for which they are responsible.

SUMMARY

Experience over the past decade has shown software inspections to be a potent defect detection method, finding 60–90 percent of all defects, as well as providing feedback that enables programmers to avoid injecting defects in future work. As well as providing checkpoints to facilitate process management, inspections enable measurement of the performance of many tools and techniques in individual process operations. Because inspection engages similar skills to those used in creating the

المنارة للاستشارات

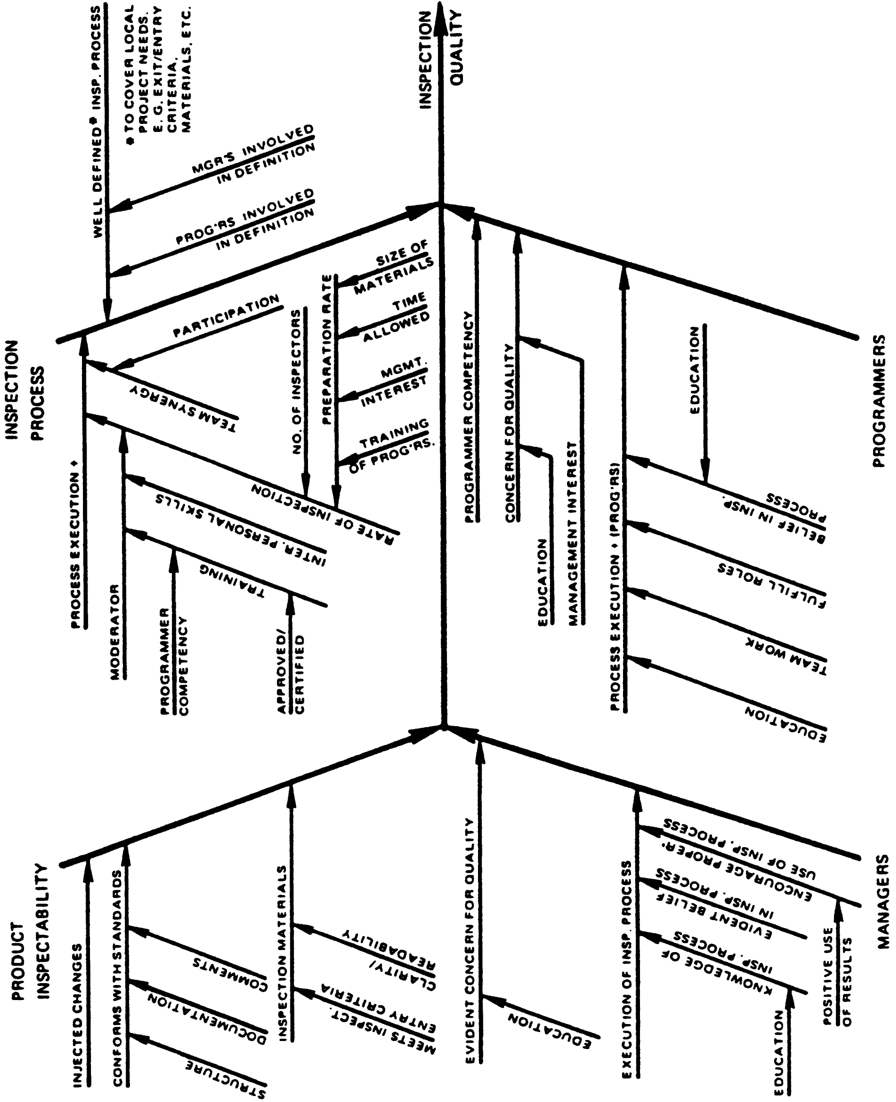


Fig. 2. Fishbone diagram of contributors to inspection quality.

المنارة للاستشارات

product (and it has been applied to virtually every design technique and coding language), it appears that anything that can be created and described can also be inspected.

Study and observation have revealed the following key aspects that must be managed to take full advantage of the many benefits that inspections offer:

Capability	Action Needed to Enhance the Capability
• Defect Detection	— Management understanding and continuing support. This starts with education.
	— Inspection moderator training (3 days).
	— Programmer training.
	— Continuing management of the contributors to inspection quality.
	— Inspect all changes.
	— Periodic review of effectiveness by management.
	— Inspect test plans and test cases.
	— Apply inspections to main defect generating operations in development *and* maintenance processes.
• Defect Prevention (or avoidance)	— Encourage programmers to understand how they created defects and what must be done to avoid them in future.
	— Feedback inspection results promptly and removes root causes of systematic de-

fects from the development or maintenance processes.
— Provide inspection results to quality circles or quality improvement teams.
— Creation of requirements for expert system tools (for defect prevention) based upon analysis of inspection data.

- Process Management — Use inspection completions as checkpoints in the development plan and measure accomplishment against them.

References

[1] L. H. Fenton, "Response to the SHARE software service task force report," IBM Corp., Kingston, NY, Mar. 6, 1984.

[2] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, 1979.

[3] *IBM Technical Newsletter GN20-3814*, Base Publication GC20-2000-0, Aug. 15, 1978.

[4] T. D. Crossman, "Inspection teams, are they worth it?" in *Proc. 2nd Nat. Symp. EDP Quality Assurance*, Chicago, IL, Mar. 24–26, 1982.

[5] R. R. Larson, "Test plan and test case inspection specification," IBM Corp., Tech. Rep. TR21.585, Apr. 4, 1975.

[6] T. D. Crossman, "Some experiences in the use of inspection teams in application development," in *Proc. Applicat. Develop. Symp.*, Monterey, CA, 1979.

[7] G. D. Brown and D. H. Sefton, "The micro vs. the applications logjam," *Datamation*, Jan. 1984.

[8] J. H. Morrissey and L. S.-Y. Wu, "Software engineering: An economical perspective," in *Proc. IEEE Conf. Software Eng.*, Munich, West Germany, Sept. 14–19, 1979.

[9] B. Boehm *et al.*, *Characteristics of Software Quality.* New York: American Elsevier, 1978.

[10] F. O. Buck, "Indicators of quality inspections," IBM Corp., Tech. Rep. IBM TR21.802, Sept. 1981.

[11] K. Ishikawa, *Guide to Quality Control.* Tokyo, Japan: Asian Productivity Organization, 1982.

المنارة للاستشارات

Michael E. Fagan (M'62) is a Senior Technical Staff Member at the IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY. While at IBM, he has had many interesting management and technical assignments in the fields of engineering, manufacturing, software development, and research. In 1972, he created the software inspection process, and has helped implement it within IBM and also promoted its use in the software industry. For this and other work, he has received IBM Outstanding Contribution and Corporate Achievement Awards. His area of interest is in studying and improving all the processes that comprise the software life cycle. For the past two years, he has been a Visiting Professor at, and is on the graduate council of, the University of Maryland.

المنارة للاستشارات

Erich Gamma

Erich Gamma, Richard Helm, Ralph Johnson,
John Vlissides
Design Patterns: Abstraction and Reuse
of Object-Oriented Design

Proceedings ECOOP'93, Kaiserslautern, July 1993,
Oscar Nierstrasz (Ed.),
Lecture Notes in Computer Science 707,
Springer-Verlag, Heidelberg
pp. 406–431

المنارة للاستشارات

Design Patterns: Abstraction and Reuse of Object-Oriented Design

Erich Gamma[1*], Richard Helm[2], Ralph Johnson[3], John Vlissides[2]

[1] Taligent, Inc.
10725 N. De Anza Blvd., Cupertino, CA 95014-2000 USA

[2] I.B.M. Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598 USA

[3] Department of Computer Science
University of Illinois at Urbana-Champaign
1034 W. Springfield Ave., Urbana, IL 61801 USA

Abstract. We propose **design patterns** as a new mechanism for expressing object-oriented design experience. Design patterns identify, name, and abstract common themes in object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. Design patterns play many roles in the object-oriented development process: they provide a common vocabulary for design, they reduce system complexity by naming and defining abstractions, they constitute a base of experience for building reusable software, and they act as building blocks from which more complex designs can be built. Design patterns can be considered reusable micro-architectures that contribute to an overall system architecture. We describe how to express and organize design patterns and introduce a catalog of design patterns. We also describe our experience in applying design patterns to the design of object-oriented systems.

1 Introduction

Design methods are supposed to promote good design, to teach new designers how to design well, and to standardize the way designs are developed. Typically, a design method comprises a set of syntactic notations (usually graphical) and a set of rules that govern how and when to use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate a design. Studies of expert programmers for conventional languages, however, have shown that knowledge is not organized simply around syntax, but in larger conceptual structures such as algorithms, data structures and idioms [1, 7, 9, 27], and plans that indicate steps necessary to fulfill a particular goal [26]. It is likely that designers do not think about the notation they are using for recording the design. Rather, they look for patterns to match against plans, algorithms, data structures, and idioms they have learned in the past. Good designers, it appears, rely

* Work performed while at UBILAB, Union Bank of Switzerland, Zurich, Switzerland.

المنارة للاستشارات

on large amounts of design experience, and this experience is just as important as the notations for recording designs and the rules for using those notations.

Our experience with the design of object-oriented systems and frameworks [15, 17, 22, 30, 31] bears out this observation. We have found that there exist idiomatic class and object structures that help make designs more flexible, reusable, and elegant. For example, the Model-View-Controller (MVC) paradigm from Smalltalk [19] is a design structure that separates representation from presentation. MVC promotes flexibility in the choice of views, independent of the model. Abstract factories [10] hide concrete subclasses from the applications that use them so that class names are not hard-wired into an application.

Well-defined design structures like these have a positive impact on software development. A software architect who is familiar with a good set of design structures can apply them immediately to design problems without having to rediscover them. Design structures also facilitate the reuse of successful architectures—expressing proven techniques as design structures makes them more readily accessible to developers of new systems. Design structures can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

To this end we propose **design patterns**, a new mechanism for expressing design structures. Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities. Design patterns have many uses in the object-oriented development process:

- Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. They reduce system complexity by naming and defining abstractions that are above classes and instances. A good set of design patterns effectively raises the level at which one programs.
- Design patterns constitute a reusable base of experience for building reusable software. They distill and provide a means to reuse the design knowledge gained by experienced practitioners. Design patterns act as building blocks for constructing more complex designs; they can be considered **micro-architectures** that contribute to overall system architecture.
- Design patterns help reduce the learning time for a class library. Once a library consumer has learned the design patterns in one library, he can reuse this experience when learning a new class library. Design patterns help a novice perform more like an expert.
- Design patterns provide a target for the reorganization or refactoring of class hierarchies [23]. Moreover, by using design patterns early in the lifecycle, one can avert refactoring at later stages of design.

The major contributions of this paper are: a definition of design patterns, a means to describe them, a system for their classification, and most importantly, a catalog containing patterns we have discovered while building our own class

المنارة للاستشارات

libraries and patterns we have collected from the literature. This work has its roots in Gamma's thesis [11], which abstracted design patterns from the ET++ framework. Since then the work has been refined and extended based on our collective experience. Our thinking has also been influenced and inspired by discussions within the Architecture Handbook Workshops at recent OOPSLA conferences [3, 4].

This paper has two parts. The first introduces design patterns and explains techniques to describe them. Next we present a classification system that characterizes common aspects of patterns. This classification will serve to structure the catalog of patterns presented in the second part of this paper. We discuss how design patterns impact object-oriented programming and design. We also review related work.

The second part of this paper (the Appendix) describes our current catalog of design patterns. As we cannot include the complete catalog in this paper (it currently runs over 90 pages [12]), we give instead a brief summary and include a few abridged patterns. Each pattern in this catalog is representative of what we judge to be good object-oriented design. We have tried to reduce the subjectivity of this judgment by including only design patterns that have seen practical application. Every design pattern we have included works—most have been used at least twice and have either been discovered independently or have been used in a variety of application domains.

2 Design Patterns

A design pattern consists of three essential parts:

1. An abstract description of a class or object collaboration and its structure. The description is abstract because it concerns abstract design, not a particular design.
2. The issue in system design addressed by the abstract structure. This determines the circumstances in which the design pattern is applicable.
3. The consequences of applying the abstract structure to a system's architecture. These determine if the pattern should be applied in view of other design constraints.

Design patterns are defined in terms of object-oriented concepts. They are sufficiently abstract to avoid specifying implementation details, thereby ensuring wide applicability, but a pattern may provide hints about potential implementation issues.

We can think of a design pattern as a micro-architecture. It is an architecture in that it serves as a blueprint that may have several realizations. It is "micro" in that it defines something less than a complete application or library. To be useful, a design pattern should be applicable to more than a few problem domains; thus design patterns tend to be relatively small in size and scope. A design pattern can also be considered a transformation of system structure. It defines the context

المنارة للاستشارات

for the transformation, the change to be made, and the consequences of this transformation.

To help readers understand patterns, each entry in the catalog also includes detailed descriptions and examples. We use a template (Figure 1) to structure our descriptions and to ensure uniformity between entries in the catalog. This template also explains the motivation behind its structure. The Appendix contains three design patterns that use the template. We urge readers to study the patterns in the Appendix as they are referenced in the following text.

3 Categorizing Design Patterns

Design patterns vary in their granularity and level of abstraction. They are numerous and have common properties. Because there are many design patterns, we need a way to organize them. This section introduces a classification system for design patterns. This classification makes it easy to refer to families of related patterns, to learn the patterns in the catalog, and to find new patterns.

		Characterization		
		Creational	**Structural**	**Behavioral**
Jurisdiction	**Class**	Factory Method	Adapter (class) Bridge (class)	Template Method
	Object	Abstract Factory Prototype Solitaire	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
	Compound	Builder	Composite Wrapper	Interpreter Iterator (compound) Walker

Table 1. Design Pattern Space

We can think of the set of all design patterns in terms of two orthogonal criteria, **jurisdiction** and **characterization**. Table 1 organizes our current set of patterns according to these criteria.

Jurisdiction is the domain over which a pattern applies. Patterns having **class** jurisdiction deal with relationships between base classes and their subclasses;

المنارة للاستشارات

Design Pattern Name **Jurisdiction Characterization**

What is the pattern's name and classification? The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Intent
What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation
A scenario in which the pattern is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This information will help the reader understand the more abstract description of the pattern that follows.

Applicability
What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

Participants
Describe the classes and/or objects participating in the design pattern and their responsibilities using CRC conventions [5].

Collaborations
Describe how the participants collaborate to carry out their responsibilities.

Diagram
A graphical representation of the pattern using a notation based on the Object Modeling Technique (OMT) [25], to which we have added method pseudo-code.

Consequences
How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation
What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

Examples
This section presents examples from real systems. We try to include at least two examples from different domains.

See Also
What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

Fig. 1. Basic Design Pattern Template

المنارة للاستشارات

class jurisdiction covers static semantics. The **object** jurisdiction concerns relationships between peer objects. Patterns having **compound** jurisdiction deal with recursive object structures. Some patterns capture concepts that span jurisdictions. For example, iteration applies both to collections of objects (i.e., object jurisdiction) and to recursive object structures (compound jurisdiction). Thus there are both object and compound versions of the Iterator pattern.

Characterization reflects what a pattern does. Patterns can be characterized as either **creational**, **structural**, or **behavioral**. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The following sections describe pattern jurisdictions in greater detail for each characterization using examples from our catalog.

3.1 Class Jurisdiction

Class Creational. Creational patterns abstract how objects are instantiated by hiding the specifics of the creation process. They are useful because it is often undesirable to specify a class name explicitly when instantiating an object. Doing so limits flexibility; it forces the programmer to commit to a particular class instead of a particular protocol. If one avoids hard-coding the class, then it becomes possible to defer class selection to run-time.

Creational class patterns in particular defer some part of object creation to subclasses. An example is the Factory Method, an abstract method that is called by a base class but defined in subclasses. The subclass methods create instances whose type depends on the subclass in which each method is implemented. In this way the base class does not hard-code the class name of the created object. Factory Methods are commonly used to instantiate members in base classes with objects created by subclasses.

For example, an abstract Application class needs to create application-specific documents that conform to the Document type. Application instantiates these Document objects by calling the factory method DoMakeDocument. This method is overridden in classes derived from Application. The subclass DrawApplication, say, overrides DoMakeDocument to return a DrawDocument object.

Class Structural. Structural class patterns use inheritance to compose protocols or code. As a simple example, consider using multiple inheritance to mix two or more classes into one. The result is an amalgam class that unites the semantics of the base classes. This trivial pattern is quite useful in making independently-developed class libraries work together [15].

Another example is the class-jurisdictional form of the Adapter pattern. In general, an Adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces. A class Adapter accomplishes this by inheriting privately from an adaptee class. The Adapter then expresses its interface in terms of the adaptee's.

المنارة للاستشارات

Class Behavioral. Behavioral class patterns capture how classes cooperate with their subclasses to fulfill their semantics. Template Method is a simple and well-known behavioral class pattern [32]. Template methods define algorithms step by step. Each step can invoke an abstract method (which the subclass must define) or a base method. The purpose of a template method is to provide an abstract definition of an algorithm. The subclass must implement specific behavior to provide the services required by the algorithm.

3.2 Object Jurisdiction

Object patterns all apply various forms of non-recursive object composition. Object composition represents the most powerful form of reusability—a collection of objects are most easily reused through variations on how they are composed rather than how they are subclassed.

Object Creational. Creational object patterns abstract how sets of objects are created. The Abstract Factory pattern (page 18) is a creational object pattern. It describes how to create "product" objects through an generic interface. Subclasses may manufacture specialized versions or compositions of objects as permitted by this interface. In turn, clients can use abstract factories to avoid making assumptions about what classes to instantiate. Factories can be composed to create larger factories whose structure can be modified at run-time to change the semantics of object creation. The factory may manufacture a custom composition of instances, a shared or one-of-a-kind instance, or anything else that can be computed at run-time, so long as it conforms to the abstract creation protocol.

For example, consider a user interface toolkit that provides two types of scroll bars, one for Motif and another for Open Look. An application programmer may not want to hard-code one or the other into the application—the choice of scroll bar will be determined by, say, an environment variable. The code that creates the scroll bar can be encapsulated in the class Kit, an abstract factory that abstracts the specific type of scroll bar to instantiate. Kit defines a protocol for creating scroll bars and other user interface elements. Subclasses of Kit redefine operations in the protocol to return specialized types of scroll bars. A MotifKit's scroll bar operation would instantiate and return a Motif scroll bar, while the corresponding OpenLookKit operation would return an Open Look scroll bar.

Object Structural. Structural object patterns describe ways to assemble objects to realize new functionality. The added flexibility inherent in object composition stems from the ability to change the composition at run-time, which is impossible with static class composition[4].

Proxy is an example of a structural object pattern. A proxy acts as a convenient surrogate or placeholder for another object. A proxy can be used as a

[4] However, object models that support dynamic inheritance, most notably Self [29], are as flexible as object composition in theory.

المنارة للاستشارات

local representative for an object in a different address space (remote proxy), to represent a large object that should be loaded on demand (virtual proxy), or to protect access to the original object (protected proxy). Proxies provide a level of indirection to particular properties of objects. Thus they can restrict, enhance, or alter an object's properties.

The Flyweight pattern is concerned with object sharing. Objects are shared for at least two reasons: efficiency and consistency. Applications that use large quantities of objects must pay careful attention to the cost of each object. Substantial savings can accrue by sharing objects instead of replicating them. However, objects can only be shared if they do not define context-dependent state. Flyweights have no context-dependent state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, flyweights may be shared freely. Moreover, it may be necessary to ensure that all copies of an object stay consistent when one of the copies changes. Sharing provides an automatic way to maintain this consistency.

Object Behavioral. Behavioral object patterns describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. For example, patterns such as Mediator and Chain of Responsibility abstract control flow. They call for objects that exist solely to redirect the flow of messages. The redirection may simply notify another object, or it may involve complex computation and buffering. The Observer pattern abstracts the synchronization of state or behavior. Entities that are co-dependent to the extent that their state must remain synchronized may exploit Observer. The classic example is the model-view pattern, in which multiple views of the model are notified whenever the model's state changes.

The Strategy pattern (page 21) objectifies an algorithm. For example, a text composition object may need to support different line breaking algorithms. It is infeasible to hard-wire all such algorithms into the text composition class and subclasses. An alternative is to objectify different algorithms and provide them as **Compositor** subclasses. The interface for Compositors is defined by the abstract Compositor class, and its derived classes provide different layout strategies, such as simple line breaks or full page justification. Instances of the Compositor subclasses can be coupled with the text composition at run-time to provide the appropriate text layout. Whenever a text composition has to find line breaks, it forwards this responsibility to its current Compositor object.

3.3 Compound Jurisdiction

In contrast to patterns having object jurisdiction, which concern peer objects, patterns with compound jurisdiction affect recursive object structures.

Compound Creational. Creational compound patterns are concerned with the creation of recursive object structures. An example is the Builder pattern. A Builder base class defines a generic interface for incrementally constructing

المنارة للاستشارات

recursive object structures. The Builder hides details of how objects in the structure are created, represented, and composed so that changing or adding a new representation only requires defining a new Builder class. Clients will be unaffected by changes to Builder.

Consider a parser for the RTF (Rich Text Format) document exchange format that should be able to perform multiple format conversions. The parser might convert RTF documents into (1) plain ASCII text and (2) a text object that can be edited in a text viewer object. The problem is how to make the parser independent of these different conversions.

The solution is to create an RTFReader class that takes a Builder object as an argument. The RTFReader knows how to parse the RTF format and notifies the Builder whenever it recognizes text or an RTF control word. The builder is responsible for creating the corresponding data structure. It separates the parsing algorithm from the creation of the structure that results from the parsing process. The parsing algorithm can then be reused to create any number of different data representations. For example, an ASCII builder ignores all notifications except plain text, while a Text builder uses the notifications to create a more complex text structure.

Compound Structural. Structural compound patterns capture techniques for structuring recursive object structures. A simple example is the Composite pattern. A Composite is a recursive composition of one or more other Composites. A Composite treats multiple, recursively composed objects as a single object.

The Wrapper pattern (page 24) describes how to flexibly attach additional properties and services to an object. Wrappers can be nested recursively and can therefore be used to compose more complex object structures. For example, a Wrapper containing a single user interface component can add decorations such as borders, shadows, scroll bars, or services like scrolling and zooming. To do this, the Wrapper must conform to the interface of its wrapped component and forward messages to it. The Wrapper can perform additional actions (such as drawing a border around the component) either before or after forwarding a message.

Compound Behavioral. Finally, behavioral compound patterns deal with behavior in recursive object structures. Iteration over a recursive structure is a common activity captured by the Iterator pattern. Rather than encoding and distributing the traversal strategy in each class in the structure, it can be extracted and implemented in an Iterator class. Iterators objectify traversal algorithms over recursive structures. Different iterators can implement pre-order, in-order, or post-order traversals. All that is required is that nodes in the structure provide services to enumerate their sub-structures. This avoids hard-wiring traversal algorithms throughout the classes of objects in a composite structure. Iterators may be replaced at run-time to provide alternative traversals.

المنارة للاستشارات

4 Experience with Design Patterns

We have applied design patterns to the design and construction of a several systems. We briefly describe two of these systems and our experience.

4.1 ET++SwapsManager

The ET++SwapsManager [10] is a highly interactive tool that lets traders value, price, and perform what-if analyses for a financial instrument called a **swap**. During this project the developers had to first learn the ET++ class library, then implement the tool, and finally design a framework for creating "calculation engines" for different financial instruments. While teaching ET++ we emphasized not only learning the class library but also describing the applied design patterns. We noticed that design patterns reduced the effort required to learn ET++. Patterns also proved helpful during development in design and code reviews. Patterns provided a common vocabulary to discuss a design. Whenever we encountered problems in the design, patterns helped us explore design alternatives and find solutions.

4.2 QOCA: A Constraint Solving Toolkit

QOCA (Quadratic Optimization Constraint Architecture) [14, 15] is a new object-oriented constraint-solving toolkit developed at IBM Research. QOCA leverages recent results in symbolic computation and geometry to support efficient incremental and interactive constraint manipulation. QOCA's architecture is designed to be flexible. It permits experimentation with different classes of constraints and domains (e.g., reals, booleans, etc.), different constraint-solving algorithms for these domains, and different representations (doubles, infinite precision) for objects in these domains. QOCA's object-oriented design allows parts of the system to be varied independently of others. This flexibility was achieved, for example, by using Strategy patterns to factor out constraint solving algorithms and Bridges to factor out domains and representations of variables. In addition, the Observable pattern is used to propagate notifications when variables change their values.

4.3 Summary of Observations

The following points summarize the major observations we have made while applying design patterns:

- Design patterns motivate developers to go beyond concrete objects; that is, they objectify concepts that are not immediately apparent as objects in the problem domain.
- Choosing intuitive class names is important but also difficult. We have found that design patterns can help name classes. In the ET++SwapsManager's calculation engine framework we encoded the name of the design pattern

المنارة للاستشارات

in the class name (for example CalculationStrategy or TableAdaptor). This convention results in longer class names, but it gives clients of these classes a hint about their purpose.

- We often apply design patterns *after* the first implementation of an architecture to improve its design. For example, it is easier to apply the Strategy pattern after the initial implementation to create objects for more abstract notions like a calculation engine or constraint solver. Patterns were also used as targets for class refactorings. We often find ourselves saying, "Make this part of a class into a Strategy," or, "Let's split the implementation portion of this class into a Bridge."
- Presenting design patterns together with examples of their application turned out to be an effective way to teach object-oriented design by example.
- An important issue with any reuse technology is how a reusable component can be adapted to create a problem-specific component. Design patterns are particularly suited to reuse because they are abstract. Though a concrete class structure may not be reusable, the design pattern underlying it often is.
- Design patterns also reduce the effort required to learn a class library. Each class library has a certain design "culture" characterized by the set of patterns used implicitly by its developers. A specific design pattern is typically reused in different places in the library. A client should therefore learn these patterns as a first step in learning the library. Once they are familiar with the patterns, they can reuse this understanding. Moreover, because some patterns appear in other class libraries, it is possible to reuse the knowledge about patterns when learning other libraries as well.

5 Related Work

Design patterns are an approach to software reuse. Krueger [20] introduces the following taxonomy to characterize different reuse approaches: software component reuse, software schemas, application generators, transformation systems, and software architectures. Design patterns are related to both software schemas and reusable software architectures. Software schemas emphasize reusing abstract algorithms and data structures. These abstractions are represented formally so they can be instantiated automatically. The Paris system [18] is representative of schema technology. Design patterns are higher-level than schemas; they focus on design structures at the level of collaborating classes and not at the algorithmic level. In addition, design patterns are not formal descriptions and cannot be instantiated directly. We therefore prefer to view design patterns as reusable software architectures. However, the examples Krueger lists in this category (blackboard architectures for expert systems, adaptable database subsystems) are all coarse-grained architectures. Design patterns are finer-grained and therefore can be characterized as reusable micro-architectures.

Most research into patterns in the software engineering community has been geared towards building knowledge-based assistants for automating the appli-

المنارة للاستشارات

cation of patterns for synthesis (that is, to write programs) and analysis (in debugging, for example) [13, 24]. The major difference between our work and that of the knowledge-based assistant community is that design patterns encode higher-level expertise. Their work has tended to focus on patterns like enumeration and selection, which can be expressed directly as reusable components in most existing object-oriented languages. We believe that characterizing and cataloging higher-level patterns that designers already use informally has an immediate benefit in teaching and communicating designs.

A common approach for reusing object-oriented software architectures are object-oriented frameworks [32]. A framework is a codified architecture for a problem domain that can be adapted to solve specific problems. A framework makes it possible to reuse an architecture together with a partial concrete implementation. In contrast to frameworks, design patterns allow only the reuse of abstract micro-architectures without a concrete implementation. However, design patterns can help define and develop frameworks. Mature frameworks usually reuse several design patterns. An important distinction between frameworks and design patterns is that frameworks are implemented in a programming language. Our patterns are ways of *using* a programming language. In this sense frameworks are more concrete than design patterns.

Design patterns are also related to the idioms introduced by Coplien [7]. These idioms are concrete design solutions in the context of C++. Coplien "focuses on idioms that make C++ programs more expressive." In contrast, design patterns are more abstract and higher-level than idioms. Patterns try to abstract design rather than programming techniques. Moreover, design patterns are usually independent of the implementation language.

There has been interest recently within the object-oriented community [8] in pattern languages for the architecture of buildings and communities as advocated by Christopher Alexander in *The Timeless Way of Building* [2]. Alexander's patterns consist of three parts:

- A context that describes when a pattern is applicable.
- The problem (or "system of conflicting forces") that the pattern resolves in that context.
- A configuration that describes physical relationships that solve the problem.

Both design patterns and Alexander's patterns share the notion of context/problem/configuration, but our patterns currently do not form a complete system of patterns and so do not strictly define a pattern language. This may be because object-oriented design is still a young technology—we may not have had enough experience in what constitutes good design to extract design patterns that cover all phases of the design process. Or this may be simply because the problems encountered in software design are different from those found in architecture and are not amenable to solution by pattern languages.

Recently, Johnson has advocated pattern languages to describe how to use use object-oriented frameworks [16]. Johnson uses a pattern language to explain how to extend and customize the Hotdraw drawing editor framework. However,

المنارة للاستشارات

these patterns are not design patterns; they are more descriptions of how to reuse existing components and frameworks instead of rules for generating new designs.

Coad's recent paper on object-oriented patterns [6] is also motivated by Alexander's work but is more closely related to our work. The paper has seven patterns: "Broadcast" is the same as Observer, but the other patterns are different from ours. In general, Coad's patterns seem to be more closely related to analysis than design. Design patterns like Wrapper and Flyweight are unlikely to be generated naturally during analysis unless the analyst knows these patterns well and thinks in terms of them. Coad's patterns could naturally arise from a simple attempt to model a problem. In fact, it is hard to see how any large model could avoid using patterns like "State Across a Collection" (which explains how to use aggregation) or "Behavior Across a Collection" (which describes how to distribute responsibility among objects in an aggregate). The patterns in our catalog are typical of a mature object-oriented design, one that has departed from the original analysis model in an attempt to make a system of reusable objects. In practice, both types of patterns are probably useful.

6 Conclusion

Design patterns have revolutionized the way we think about, design, and teach object-oriented systems. We have found them applicable in many stages of the design process—initial design, reuse, refactoring. They have given us a new level of abstraction for system design.

New levels of abstraction often afford opportunities for increased automation. We are investigating how interactive tools can take advantage of design patterns. One of these tools lets a user explore the space of objects in a running program and watch their interaction. Through observation the user may discover existing or entirely new patterns; the tool lets the user record and catalog his observations. The user may thus gain a better understanding of the application, the libraries on which it is based, and design in general.

Design patterns may have an even more profound impact on how object-oriented systems are designed than we have discussed. Common to most patterns is that they permit certain aspects of a system to be varied independently. This leads to thinking about design in terms of "What aspect of a design should be variable?" Answers to this question lead to certain applicable design patterns, and their application leads subsequently to modification of a design. We refer to this design activity as **variation-oriented design** and discuss it more fully in the catalog of patterns [12].

But some caveats are in order. Design patterns should not be applied indiscriminately. They typically achieve flexibility and variability by introducing additional levels of indirection and can therefore complicate a design. A design pattern should only be applied when the flexibility it affords is actually needed. The consequences described in a pattern help determine this. Moreover, one is

المنارة للاستشارات

often tempted to brand any new programming trick a new design pattern. A true design pattern will be non-trivial and will have had more than one application.

We hope that the design patterns described in this paper and in the companion catalog will provide the object-oriented community both a common design terminology and a repertoire of reusable designs. Moreover, we hope the catalog will motivate others to describe their systems in terms of design patterns and develop their own design patterns for others to reuse.

7 Acknowledgements

The authors wish to thank Doug Lea and Kent Beck for detailed comments and discussions about this work, and Bruce Anderson and the participants of the Architecture Handbook workshops at OOPSLA '91 and '92.

References

1. B. Adelson and Soloway E. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, 1985.
2. Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.
3. Association for Computing Machinery. *Addendum to the Proceedings, Object-Oriented Programming Systems, Languages, and Applications Conference*, Phoenix, AZ, October 1991.
4. Association for Computing Machinery. *Addendum to the Proceedings, Object-Oriented Programming Systems, Languages, and Applications Conference*, Vancouver, British Columbia, October 1992.
5. Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 1–6, New Orleans, LA, October 1989.
6. Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, September 1992.
7. James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Massechusetts, 1992.
8. Ward Cunningham and Kent Beck. Constructing abstractions for object-oriented applications. Technical Report CR-87-25, Computer Research Laboratory, Tektronix, Inc., 1987.
9. Bill Curtis. Cognitive issues in reusing software artifacts. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II*, pages 269–287. Addison-Wesley, 1989.
10. Thomas Eggenschwiler and Erich Gamma. The ET++SwapsManager: Using object technology in the financial engineering domain. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 166–178, Vancouver, British Columbia, October 1992.
11. Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*. Springer-Verlag, Berlin, 1992.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. A catalog of object-oriented design patterns. Technical Report in preparation, IBM Research Division, 1992.

المنارة للاستشارات

13. Mehdi T. Harandi and Frank H. Young. Software design using reusable algorithm abstraction. In *In Proc. 2nd IEEE/BCS Conf. on Software Engineering*, pages 94–97, 1985.
14. Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for user interfaces. In *Graphics Interface*, pages 301–309, Vancouver, British Columbia, 1992.
15. Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, pages 1–22, Champéry, Switzerland, October 1992. Also available as IBM Research Division Technical Report RC 18524 (79392).
16. Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 63–76, Vancouver, BC, October 1992.
17. Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, *Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, 1992. Springer-Verlag.
18. S. Katz, C.A. Richter, and K.-S. The. Paris: A system for reusing partially interpreted schemas. In *Proc. of the Ninth International Conference on Software Engineering*, 1987.
19. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.
20. Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), June 1992.
21. Mark A. Linton. Encapsulating a C++ library. In *Proceedings of the 1992 USENIX C++ Conference*, pages 57–66, Portland, OR, August 1992.
22. Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.
23. William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA Conference Proceedings*, pages 145–161, Marist College, Poughkeepsie, NY, September 1990.
24. Charles Rich and Richard C. Waters. Formalizing reusable software components in the programmer's apprentice. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II*, pages 313–343. Addison-Wesley, 1989.
25. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
26. Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5), September 1984.
27. James C. Spohrer and Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, July 1992.
28. ParcPlace Systems. *ParcPlace Systems, Objectworks/Smalltalk Release 4 Users Guide*. Mountain View, California, 1990.
29. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 227–242, Orlando, Florida, October 1987.

المنارة للاستشارات

30. John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.
31. André Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 46–57, San Diego, CA, September 1988.
32. Rebecca Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

المنارة للاستشارات

A Catalog Overview

The following summarizes the patterns in our current catalog.

Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Adapter makes the protocol of one class conform to the protocol of another.

Bridge separates an abstraction from its implementation. The abstraction may vary its implementations transparently and dynamically.

Builder provides a generic interface for incrementally constructing aggregate objects. A Builder hides details of how objects in the aggregate are created, represented, and composed.

Command objectifies the request for a service. It decouples the creator of the request for a service from the executor of that service.

Composite treats multiple, recursively-composed objects as a single object.

Chain of Responsibility defines a hierarchy of objects, typically arranged from more specific to more general, having responsibility for handling a request.

Factory Method lets base classes create instances of subclass-dependent objects.

Flyweight defines how objects can be shared. Flyweights support object abstraction at the finest granularity.

Glue defines a single point of access to objects in a subsystem. It provides a higher level of encapsulation for objects in the subsystem.

Interpreter defines how to represent the grammar, abstract syntax tree, and interpreter for simple languages.

Iterator objectifies traversal algorithms over object structures.

Mediator decouples and manages the collaboration between objects.

Memento opaquely encapsulates a snapshot of the internal state of an object and is used to restore the object to its original state.

Observer enforces synchronization, coordination, and consistency constraints between objects.

Prototype creates new objects by cloning a prototypical instance. Prototypes permit clients to install and configure dynamically the instances of particular classes they need to instantiate.

Proxy acts as a convenient surrogate or placeholder for another object. Proxies can restrict, enhance, or alter an object's properties.

Solitaire defines a one-of-a-kind object that provides access to unique or well-known services and variables.

State lets an object change its behavior when its internal state changes, effectively changing its class.

Strategy objectifies an algorithm or behavior.

Template Method implements an abstract algorithm, deferring specific steps to subclass methods.

Walker centralizes operations on object structures in one class so that these operations can be changed independently of the classes defining the structure.

Wrapper attaches additional services, properties, or behavior to objects. Wrappers can be nested recursively to attach multiple properties to objects.

المنارة للاستشارات

ABSTRACT FACTORY

Object Creational

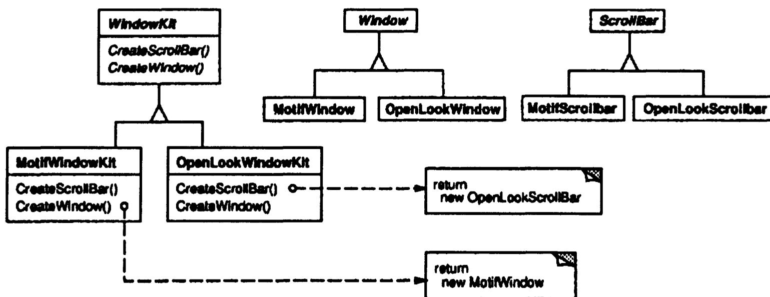
Intent

Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Motivation

Consider a user interface toolkit that supports multiple standard look-and-feels, say, Motif and Open Look, and provides different scroll bars for each. It is undesirable to hard-code dependencies on either standard into the application—the choice of look-and-feel and hence scroll bar may be deferred until run-time. Specifying the class of scroll bar limits flexibility and reusability by forcing a commitment to a particular class instead of a particular protocol. An Abstract Factory avoids this commitment.

An abstract base class WindowKit declares services for creating scroll bars and other controls. Controls for Motif and Open Look are derived from common abstract classes. For each look-and-feel there is a concrete subclass of WindowKit that defines services to create the appropriate control. For example, the CreateScrollBar() operation on the MotifKit would instantiate and return a Motif scroll bar, while the corresponding operation on the OpenLookKit returns an Open Look scroll bar. Clients access a specific kit through the interface declared by the WindowKit class, and they access the controls created by a kit only by their generic interface.



Applicability

When the classes of the product objects are variable, and dependencies on these classes must be removed from a client application.

When variations on the creation, composition, or representation of aggregate objects or subsystems must be removed from a client application. Differences in configuration can be obtained by using different concrete factories. Clients do not explicitly create and configure the aggregate or subsystem but defer this responsibility to an AbstractFactory class. Clients instead call a method of the AbstractFactory that returns an object providing access to the aggregate or subsystem.

المنارة للاستشارات

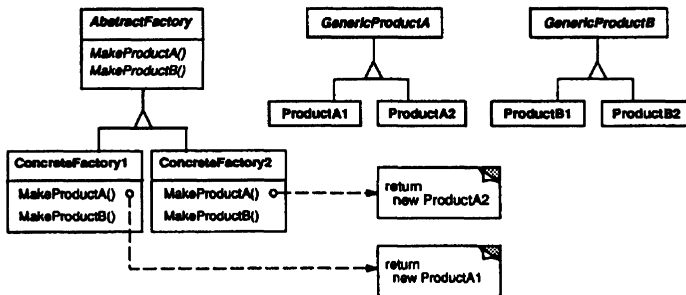
Participants

- **AbstractFactory**
  - declares a generic interface for operations that create generic product objects.
- **ConcreteFactory**
  - defines the operations that create specific product objects.
- **GenericProduct**
  - declares a generic interface for product objects.
- **SpecificProduct**
  - defines a product object created by the corresponding concrete factory.
  - all product classes must conform to the generic product interface.

Collaborations

- Usually a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To use different product objects, clients must be configured to use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclasses.

Diagram



Consequences

Abstract Factory provides a focus during development for changing and controlling the types of objects created by clients. Because a factory objectifies the responsibility for and the process of creating product objects, it isolates clients from implementation classes. Only generic interfaces are visible to clients. Implementation class names do not appear in client code. Clients can be defined and implemented solely in terms of protocols instead of classes.

Abstract factories that encode class names in operation signatures can be difficult to extend with new kinds of product objects. This can require redeclaring the AbstractFactory and all ConcreteFactories. Abstract factories can be composed with subordinate factory objects. Responsibility for creating objects is delegated

المنارة للاستشارات

to these sub-factories. Composition of abstract factories provides a simple way to extend the kinds of objects a factory is responsible for creating.

Examples

InterViews uses the "Kit" suffix [21] to denote abstract factory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending on the layout desired.

ET++ [31] employs the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects representing window system resources (for example, MakeWindow, MakeFont, MakeColor). Concrete subclasses implement the interfaces for a specific window system. At runtime ET++ creates an instance of a concrete WindowSystem subclass that creates system resource objects.

Implementation

A novel implementation is possible in Smalltalk. Because classes are first-class objects, it is not necessary to have distinct ConcreteFactory subclasses to create the variations in products. Instead, it is possible to store classes that create these products in variables inside a concrete factory. These classes create new instances on behalf of the concrete factory. This technique permits variation in product objects at finer levels of granularity than by using distinct concrete factories. Only the classes kept in variables need to be changed.

See Also

Factory Method: Abstract Factories are often implemented using Factory Methods.

المنارة للاستشارات

STRATEGY — Object Behavioral

Intent

A Strategy objectifies an algorithm or behavior, allowing the algorithm or behavior to be varied independently of its clients.

Motivation

There are many algorithms for breaking a text stream into lines. It is impossible to hard-wire all such algorithms into the classes that require them. Different algorithms might be appropriate at different times.

One way to address this problem is by defining separate classes that encapsulate the different linebreaking algorithms. An algorithm objectified in this way is called a Strategy. InterViews [22] and ET++ [31] use this approach.

Suppose a Composition class is responsible for maintaining and updating the line breaks of text displayed in a text viewer. Linebreaking strategies are not implemented by the class Composition. Instead, they are implemented separately by subclasses of the Compositor class. Compositor subclasses implement different strategies as follows:

- **SimpleCompositor** implements a simple strategy that determines line breaks one at a time.
- **TeXCompositor** implements the TEXalgorithm for finding line breaks. This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
- **ArrayCompositor** implements a strategy that is used not for text but for breaking a collection of icons into rows. It selects breaks so that each row has a fixed number of items.

A Composition maintains a reference to a Compositor object. Whenever a Composition is required to find line breaks, it forwards this responsibility to its current Compositor object. The client of Composition specifies which Compositor should be used by installing the corresponding Compositor into the Composition (see the diagram below).

Applicability

Whenever an algorithm or behavior should be selectable and replaceable at run-time, or when there exist variations in the implementation of the algorithm, reflecting different space-time tradeoffs, for example.

Use a Strategy whenever many related classes differ only in their behavior. Strategies provide a way to configure a single class with one of many behaviors.

المنارة للاستشارات

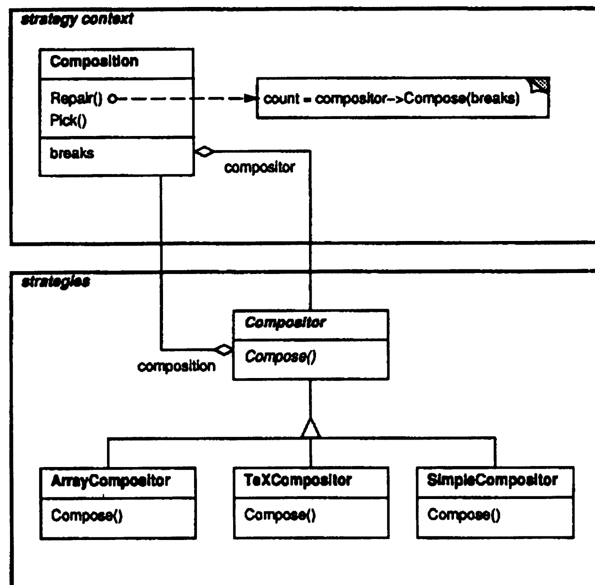
Participants

- **Strategy**
  - objectifies and encapsulates an algorithm or behavior.
- **StrategyContext**
  - maintains a reference to a Strategy object.
  - maintains the state manipulated by the Strategy.
  - can be configured by passing it an appropriate Strategy object.

Collaborations

- Strategy manipulates the StrategyContext. The StrategyContext normally passes itself as an argument to the Strategy's methods. This allows the Strategy to call back the StrategyContext as required.
- StrategyContext forwards requests from its clients to the Strategy. Usually clients pass Strategy objects to the StrategyContext. Thereafter clients only interact with the StrategyContext. There is often a family of Strategy classes from which a client can choose.

Diagram



Consequences

Strategies can define a family of policies that a StrategyContext can reuse. Separating a Strategy from its context increases reusability, because the Strategy may vary independently from the StrategyContext.

Variations on an algorithm can also be implemented with inheritance, that is, with an abstract class and subclasses that implement different behaviors. However, this hard-wires the implementation into a specific class; it is not possible to change

المنارة للاستشارات

behaviors dynamically. This results in many related classes that differ only in some behavior. It is often better to break out the variations of behavior into their own classes. The Strategy pattern thus increases modularity by localizing complex behavior. The typical alternative is to scatter conditional statements throughout the code that select the behavior to be performed.

Implementation

The interface of a Strategy and the common functionality among Strategies is often factored out in an abstract class. Strategies should avoid maintaining state across invocations so that they can be used repeatedly and in multiple contexts.

Examples

In the RTL System for compiler code optimization [17], Strategies define different register allocation schemes (RegisterAllocator) and different instruction set scheduling policies (RISCscheduler, CISCscheduler). This gives flexibility in targeting the optimizer for different machine architectures.

The ET++SwapsManager calculation engine framework [10] computes prices for different financial instruments. Its key abstractions are Instrument and YieldCurve. Different instruments are implemented as subclasses of Instrument. The YieldCurve calculates discount factors to present value future cash flows. Both of these classes delegate some behavior to Strategy objects. The framework provides a family of Strategy classes that define algorithms to generate cash flows, to value swaps, and to calculate discount factors. New calculation engines are created by parameterizing Instrument and YieldCurve with appropriate Strategy objects. This approach supports mixing and matching existing Strategy implementations while permitting the definition of new Strategy objects.

See Also

Walker often implements algorithms over recursive object structures. Walkers can be considered compound strategies.

المنارة للاستشارات

Intent

A Wrapper attaches additional services, properties, or behavior to objects. Wrappers can be nested recursively to attach multiple properties to objects.

Motivation

Sometimes it is desirable to attach properties to individual objects instead of classes. In a graphical user interface toolkit, for example, properties such as borders or services like scrolling should be freely attachable to any user interface component.

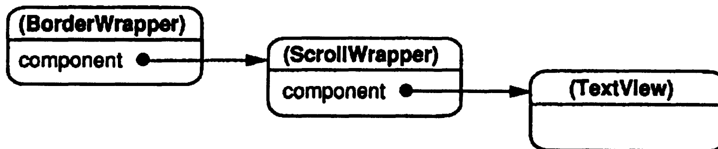
One way to attach properties to components is via inheritance. Inheriting a border from a base class will give all instances of its derived classes a border. This is inflexible because the choice of border is made statically. It is more flexible to let a client decide how and when to decorate the component with a border.

This can be achieved by enclosing the component in another object that adds the border. The enclosing object, which must be transparent to clients of the component, is called a Wrapper. This transparency is the key for nesting Wrappers recursively to construct more complex user interface components. A Wrapper forwards requests to its enclosed user interface component. The Wrapper may perform additional actions before or after forwarding the request, such as drawing a border around a user interface component.

Typical properties or services provided by user interface Wrappers are:

- decorations like borders, shadows, or scroll bars; or
- services like scrolling or zooming.

The following diagram illustrates the composition of a TextView with a BorderWrapper and a ScrollWrapper to produce a bordered, scrollable TextView.



Applicability

When properties or behaviors should be attachable to individual objects dynamically and transparently.

When there is a need to extend classes in an inheritance hierarchy. Rather than modifying their base class, instances are enclosed in a Wrapper that adds the additional behavior and properties. Wrappers thus provide an alternative to extending the base class without requiring its modification. This is of particular concern when the base class comes from a class library that cannot be modified.

المنارة للاستشارات

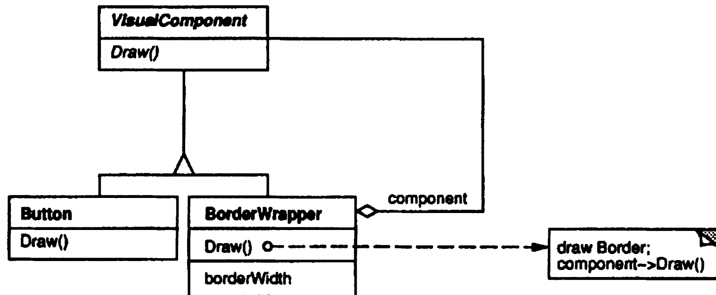
Participants

- **Component**
  - the object to which additional properties or behaviors are attached.
- **Wrapper**
  - encapsulates and enhances its Component. It defines an interface that conforms to its Component's.
  - Wrapper maintains a reference to its Component.

Collaborations

- Wrapper forwards requests to its Component. It may optionally perform additional operations before and after forwarding the request.

Diagram



Consequences

Using Wrappers to add properties is more flexible than using inheritance. With Wrappers, properties can be attached and detached at run-time simply by changing the Wrapper. Inheritance would require creating a new class for each property composition (for example, BorderdScrollableTextView, BorderedTextView). This clutters the name space of classes unnecessarily and should be avoided. Moreover, providing different Wrapper classes for a specific Component class allows mixing and matching behaviors and properties.

Examples

Most object-oriented user interface toolkits use Wrappers to add graphical embellishments to widgets. Examples include InterViews [22], ET++ [31], and the ParcPlace Smalltalk class library [28]. More exotic applications of Wrappers are the DebuggingGlyph from InterViews and the PassivityWrapper from ParcPlace Smalltalk. A DebuggingGlyph prints out debugging information before and after it forwards a layout request to its enclosed object. This trace information can be used to analyse and debug the layout behavior of objects in a complex object composition. The PassivityWrapper can enable or disable user interactions with the enclosed object.

Implementation

Implementation of a set of Wrapper classes is simplified by an abstract base class, which forwards all requests to its component. Derived classes can then override only those operations for which they want to add behavior. The abstract base class ensures that all other requests are passed automatically to the Component.

المنارة للاستشارات

See Also

Adapter: A Wrapper is different from an Adapter, because a Wrapper only changes an object's properties and not its interface; an Adapter will give an object a completely new interface.

Composite: A Wrapper can be considered a degenerate Composite with only one component. However, a Wrapper adds additional services—it is not intented for object aggregation.

المنارة للاستشارات

John Guttag

Abstract Data Types and the Development of Data Structures

Communications of the ACM, Vol. 20 (6), 1977
pp. 396–404

المنـــارة للاستشارات

B. Wegbreit
Editor

Abstract Data Types and the Development of Data Structures

John Guttag
University of Southern California

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work was supported in part by the National Science Foundation under grant number MCS76-06089.

A version of this paper was presented as the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, Salt Lake City, Utah, March 22–24, 1976.

Author's address: Computer Science Department, University of Southern California, Los Angeles, CA 90007.

المنارة للاستشارات

Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. This paper presents and discusses the application of an algebraic technique for the specification of abstract data types. Among the examples presented is a top-down development of a symbol table for a block structured language; a discussion of the proof of its correctness is given. The paper also contains a brief discussion of the problems involved in constructing algebraic specifications that are both consistent and complete.

Key Words and Phrases: abstract data type, correctness proof, data type, data structure, specification, software specification

CR Categories: 4.34, 5.24

1. Introduction

Dijkstra [4] and many others have made the point that the amount of complexity that the human mind can cope with at any instant in time is considerably less than that embodied in much of the software that one might wish to build. Thus the key problem in the design and implementation of large software systems is reducing the amount of complexity or detail that must be considered at any one time. One way to do this is via the process of abstraction.

One of the most significant aids to abstraction used in programming is the self-contained subroutine. At the point where one decides to invoke a subroutine, one can (and most often should) treat it as a "black box." It performs a specific arbitrarily abstract function by means of an unprescribed algorithm. Thus, at the level

المنارة للاستشارات

where it is invoked, it separates the relevant detail of "what" from the irrelevant detail of "how." Similarly, at the level where it is implemented, it is usually unnecessary to complicate the "how" by considering the "why," i.e. the exact reasons for invoking a subroutine often need not be of concern to its implementor. By nesting subroutines, one may develop a hierarchy of abstractions.

Unfortunately, the nature of the abstractions that may be conveniently achieved through the use of subroutines is limited. Subroutines, while well suited to the description of abstract events (operations), are not particularly well suited to the description of abstract objects. This is a serious drawback, for in a great many applications the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem.

2. The Abstraction of Data

The large knot of complexly interrelated attributes associated with a data object may be separated according to the nature of the information that the attributes convey regarding the data objects that they qualify. Two kinds of attributes, each of which may be studied in isolation, are:

(1) those that describe the representation of objects and the implementations of the operations associated with them in terms of other objects and operations, e.g. in terms of a physical store and a processor's order code;

(2) those that specify the names and define the abstract meanings of the operations associated with

المنارة للاستشارات

an object. Though these two kinds of attributes are in practice highly interdependent, they represent logically independent concepts.

The emphasis in this paper is on the second kind of attribute, i.e. on the specification of the operations associated with classes of data objects. At most points in a program one is concerned solely with the behavioral characteristics of a data object. One is interested in what one can do with it, not in how the various operations on it are implemented. The analogy with a closed procedure is exact. More often than not, one need be no more concerned with the underlying representation of the object being operated on than one is with the algorithm used to implement an invoked procedure.

If at a given level of refinement one is interested only in the behavioral characteristics of certain data objects, then any attempt to abstract data must be based upon those characteristics, and only those characteristics. The introduction of other attributes, e.g. a representation, can only serve to cloud the relevant issues. We use the term "abstract data type" to refer to a class of objects defined by a representation-independent specification.

The class construct of SIMULA 67 [3] has been used as the starting point for much of the more recent work on embedding abstract types in programming languages, e.g. [14, 16, 18]. While each of these offers a mechanism for binding together the operations and storage structures representing a type, they offer no representation-independent means for specifying the behavior of the operations. The only representation-independent information that one can supply are the domains and ranges of the various operations. One

للاستشارات

could, for example, define a type Queue (of Items) with the operations

NEW:	→ Queue
ADD:	Queue × Item → Queue
FRONT:	Queue → Item
REMOVE:	Queue → Queue
IS_EMPTY?:	Queue → Boolean

Unfortunately, however, short of supplying a representation, the only mechanism for denoting what these operations "mean" is a judicious choice of names. Except for intuitions about the meaning of such words as Queue and FRONT, the operations might just as easily be defining type Stack as type Queue. The domain and range specifications for these two types are isomorphic. To rely on one's intuition about the meaning of names can be dangerous even when dealing with familiar types [19]. When dealing with unfamiliar types it is almost impossible. What is needed, therefore, is a mechanism for specifying the semantics of the operations of the type.

There are, of course, many possible approaches to the specification of the semantics of an abstract data type. Most, however, can be placed in one of two categories: operational or definitional. In an operational specification, instead of trying to describe the properties of the abstract data type, one gives a recipe for constructing it. One begins with some well-understood language or discipline and builds a model for the type in terms of that discipline. Wulf [24], for example, makes good use of sequences in modeling various data structures.

The operational approach to formal specification has many advantages. Most significantly, operational

المنارة للاستشارات

specifications seem to be relatively (compared to definitional specifications) easily constructed by those trained as programmers—chiefly because the construction of operational specifications so closely resembles programming. As the operations to be specified grow complex, however, operational specifications tend to get too long (see, for example, Batey [1]) to permit substantial confidence in their aptness. As the number of operations grows, problems arise because the relations among the operations are not explicitly stated, and inferring them becomes combinatorially harder.

The most serious problem associated with operational specifications is that they almost always force one to overspecify the abstraction. By introducing extraneous detail, they associate nonessential attributes with the type. This extraneous detail complicates the problem of proving the correctness of an implementation by introducing conditions that are irrelevant, yet nevertheless must be verified. More importantly, the introduction of extraneous detail places unnecessary constraints on the choice of an implementation and may potentially eliminate the best solutions to the problem.

Axiomatic definitions avoid this problem. The algebraic approach used here owes much to the work of Hoare [13] (which in turn owes much to Floyd [5]) and is closely related to Standish's "axiomatic specifications" [22] and Zilles' "algebraic specifications" [25]. Its formal basis stems from the heterogeneous algebras of Birkhoff and Lipson [2]. An algebraic specification of an abstract type consists of two pairs: a syntactic specification and a set of relations. The syntactic specification provides the syntactic information that many programming languages already require: the names, domains, and ranges of the operations associated with

المنارة للاستشارات

domains, and ranges of the operations associated with the type. The set of relations defines the meanings of the operations by stating their relationships to one another.

3. A Short Example

Consider type Queue (of Items) with the operations listed in the previous section. The syntactic specification is as above:

NEW:	→ Queue
ADD:	Queue × Item → Queue
FRONT:	Queue → Item
REMOVE:	Queue → Queue
IS_EMPTY?:	Queue → Boolean

The distinguishing characteristic of a queue is that it is a first in–first out storage device. A good axiomatic definition of the above operations must therefore assert that and only that characteristic. The relations (or axioms) below comprise just such a definition. The meanings of the axioms should be relatively clear. ("=" has its standard meaning, "q" and "i" are typed free variables, and "error" is a distinguished value with the property that the value of any operation applied to an argument list containing error is error, e.g. $f_n(x_1, \ldots, x_i, \text{error}, x_{i+2}, \ldots, x_n) = \text{error}$.)

(1) IS_EMPTY? (NEW) = true
(2) IS_EMPTY? (ADD(q,i)) = false
(3) FRONT(NEW) = error
(4) FRONT (ADD(q,i)) = **if** IS_EMPTY? (q)
then i
else FRONT(q)
(5) REMOVE(NEW) = error

المنارة للاستشارات

(6) REMOVE (ADD(q,i)) = **if** IS_EMPTY? (q)
then NEW
else ADD(REMOVE(q),i)

Note that this set of axioms involves no assumption about the attributes of type Item. In effect Item is a parameter of type Type, and the specification may be viewed as defining a type schema rather than a single type. This will be the case for many algebraic type specifications.

With some practice, one can become quite adept at reading algebraic axiomatizations. Practice also makes it easier to construct such specifications; see Guttag [11]. Unfortunately, it does not make it trivial. It is not always immediately clear how to attack the problem. Nor, once one has constructed an axiomatization, is it always easy to ascertain whether or not the axiomatization is consistent and sufficiently complete. The meaning of the operations is supplied by a set of individual statements of fact. If any two of these are contradictory, the axiomatization is inconsistent. If the combination of statements is not sufficient to convey all of the vital information regarding the meaning of the operations of the type, the axiomatization is not sufficiently complete.[1]

Experience indicates that completeness is, in a practical sense, a more severe problem than consistency. If one has an intuitive understanding of the type being specified, one is unlikely to supply contradictory axioms. It is, on the other hand, extremely easy to overlook one or more cases. Boundary conditions, e.g.

---

[1] Sufficiently complete is a technical notion first developed in Guttag [8]. It differs considerably from both the notion of completeness commonly used in logic and that used in Zilles [25].

المنارة للاستشارات

REMOVE(NEW), are particularly likely to be overlooked.

In an attempt to ameliorate this problem, we have devised heuristics to aid the user in the initial presentation of an axiomatic specification of the operations of an abstract type and a system to mechanically "verify" the sufficient-completeness of that specification. As the first step in defining a new type, the user would supply the system with the syntactic specification of the type and an axiomatization constructed with the aid of the heuristics mentioned above. Given this preliminary specification, the system would begin to prompt the user to supply the additional information necessary for the system to derive a sufficiently complete axiom set for the operations. A detailed look at sufficient-completeness is contained in Guttag [8, 9].

4. An Extended Example

A common data structuring problem is the design of the symbol table component of a compiler for a block structured language. Many sources contain good discussions of various symbol table organizations. Setting aside variations in form, the basic operations described vary little from source to source. They are:

INIT:	Allocate and initialize the symbol table.
ENTERBLOCK:	Prepare a new local naming scope.
LEAVEBLOCK:	Discard entries from the most recent scope entered, and reestablish the next outer scope.
IS_INBLOCK?:	Has a specified identifier already been declared in this scope? (Used to avoid duplicate declarations.)
ADD:	Add an identifier and its attributes to the symbol table.

المنارة للاستشارات

RETRIEVE:	Return the attributes associated (in the most local scope in which it occurs) with a specified identifier.

Though many references provide insights into how these operations can be implemented, none presents a formal definition (other than implementations) of exactly what they mean. The abstract concept "symbol table" thus goes undefined. Those who attempt to write compilers in a top-down fashion suffer from a similar problem. Early refinements of parts of the compiler make use of the basic symbol table operations, but the "meaning" of these operations is provided only by subsequent levels of refinement. This is infelicitous in that the clear separation of levels of abstraction is lost and with it many of the advantages of top-down design. By providing axiomatic semantics for the operations, this problem can be avoided.

The thought of providing rigorous definitions for so many operations may, at first, seem a bit intimidating. Nevertheless, if one is to understand the refinement, one must know what each operation means. The following specification of abstract type Symboltable supplies these meanings.

Type: Symboltable

Operations:

INIT:	→ Symboltable
ENTERBLOCK:	Symboltable → Symboltable
LEAVEBLOCK:	Symboltable → Symboltable
ADD:	Symboltable × Identifier × Attributelist → Symboltable
IS_INBLOCK?:	Symboltable × Identifier → Boolean
RETRIEVE:	Symboltable × Identifier → Attributelist

المنارة للاستشارات

Axioms:

(1) LEAVEBLOCK(INIT) = error
(2) LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab
(3) LEAVEBLOCK(ADD(symtab, id, attrs)) = LEAVE-
BLOCK(symtab)
(4) IS_INBLOCK? (INIT, id) = false
(5) IS_INBLOCK? (ENTERBLOCK(symtab), id) = false
(6) IS_INBLOCK? (ADD(symtab, id, attrs), idl) =
if IS_SAME? (id, idl)[2]
then true
else IS_INBLOCK? (symtab, id)
(7) RETRIEVE(INIT, id) = error
(8) RETRIEVE(ENTERBLOCK(symtab), id) =
RETRIEVE(symtab, id)
(9) RETRIEVE(ADD(symtab, id, attrs), idl)=
if IS_SAME? (id, idl)
then attrs
else RETRIEVE(symtab, idl)

This set of relations serves a dual purpose. Not only does it define an abstract type that can be used in the specification of various parts of the compiler, but it also provides a complete self-contained specification for a major subsystem of the compiler. If one wished to delegate the design and implementation of the symbol table subsystem, the algebraic characterization of the abstract type would (unlike the informal description in, say, McKeeman [15]) be a sufficient specification of the problem. In fact, the procedure discussed earlier can be used to formally prove the sufficient-completeness of this specification.

The next step in the design process is to further refine type Symboltable, i.e. to provide implementa-

---

[2] The definition of IS_SAME? is part of the specification of an independently defined type Identifier.

المنارة للاستشارات

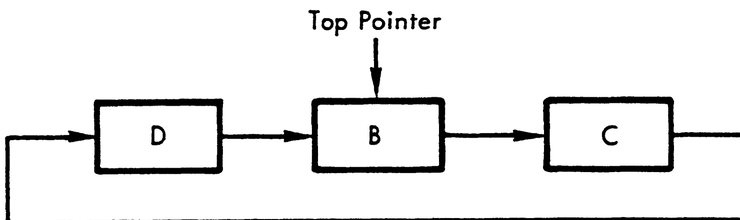
tions of the operations of the type. These implementations will implicitly furnish representation for values of type Symboltable.

A representation of a type T consists of (i) any interpretation (implementation) of the operations of the type that is a model for the axioms of the specification of T, and (ii) a function Φ that maps terms in the model domain onto their representatives in the abstract domain. (This is basically the abstraction function of Hoare [12].)

It is important to note that Φ may not have a proper inverse. Consider, for example, type Bounded Queue (with a maximum length of three). A reasonable representation of the values of this type might be based on a ring-buffer and top pointer. Given this representation, the program segment:

```
x := EMPTY.Q
x := ADD.Q(x, A)
x := ADD.Q(x, B)
x := ADD.Q(x, C)
x := REMOVE.Q(x)
x := ADD.Q(x, D)
```

would translate to a representation for x of the form:



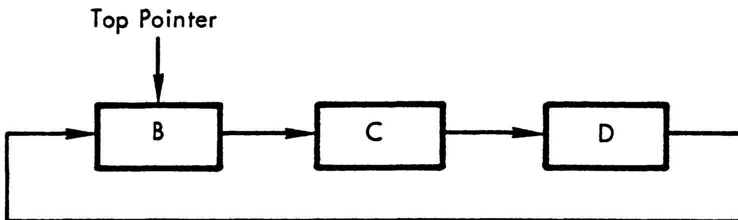
Similarly:

```
x := EMPTY.Q
x := ADD.Q(x, B)
```

المنارة للاستشارات

x := ADD.Q(x, C)
x := ADD.Q(x, D)

would yield a representation for x of the form:



It is clear that these two representations though not identical, refer to the same abstract value. That is to say, the mapping from values to representations, Φ^{-1}, may be one-to-many.

The representation of type Symboltable will make use of the abstract data types Stack (of arrays) and Array (of attributelists) as defined below.

Type: Stack

Operations:

NEWSTACK: $\rightarrow$ Stack
PUSH: Stack × Array $\rightarrow$ Stack
POP: Stack $\rightarrow$ Stack
TOP: Stack $\rightarrow$ Array
IS_NEWSTACK?: Stack $\rightarrow$ Boolean
REPLACE: Stack × Array $\rightarrow$ Stack

Axioms:

(10) IS_NEWSTACK? (NEWSTACK) = true
(11) IS_NEWSTACK? (PUSH(stk, arr)) = false
(12) POP(NEWSTACK) = error
(13) POP(PUSH(stk, arr)) = stk
(14) TOP(NEWSTACK) = error
(15) TOP(PUSH(stk, arr)) = arr
(16) REPLACE(stk, arr) = **if** IS_NEWSTACK? (stk)
then error
else PUSH(POP(stk), arr)

المنارة للاستشارات

Type: Array

Operations:

```
EMPTY:                        → Array
ASSIGN:          Array × Identifier × Attributelist → Array
READ:            Array × Identifier → Attributelist
IS_UNDEFINED?:   Array × Identifier → Boolean
```

Axioms:

```
(17) IS_UNDEFINED? (EMPTY, id) = true
(18) IS_UNDEFINED? (ASSIGN(arr, id, attrs), idl) =
        if IS_SAME? (id, idl)
          then false
          else IS_UNDEFINED? (arr, idl)
(19) READ(EMPTY, id) = error
(20) READ(ASSIGN(arr, id, attrs), idl) = if IS_SAME? (id, idl)
                                           then attrs
                                           else READ(arr, idl)
```

The general scheme of the representation of type Symboltable is to treat a value of the type as a stack of arrays (with index type Identifier), where each array contains the attributes for the identifiers declared in a single block. For every function f in the more abstract domain (e.g. type Symboltable), a function f' is defined in the lower-level domain; thus we have:

```
INIT':                       → Stack
ENTERBLOCK':     Stack → Stack
LEAVEBLOCK':     Stack → Stack
ADD':            Stack × Identifier × Attributelist → Stack
IS_INBLOCK?':    Stack × Identifier → Boolean
RETRIEVE':       Stack × Identifier → Attributelist
```

The “code” for each of these functions is (“::” means “is defined as”):

```
INIT' :: PUSH(NEWSTACK, EMPTY)
ENTERBLOCK'(stk) :: PUSH(stk, EMPTY)
LEAVEBLOCK'(stk) :: if IS_NEWSTACK? (POP(stk))
```

المنارة للاستشارات

```
                                   then error
                                   else POP(stk)
ADD'(stk, id, attrs) :: REPLACE(stk, ASSIGN(TOP(stk), id,
  attrs))
IS_INBLOCK?'(stk, id) :: if IS_NEWSTACK? (stk)
                            then false
                            else ¬ IS_UNDEFINED? (TOP(stk),
                              id)
RETRIEVE'(stk, id) :: if IS_NEWSTACK? (stk)
                        then error
                        else ¬ IS_UNDEFINED? (TOP(stk), id)
                          then RETRIEVE'(POP(stk), id)
                          else READ(TOP(stk), id)
```

The interpretation function Φ is defined by:

```
(a) Φ(error) = error
(b) Φ(NEWSTACK) = error
(c) Φ(PUSH(stk, EMPTY)) = if IS_NEWSTACK? (stk)
                             then INIT
                             else ENTERBLOCK(Φ(stk))
(d) Φ(PUSH(stk, ASSIGN(arr, id, attrs))) = ADD(ΦPUSH(stk,
      arr)), id, attrs))
```

Before continuing to refine these operations, i.e. before supplying representations for types Array and Stack, let us consider the problem of proving that the above implementation of type Symboltable is correct.

In the course of such a proof two kinds of invariants may have to be verified: inherent invariants and representation invariants. The inherent invariants represent those invariant relationships that must be maintained by any representation of the type. They correspond to the axioms used in the specification of the type.. A representation invariant, on the other hand, is peculiar to a particular representation of a type.

The basic procedure followed in verifying the inherent invariants is to take each axiom for type Symboltable and replace all instances of each function appearing

المنارة للاستشارات

in the axiomatization with its interpretation. Then, by using the axiomatizations of the operations used in constructing the representations, it is shown that the left-hand side of each axiom is equivalent to the right-hand side of that axiom. That is to say, they represent the same abstract value.

What must be shown therefore is that for every relation $f'(x^*) = z$ (where x^* is a list, possibly empty, of arguments), derived from the axiomatization of type Symboltable,

(a) if the range of f is the type being defined (i.e., Symboltable), $\Phi(f'(x^*)) = \Phi(z)$ for all legal assignments to the free variables of x^* and z, or

(b) if the range of f is a type other than that being defined, $f'(x^*) = z$ for all legal assignments to the free variables of x^* and z.

To show this, we have at our disposal a proof system consisting of the axioms and rules of inference of our programming language plus the axioms defining the abstract types used in the representation.

The proof depends upon the assumption that objects of type Symboltable are created and manipulated only via the operations defined in the specification of that type. (The use of classes as described in Palme [18] makes this assumption relatively easy to verify.) All that need be shown is that INIT′ establishes the invariants and that if on entry to an operation all invariants hold for all objects of type Symboltable to be manipulated by that operation, then all invariants on those objects hold upon completion of that operation. More complete discussions of how this may be done are contained in Guttag [8], Spitzen [21], and Wegbreit [23] (where it is called generator induction).

المنارة للاستشارات

To verify that the implementation is consistent with Axioms 1 through 8 is quite straightforward. (It has, in fact, been done completely mechanically by David Musser [17] using the program verification system at the University of Southern California Information Sciences Institute [7]. Thus the proofs will not be presented here. Axiom 9, on the other hand, presents some problems that make the portion of the proof pertinent to that axiom worth examining.

The proof that the implementation satisfies Axiom 9 is based upon an assumption about the environment in which the operations of the type are to be used. In effect, the assumption asserts that an identifier is never added to an empty symbol table, i.e. a scope must have been established (on a more concrete level, an array must have been pushed onto the stack) before an identifier can be added. The concrete manifestation of this assumption is formally expressed:

Assumption 1. For any term, ADD′(symtab, id, attrs), IS_NEWSTACK? (symtab) = false.

The validity of the above assumption can be assured by adding to the implementation of ADD′ a check for this condition and having it execute an ENTERBLOCK′ if necessary. This would make it possible to construct a completely self-contained proof of the correctness of the representation. In most cases, however, it would also introduce needless inefficiency. The compiler must somewhere check for mismatched (i.e. extra) "end" statements. Any check in ADD′ would therefore be redundant.

This observation leads to a notion of conditional correctness, i.e. the representation of the abstract type is correct if the enclosing program obeys certain constraints. In practice, this is often an extremely useful

المنارة للاستشارات

notion of correctness, especially if the constraint is easily checked. If, on the other hand, the environment in which the abstract type is to be used is unknown (e.g. if the type is to be included in a library), this is probably unacceptably dangerous. Given the above assumption, the verification of Axiom 9 is straightforward but lengthy and will therefore not be presented here. It does appear in Guttag [8].

Now we know that, given implementations of types Stack and Array that are consistent with their specifications, the implementation of type Symboltable is "correct." Assuming PL/I-like based variables, pointers, and structures, the implementation of type Stack is trivial. The basic scheme is to represent a stack as a pointer to a list of structures of the form:

```
1. stack elem based,
   2. val Array,
   2. prev pointer.
```

The operations may be implemented as follows (PL/I keywords have been boldfaced):

```
NEWSTACK' :: null
PUSH'(symtab, newblock) ::
  procedure(symtab: pointer, newblock: Array)returns(pointer)
    declare elem_ptr pointer
    allocate(stack_elem) set(elem_ptr)
    elem_ptr → prev := symtab
    elem_ptr → val := newblock
    return(elem_ptr)
  end
POP'(symtab) ::
  procedure(symtab: pointer) returns(pointer)
    if symtab = null
      then return(error)
      else return(symtab → prev)
  end
```

المنارة للاستشارات

```
TOP'(symtab) ::
  procedure(symtab: pointer) returns(Array)
    if symtab = null
      then return(error)
      else return(symtab → val)
  end
IS_NEWSTACK?'(symtab) :: symtab = null
REPLACE'(symtab, newblock) ::
  procedure(symtab: pointer, newblock: Array) returns(pointer)
    if symtab = null
      then return(error)
      else symtab → val := newblock
        return(symtab)
  end
```

Φ is defined by the mapping:

```
Φ(symtab) :: if symtab = null
               then NEWSTACK
               else PUSH(Φ(symtab → prev), symtab → val))
```

The implementation chosen for type Array is a bit more complicated. The basic scheme is to represent an array as a PL/I-like array, hash_tab, of *n* pointers to lists of structures of the form:

```
1. entry based,
  2. id Identifier
  2. attributes Attributelist,
  2. next pointer.
```

The correct element of hash_tab is selected by performing a hash on values of type Identifier. Therefore, in addition to the operations used in the code above, the implementation of type Array uses an operation

HASH:Identifier → {1, 2, . . . , n}

which is assumed to be defined in the type Identifier specification. The "code" implementing type Array is:

المنارة للاستشارات

```
declare hash_tab(n) pointer based

EMPTY' ::
  procedure returns(pointer)
    declare new_hash_tab pointer
    allocate (hash_tab) set (new_hash_tab)
    do i := 1 to n
      new_hash_tab → hash_tab(i) := null
    end
    return(new_hash_tab)
  end
ASSIGN'(arr, indx, atr) ::
  procedure(arr: pointer, indx: Identifier, atr: Attributelist)
        returns(pointer)
    declare new_entry pointer
    allocate(entry) set (new_entry)
      new_entry → id := indx
      new_entry → attributes := atr
      new_entry → next := arr → hash_tab(HASH(indx))
      arr → hash_tab(HASH(indx)) := new_entry
    return(arr)
  end
READ'(arr, indx) ::
  procedure(arr: pointer, indx: Identifier) returns(Attributelist)
    declare bucket_ptr pointer
      bucket_ptr := arr → hash_tab(HASH(indx))
    do while(bucket_ptr ≠ null & ¬ IS_SAME?(bucket_ptr → id,
        indx))
      bucket_ptr := bucket_ptr → next
    end
    if bucket_ptr = null
      then return(error)
      else return (bucket_ptr → attributes)
  end
IS_UNDEFINED?'(arr, indx) ::
  procedure(arr: pointer, indx: Identifier) returns(Boolean)
    declare bucket_ptr pointer
      bucket_ptr := arr → hash_tab(HASH(indx))
    do while (bucket_ptr ≠ null & ¬ IS_SAME? (bucket_ptr → id,
        indx))
```

المنارة للاستشارات

```
    bucket_ptr := arr → hash_tab(HASH(indx))
  do while (bucket_ptr ≠ null & ¬ IS_SAME? (bucket_ptr → id,
      indx))
    bucket_ptr := bucket_ptr → next
  end
  return (bucket_ptr = null)
end
```

As one might expect, Φ is a bit more complex for this representation. It is defined by using two intermediate functions: Φ1 to construct a union over all the entries in the hash table, and Φ2 to construct a union over the elements of an individual bucket.

```
(a) Φ(hash_tab_ptr) = Φ1(hash_tab_ptr, EMPTY, 1)
(b) Φ1(hash_tab_ptr, arr, i) =
       if i > n
         then arr
         else Φ1(hash_tab_ptr, Φ2(hash_tab_ptr → hash_tab(i), arr),
           i + 1)
(c) Φ2(bucket_ptr,arr) =
       if bucket_ptr = null
         then arr
         else ASSIGN(Φ2(bucket_ptr → next, arr), bucket_ptr → id,
           bucket_ptr → attributes)
```

The design of the symbol table subsystem of the compiler is now essentially complete. Given implementations of types Identifier and Attributelist and some obvious syntactic transformations, the above code could be compiled by a PL/I compiler. Before doing so, however, it would be wise to prove that the implementations of types Stack and Array are consistent with the specifications of those types. While such a proof would involve substantial issues related to the general program verification problem (e.g. vis à vis the integrity of the pointers and the question of modifying shared data structures), it would not shed further light on the role

المنارة للاستشارات

of abstract data types in program verification and is not presented in these pages.

The ease with which algebraic specifications can be adapted for different applications is one of the major strengths of the technique. Because the relationships among the various operations appear explicitly, the process of deciding which axioms must be altered to effect a change is straightforward. Let us consider a rather substantial change in the language to be compiled. Assume that the language permits the inheritance of global variables only if they appear in a "knows list," which lists, at block entry, all nonlocal variables to be used within the block [6]. The symbol table operations in a compiler for such a language would be much like those already discussed. The only difference visible to parts of the compiler other than the symbol table module would be in the ENTERBLOCK operation: It would have to be altered to include an argument of abstract type Knowlist. Within the specification of type Symboltable, all relations, and only those relations, that explicitly deal with the ENTERBLOCK operation would have to be altered. An appropriate set of axioms would be:

```
IS_INBLOCK?(ENTERBLOCK(symtab, klist), id) = false
LEAVEBLOCK(ENTERBLOCK(symtab, klist)) = symtab
RETRIEVE(ENTERBLOCK(symtab, klist), id) =
  if IS_IN?(klist, id)
    then RETRIEVE(symtab, id)
    else error
```

Note that the above relations are not well defined. The undefined symbol IS_IN?, an operation of the abstract type Knowlist, appears in the third axiom. The solution to this problem is simply to add another level

المنارة للاستشارات

to the specification by supplying an algebraic specification of the abstract type Knowlist. An appropriate set of operations might be:

```
CREATE:                           → Knowlist
APPEND:   Knowlist × Identifier → Knowlist
IS_IN?:   Knowlist × Identifier → Boolean
```

These operations could then be precisely defined by the following axioms:

```
IS_IN?(CREATE) = false
IS_IN?(APPEND(klist, id), idl) = if IS_SAME?(id, idl)
                                    then true
                                    else IS_IN?(klist, idl)
```

The implementation of abstract type Knowlist is trivial. The changes necessary to adapt the previously presented implementation of abstract type Symboltable would be more substantial. The kind of changes necessary can, however, be inferred from the changes made to the axiomatization.

5. Conclusions

We have not yet applied the techniques discussed in this paper to realistically large software projects. Nevertheless, there is reason to believe that the techniques demonstrated will "scale up." The size and complexity of a specification at any level of abstraction are essentially independent of both the size and complexity of the system being described and of the amount of mechanism ultimately used in the implementation. The independence springs in large measure from the ability to separate the precise meaning of a complex

المنارة للاستشارات

abstract data type from the details involved in its implementation. It is the ability to be precise without being detailed that encourages the belief that the approach outlined here can be applied even to "very large" systems can and perhaps reduce systems that were formerly "very large" (i.e. incomprehensible) to more manageable proportions.

Abstract types may thus play a vital role in the formulation and presentation of precise specifications for software. Many complex systems can be viewed as instances of an abstract type. A database management system, for example, might be completely characterized by an algebraic specification of the various operations available to users. For those systems that are not easily totally characterized in terms of algebraic relations, the use of algebraic type specifications to abstract various complex subsystems may still make a substantial contribution to the design process. The process of functional decomposition requires some means for specifying the communication among the various functions—data often fulfills this need. The use of algebraic specifications to provide abstract definitions of the operations used to establish communication among the various functions may thus play a significant role in simplifying the process of functional abstraction.

The extensive use of algebraic specifications of abstract types may also lead to better-designed data structures. The premature choice of a storage structure and set of access routines is a common cause of inefficiencies in software. Because they serve as the main means of communication among the various components of many systems, the data structures are often the first components designed. Unfortunately, the information required to make an intelligent choice among the var-

المنارة للاستشارات

ious options is often not available at this stage of the design process. The designer may, for example, have poor insight into the relative frequency of the various operations to be performed on a data structure. By providing a representation-free, yet precise, description of the operations on a data structure, algebraic type definitions enable the designer to delay the moment at which a storage structure must be designed and frozen.

The second area in which we expect the algebraic specification of abstract types to have a substantial impact is on proofs of program properties. For verifications of programs that use abstract types, the algebraic specification of the types used provides a set of powerful rules of inference that can be used to demonstrate the consistency of the program and its specification. That is to say, the presence of axiomatic definitions of the abstract types provides a mechanism for proving a program to be consistent with its specifications, provided that the implementations of the abstract operations that it uses are consistent with their specifications. Thus a technique for factoring the proof is provided, for the algebraic type definitions serve as the specification of intent at a lower level of abstraction. For proofs of the correctness of representations of abstract types, the algebraic specification provides exactly those assertions that must be verified. The value of having such a set of assertions available should be apparent to anyone who has attempted to construct, *a posteriori*, assertions appropriate to a correctness proof for a program. A detailed discussion of the use of algebraic specifications in a semiautomatic program verification system is contained in Guttag [10].

المنارة للاستشارات

a single value. Most programs, on the other hand, are laden with procedures that return several values (via parameters) or no value at all. (The latter kind of procedure is invoked purely for its side effects.) The inability to specify such procedures is a serious problem, but one that we believe can be solved with only minor changes to the specification techniques [10].

The value of abstraction in general and abstraction of data types in particular has been stressed throughout this paper. Nevertheless, the process is not without its dangers. It is all too easy to create abstractions that ignore crucial distinctions or attributes. The specification technique presented here, for example, provides no mechanism for specifying performance constraints and thus encourages one to ignore distinctions based on such criteria. In some environments, such considerations are crucial, and to abstract them out can be disastrous.

Another problem with algebraic specifications is that they supply little direction to implementors. Only experience will tell how easy it is to go from an algebraic specification to an implementation. It is clear, however, that the transition is less easy than from an operational specification.

Our most important reservation pertains to the ease with which algebraic specifications can be constructed and read. They should present no problem to those with formal training in computer science. At present, however, most people involved in the production of software have no such training. The extent to which the techniques described in this paper are generally applicable is thus somewhat open to conjecture.

المنارة للاستشارات

Acknowledgment. The author is greatly indebted to J.J. Horning of the University of Toronto, who, as the author's thesis supervisor, provided three years of good advice.

References

1. Batey, M., Ed. Working Draft of ECMA/ANSI PL/I Standard Tenth Rev., ANSI, New York, (Sept. 1973).

2. Birkhoff, G., and Lipson, J.D. Heterogeneous algebras. *J. Combinatorial Theory 8* (1970), 115–133.

3. Dahl, O.-J., Nygaard, K., and Myhrhaug, B. The SIMULA 67 Common Base Language. Norwegian Comptng. Centre, Oslo, 1968.

4. Dijkstra, E.W. Notes on structured programming. In *Structured Programming*, Academic Press, New York, 1972.

5. Floyd, R.W. Assigning Meaning to Programs. Proc. Symp. in Applied Math., Vol. XIX, AMS, Providence, R.I., 1967, pp. 19–32.

6. Gannon, J.D. Language design to enhance programming reliability. Ph.D. Th., Comptr. Syst. Res. Group Tech. Rep. CSRG-47, Dept. Comptr. Sci., U. of Toronto, Ontario, 1975.

7. Good, D.I., London, R.L., and Bledsoe, W.W. An interactive program verification system. *IEEE Trans. on Software Engineering SE-1*, 1 (March 1975), 59–67.

8. Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. Th., Comptr. Syst. Res. Group Tech. Rep. CSRG-59, Dept. Comptr. Sci. 1975, U. of Toronto, Ontario, 1975.

9. Guttag, J.V. and Horning, J.J., The algebraic specifications of abstract data types. *Acta Informatica* (to appear).

10. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. Tech. Rep., Inform. Sci. Inst., U. of Southern California, Los Angeles, 1976.

11. Guttag, J.V., Horowitz, E., and Musser, D.R. The design of data type specifications. Proc. Second Int. Conf. on Software Eng., San Francisco, Oct. 1976, pp. 414–420.

12. Hoare, C.A.R., Proof of correctness of data representations. *Acta Informatica 1* (1972), 271–281.

13. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335–355.

14. Liskov, B.H., and Zilles, S.N. Programming with abstract data types. Proc. ACM SIGPLAN Symp. on Very High Level Languages, SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50–59.

المنارة للاستشارات

15. McKeeman, W.M., Symbol Table Access. In *Compiler Construction, An Advanced Course*, T.L. Bauer, and J. Eichel, Eds., Springer-Verlag, New York, 1974.
16. Morris, J.H. Types are not sets. Conf. Rec. ACM Symp. on the Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 120–124.
17. Musser, D. Private communication, 1975.
18. Palme, J. Protected program modules in SIMULA 67. FOAP Rep. C8372-M3(E5), Res. Inst. of National Defense, Stockholm, 1973.
19. Parnas, D.L. A technique for the specification of software modules with examples. *Comm. ACM 15,* 5 (May 1973), 330–336.
20. Parnas, D.L. Information distribution aspects of design methodology. Information Processing 71, North Holland Pub. Co., Amsterdam, 1971, pp. 339–344.
21. Spitzen, J., and Wegbreit, B. The verification and synthesis of data structures. *Acta Informatica 4* (1975), 127–144.
22. Standish, T.A. Data structures: An axiomatic approach. BBN Rep. No. 2639, Bolt, Beranek and Newman, Cambridge, Mass., (1973).
23. Wegbreit, B., and Spitzen, J. Proving properties of complex data structures. *J. ACM 23,* 2 (April 1976), 389–396.
24. Wulf, W.A., London, R.L., and Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology. USC Inform. Sci. Tech. Rep., U. of Southern California, Los Angeles, 1976.
25. Zilles, S.N. Abstract specifications for data types. IBM Res. Lab., San Jose, Calif., 1975.

المنارة للاستشارات

C.A.R. Hoare

An Axiomatic Basis for Computer Programming

Communications of the ACM, Vol. 12 (10), 1969
pp. 576–580, 583

المنارة للاستشارات

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to

* Department of Computer Science

المنارة للاستشارات

elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are con-

المنارة للاستشارات

TABLE I

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leqslant x \supset (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

TABLE II

1. Strict Interpretation

+	0	1	2	3
0	0	1	2	3
1	1	2	3	*
2	2	3	*	*
3	3	*	*	*

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	*	*
3	0	3	*	*

* nonexistent

2. Firm Boundary

+	0	1	2	3
0	0	1	2	3
1	1	2	3	3
2	2	3	3	3
3	3	3	3	3

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	3
3	0	3	3	3

3. Modulo Arithmetic

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

المنارة للاستشارات

fined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$$A10_I \quad \neg \exists x \forall y \qquad (y \leqslant x),$$

where all finite arithmetics satisfy:

$$A10_F \quad \forall x \qquad (x \leqslant \max)$$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S \quad \neg \exists x \quad (x = \max + 1)$ (strict interpretation)

$A11_B \quad \max + 1 = \max$ (firm boundary)

المنارة للاستشارات

$A11_M$ max + 1 = 0 (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however, these properties will not necessarily obtain, unless the program is executed on an implementation which satisfies the chosen axiom.

3. Program Execution

As mentioned above, the purpose of this study is to provide a logical basis for proofs of the properties of a program. One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program. These assertions will usually not ascribe particular values to each variable, but will rather specify certain general properties of the values and the relationships holding between them. We use the normal notations of mathematical logic to express these assertions, and the familiar rules of operator precedence have been used wherever possible to improve legibility.

In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q) and a description of the result of its execution (R), we introduce a new notation:

$$P\{Q\}R.$$

This may be interpreted "If the assertion P is true before initiation of a program Q, then the assertion R will be

المنارة للاستشارات

true on its completion." If there are no preconditions imposed, we write **true** $\{Q\}R$.[1]

The treatment given below is essentially due to Floyd [8] but is applied to texts rather than flowcharts.

3.1. Axiom of Assignment

Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic.

Consider the assignment statement:

$$x := f$$

where

x is an identifier for a simple variable;

f is an expression of a programming language without side effects, but possibly containing x.

Now any assertion $P(x)$ which is to be true of (the value of) x *after* the assignment is made must also have been true of (the value of) the expression f, taken *before* the assignment is made, i.e. with the old value of x. Thus if $P(x)$ is to be true after the assignment, then $P(f)$ must be true before the assignment. This fact may be expressed more formally:

D0 Axiom of Assignment

$\vdash P_0 \{x := f\}\, P$

where

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting f for all occurrences of x.

[1] If this can be proved in our formal system, we use the familiar logical symbol for theoremhood: $\vdash P\ \{Q\}\ R$

للاستشارات المنارة

It may be noticed that D0 is not really an axiom at all, but rather an axiom schema, describing an infinite set of axioms which share a common pattern. This pattern is described in purely syntactic terms, and it is easy to check whether any finite text conforms to the pattern, thereby qualifying as an axiom, which may validly appear in any line of a proof.

3.2. Rules of Consequence

In addition to axioms, a deductive science requires at least one rule of inference, which permits the deduction of new theorems from one or more axioms or theorems already proved. A rule of inference takes the form "If $\vdash X$ and $\vdash Y$ then $\vdash Z$", i.e. if assertions of the form X and Y have been proved as theorems, then Z also is thereby proved as a theorem. The simplest example of an inference rule states that if the execution of a program Q ensures the truth of the assertion R, then it also ensures the truth of every assertion logically implied by R. Also, if P is known to be a precondition for a program Q to produce result R, then so is any other assertion which logically implies P. These rules may be expressed more formally:

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$
If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

3.3. Rule of Composition

A program generally consists of a sequence of statements which are executed one after another. The statements may be separated by a semicolon or equivalent symbol denoting procedural composition: $(Q_1; Q_2; \cdots; Q_n)$. In order to avoid the awkwardness of dots, it is possible to deal initially with only two statements $(Q_1; Q_2)$, since longer sequences can be reconstructed by nesting, thus $(Q_1; (Q_2; (\cdots (Q_{n-1}; Q_n) \cdots)))$. The removal of the brackets of

المنارة للاستشارات

this nest may be regarded as convention based on the associativity of the ";-operator", in the same way as brackets are removed from an arithmetic expression $(t_1 + (t_2 + (\cdots (t_{n-1} + t_n) \cdots)))$.

The inference rule associated with composition states that if the proven result of the first part of a program is identical with the precondition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied.

In more formal terms:

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1;\ Q_2)\}R$

3.4. Rule of Iteration

The essential feature of a stored program computer is the ability to execute some portion of program (S) repeatedly until a condition (B) goes false. A simple way of expressing such an iteration is to adapt the Algol 60 **while** notation:

while B **do** S

In executing this statement, a computer first tests the condition B. If this is false, S is omitted, and execution of the loop is complete. Otherwise, S is executed and B is tested again. This action is repeated until B is found to be false. The reasoning which leads to a formulation of an inference rule for iteration is as follows. Suppose P to be an assertion which is always true on completion of S, provided that it is also true on initiation. Then obviously P will still be true after any number of iterations of the statement S (even no iterations). Furthermore, it is known that the controlling condition B is false when the iteration finally terminates. A slightly more powerful formulation is possible in light of the fact that B may be assumed to be true on initiation of S:

المنارة للاستشارات

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P$

3.5. Example

The axioms quoted above are sufficient to construct the proof of properties of simple programs, for example, a routine intended to find the quotient q and remainder r obtained on dividing x by y. All variables are assumed to range over a set of nonnegative integers conforming to the axioms listed in Table I. For simplicity we use the trivial but inefficient method of successive subtraction. The proposed program is:

$$((r := x;\quad q := 0);\quad \textbf{while}\quad y \leqslant r \textbf{ do } (r := r - y;\quad q := 1 + q))$$

An important property of this program is that when it terminates, we can recover the numerator x by adding to the remainder r the product of the divisor y and the quotient q (i.e. $x = r + y \times q$). Furthermore, the remainder is less than the divisor. These properties may be expressed formally:

$$\textbf{true } \{Q\}\ \neg y \leqslant r \wedge x = r + y \times q$$

where Q stands for the program displayed above. This expresses a necessary (but not sufficient) condition for the "correctness" of the program.

A formal proof of this theorem is given in Table III. Like all formal proofs, it is excessively tedious, and it would be fairly easy to introduce notational conventions which would significantly shorten it. An even more powerful method of reducing the tedium of formal proofs is to derive general rules for proof construction out of the simple rules accepted as postulates. These general rules would be shown to be valid by demonstrating how every theorem proved with their assistance could equally well (if more tediously) have been proved without. Once a powerful set

المنارة للاستشارات

TABLE III

Line number	Formal proof	Justification
1	$\mathbf{true} \supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0 \{r := x\} x = r + y \times 0$	D0
3	$x = r + y \times 0 \; \{q := 0\} \; x = r + y \times q$	D0
4	$\mathbf{true} \; \{r := x\} \; x = r + y \times 0$	D1 (1, 2)
5	$\mathbf{true} \; \{r := x; \;\; q := 0\} \; x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x =$ $(r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q) \{r := r-y\} x =$ $r + y \times (1+q)$	D0
8	$x = r + y \times (1+q) \{q := 1+q\} x =$ $r + y \times q$	D0
9	$x = (r-y) + y \times (1+q) \{r := r-y;$ $q := 1+q\} \; x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r \; \{r := r-y;$ $q := 1+q\} \; x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q \; \{\mathbf{while} \; y \leqslant r \; \mathbf{do}$ $(r := r-y; \;\; q := 1+q)\}$ $\neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\mathbf{true} \; \{((r := x; \;\; q := 0); \;\; \mathbf{while} \; y \leqslant r \; \mathbf{do}$ $(r := r-y; \;\; q := 1+q))\} \; \neg y \leqslant r \wedge x =$ $r + y \times q$	D2 (5, 11)

NOTES

1. The left hand column is used to number the lines, and the right hand column to justify each line, by appealing to an axiom, a lemma or a rule of inference applied to one or two previous lines, indicated in brackets. Neither of these columns is part of the formal proof. For example, line 2 is an instance of the axiom of assignment (D0); line 12 is obtained from lines 5 and 11 by application of the rule of composition (D2).

2. Lemma 1 may be proved from axioms A7 and A8.

3. Lemma 2 follows directly from the theorem proved in Sec. 2.

المنارة للاستشارات

of supplementary rules has been developed, a "formal proof" reduces to little more than an informal indication of how a formal proof could be constructed.

4. General Reservations

The axioms and rules of inference quoted in this paper have implicitly assumed the absence of side effects of the evaluation of expressions and conditions. In proving properties of programs expressed in a language permitting side effects, it would be necessary to prove their absence in each case before applying the appropriate proof technique. If the main purpose of a high level programming language is to assist in the construction and verification of correct programs, it is doubtful whether the use of functional notation to call procedures with side effects is a genuine advantage.

Another deficiency in the axioms and rules quoted above is that they give no basis for a proof that a program successfully terminates. Failure to terminate may be due to an infinite loop; or it may be due to violation of an implementation-defined limit, for example, the range of numeric operands, the size of storage, or an operating system time limit. Thus the notation "$P\{Q\}R$" should be interpreted "provided that the program successfully terminates, the properties of its results are described by R." It is fairly easy to adapt the axioms so that they cannot be used to predict the "results" of nonterminating programs; but the actual use of the axioms would now depend on knowledge of many implementation-dependent features, for example, the size and speed of the computer, the range of numbers, and the choice of overflow technique. Apart from proofs of the avoidance of infinite loops, it is probably better to prove the "conditional" correctness of a program and rely on an implementation to give a warning if it has had to abandon execution of the program as a result of violation of an implementation limit.

Finally it is necessary to list some of the areas which have not been covered: for example, real arithmetic, bit and character manipulation, complex arithmetic, fractional arithmetic, arrays, records, overlay definition, files, input/output, declarations, subroutines, parameters, recursion, and parallel execution. Even the characterization of integer arithmetic is far from complete. There does not appear to be any great difficulty in dealing with these points, provided that the programming language is kept simple. Areas which do present real difficulty are labels and jumps, pointers, and name parameters. Proofs of programs which made use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

5. Proofs of Program Correctness

The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program, provided that the implementation of the programming language conforms to the axioms and rules which have been used in the proof. This fact itself might also be established by deductive reasoning, using an axiom set which describes the logical properties of the hardware circuits. When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.

The practice of supplying proofs for nontrivial programs will not become widespread until considerably more power-

للاستشارات

ful proof techniques become available, and even then will not be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error. At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out in particular cases and to modify it if the results produced do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. The time spent in this program testing is often more than half the time spent on the entire programming project; and with a realistic costing of machine time, two thirds (or more) of the cost of the project is involved in removing errors during this phase.

The cost of removing errors discovered after a program has gone into use is often greater, particularly in the case of items of computer manufacturer's software for which a large part of the expense is borne by the user. And finally, the cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus the practice of program proving is not only a theoretical pursuit, followed in the interests of academic respectability, but a serious recommendation for the reduction of the costs associated with programming error.

The practice of proving programs is likely to alleviate some of the other problems which afflict the computing world. For example, there is the problem of program documentation, which is essential, firstly, to inform a potential user of a subroutine how to use it and what it accomplishes, and secondly, to assist in further development when it becomes necessary to update a program to meet changing circumstances or to improve it in the light of increased knowledge. The most rigorous method of formulating the

المنارة للاستشارات

purpose of a subroutine, as well as the conditions of its proper use, is to make assertions about the values of variables before and after its execution. The proof of the correctness of these assertions can then be used as a lemma in the proof of any program which calls the subroutine. Thus, in a large program, the structure of the whole can be clearly mirrored in the structure of its proof. Furthermore, when it becomes necessary to modify a program, it will always be valid to replace any subroutine by another which satisfies the same criterion of correctness. Finally, when examining the detail of the algorithm, it seems probable that the proof will be helpful in explaining not only *what* is happening but *why*.

Another problem which can be solved, insofar as it is soluble, by the practice of program proofs is that of transferring programs from one design of computer to another. Even when written in a so-called machine-independent programming language, many large programs inadvertently take advantage of some machine-dependent property of a particular implementation, and unpleasant and expensive surprises can result when attempting to transfer it to another machine. However, presence of a machine-dependent feature will always be revealed in advance by the failure of an attempt to prove the program from machine-independent axioms. The programmer will then have the choice of formulating his algorithm in a machine-independent fashion, possibly with the help of environment enquiries; or if this involves too much effort or inefficiency, he can deliberately construct a machine-dependent program, and rely for his proof on some machine-dependent axiom, for example, one of the versions of A11 (Section 2). In the latter case, the axiom must be explicitly quoted as one of the preconditions of successful use of the program. The program can still, with complete confidence, be transferred to any other machine which happens to satisfy the

المنارة للاستشارات

same machine-dependent axiom; but if it becomes necessary to transfer it to an implementation which does not, then all the places where changes are required will be clearly annotated by the fact that the proof at that point appeals to the truth of the offending machine-dependent axiom.

Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased only at the price of simplicity.

6. Formal Language Definition

A high level programming language, such as ALGOL, FORTRAN, or COBOL, is usually intended to be implemented on a variety of computers of differing size, configuration, and design. It has been found a serious problem to define these languages with sufficient rigour to ensure compatibility among all implementors. Since the purpose of compatibility is to facilitate interchange of programs expressed in the language, one way to achieve this would be to insist that all implementations of the language shall "satisfy" the axioms and rules of inference which underlie proofs of the properties of programs expressed in the language, so that all predictions based on these proofs will be fulfilled, except in the event of hardware failure. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.

Apart from giving an immediate and possibly even provable criterion for the correctness of an implementation, the axiomatic technique for the definition of programming

المنارة للاستشارات

language semantics appears to be like the formal syntax of the ALGOL 60 report, in that it is sufficiently simple to be understood both by the implementor and by the reasonably sophisticated user of the language. It is only by bridging this widening communication gap in a single document (perhaps even provably consistent) that the maximum advantage can be obtained from a formal language definition.

Another of the great advantages of using an axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined*, for example, range of integers, accuracy of floating point, and choice of overflow technique. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs. Thus a programming language standard should consist of a set of axioms of universal applicability, together with a choice from a set of supplementary axioms describing the range of choices facing an implementor. An example of the use of axioms for this purpose was given in Section 2.

Another of the objectives of formal language definition is to assist in the design of better programming languages. The regularity, clarity, and ease of implementation of the ALGOL 60 syntax may at least in part be due to the use of an elegant formal technique for its definition. The use of axioms may lead to similar advantages in the area of "semantics," since it seems likely that a language which can be described by a few "self-evident" axioms from which proofs will be relatively easy to construct will be preferable to a language with many obscure axioms which are difficult to apply in proofs. Furthermore, axioms enable the language designer to express his general *intentions* quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions. Finally, ax-

المنارة للاستشارات

ioms can be formulated in a manner largely independent of each other, so that the designer can work freely on one axiom or group of axioms without fear of unexpected interaction effects with other parts of the language.

Acknowledgments. Many axiomatic treatments of computer programming [1, 2, 3] tackle the problem of proving the equivalence, rather than the correctness, of algorithms. Other approaches [4, 5] take recursive functions rather than programs as a starting point for the theory. The suggestion to use axioms for defining the primitive operations of a computer appears in [6, 7]. The importance of program proofs is clearly emphasized in [9], and an informal technique for providing them is described. The suggestion that the specification of proof techniques provides an adequate formal definition of a programming language first appears in [8]. The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be: (1) a suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined; (2) a comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for formal language definition.

However, the formal material presented here has only an expository status and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by others.

RECEIVED NOVEMBER, 1968; REVISED MAY, 1969

المنارة للاستشارات

REFERENCES

1. YANOV, YU I. Logical operator schemes. *Kybernetika 1*, (1958).
2. IGARASHI, S. An axiomatic approach to equivalence problems of algorithms with applications. Ph.D. Thesis 1964. Rep. Compt. Centre, U. Tokyo, 1968, pp. 1–101.
3. DE BAKKER, J. W. Axiomatics of simple assignment statements. M.R. 94, Mathematisch Centrum, Amsterdam, June 1968.
4. MCCARTHY, J. Towards a mathematical theory of computation. Proc. IFIP Cong. 1962, North Holland Pub. Co., Amsterdam, 1963.
5. BURSTALL, R. Proving properties of programs by structural induction. Experimental Programming Reports: No. 17 DMIP, Edinburgh, Feb. 1968.
6. VAN WIJNGAARDEN, A. Numerical analysis as an independent science. *BIT 6* (1966), 66–81.
7. LASKI, J. Sets and other types. ALGOL Bull. 27, 1968.
8. FLOYD, R. W. Assigning meanings to programs. Proc. Amer. Math. Soc. Symposia in Applied Mathematics, Vol. 19, pp. 19–31.
9. NAUR, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310–316.

المنارة للاستشارات

C.A.R. Hoare

Proof of Correctness of Data Representations

Acta Informatica, Vol. 1, Fasc. 4, 1972
pp. 271–281

المنارة للاستشارات

Acta Informatica 1, 271—281 (1972)
© by Springer-Verlag 1972

Proof of Correctness of Data Representations

C. A. R. Hoare

Received February 16, 1972

Summary. A powerful method of simplifying the proofs of program correctness is suggested; and some new light is shed on the problem of functions with side-effects.

1. Introduction

In the development of programs by stepwise refinement [1–4], the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" program. A similar suggestion was made more formally in algebraic terms in [5], which gives a general definition of simulation. However, a more restricted definition may prove to be more useful in practical program proofs.

If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorising it in this way. Furthermore, the two parts of the proof correspond to the successive stages in program development, thereby contributing to a constructive approach to the correctness of programs [6]. Finally, it must be recalled that in the case of larger and more complex programs the description given above in terms of two stages readily generalises to multiple stages.

2. Concepts and Notations

Suppose in an abstract program there is some abstract variable t which is regarded as being of type T (say a small set of integers). A concrete representation of t will usually consist of several variables $c_1, c_2, \ldots, c_n$ whose types are directly (or more directly) represented in the computer store. The primitive operations on the variable t are represented by procedures $p_1, p_2, \ldots, p_m$, whose bodies carry out on the variables $c_1, c_2, \ldots, c_n$ a series of operations directly (or more directly) performed by computer hardware, and which correspond to meaningful operations on the abstract variable t. The entire concrete representation of the type T can

المنارة للاستشارات

be expressed by declarations of these variables and procedures. For this we adopt the notation of the SIMULA 67 [7] class declaration, which specifies the association between an abstract type T and its concrete representation:

class T;
    begin ... declarations of $c_1, c_2, \ldots, c_n \ldots$;
        procedure p_1 ⟨formal parameter part⟩; Q_1;
        procedure p_2 ⟨formal parameter part⟩; Q_2;
            . . . .   (1)
        procedure p_m ⟨formal parameter part⟩; Q_m;
        Q
    end;

where Q is a piece of program which assigns initial values (if desired) to the variables $c_1, c_2, \ldots, c_n$. As in ALGOL 60, any of the p's may be functions; this is signified by preceding the procedure declaration by the type of the procedure.

Having declared a representation for a type T, it will be required to use this in the abstract program to declare all variables which are to be represented in that way. For this purpose we use the notation:

var (T) t;

or for multiple declarations:

var (T) $t_1, t_2, \ldots$;

The same notation may be used for specifying the types of arrays, functions, and parameters. Within the block in which these declarations are made, it will be required to operate upon the variables $t, t_1, \ldots,$ in the manner defined by the bodies of the procedures $p_1, p_2, \ldots, p_m$. This is accomplished by introducing a compound notation for a procedure call:

$$t_i \cdot p_j \text{ ⟨actual parameter part⟩};$$

where t_i names the variable to be operated upon and p_j names the operation to be performed.

If p_j is a function, the notation displayed above is a function designator; otherwise it is a procedure statement. The form $t_i \cdot p_j$ is known as a *compound identifier*.

These concepts and notations have been closely modelled on those of SIMULA 67. The only difference is the use of **var** (T) instead of **ref** (T). This reflects the fact that in the current treatment, objects of declared classes are not expected to be addressed by reference; usually they will occupy storage space contiguously in the local workspace of the block in which they are declared, and will be addressed by offset in the same way as normal integer and real variables of the block.

3. Example

As an example of the use of these concepts, consider an abstract program which operates on several small sets of integers. It is known that none of these sets ever has more than a hundred members. Furthermore, the only operations

المنارة للاستشارات

actually used in the abstract program are the initial clearing of the set, and the insertion and removal of individual members of the set. These are denoted by procedure statements

$$s \cdot \text{insert}(i)$$

and

$$s \cdot \text{remove}(i).$$

There is also a function "$s \cdot \text{has}(i)$", which tests whether i is a member of s.

It is decided to represent each set as an array A of 100 integer elements, together with a pointer m to the last member of the set; m is zero when the set is empty. This representation can be declared:

```
class smallintset;
begin integer m; integer array A [1:100];

        procedure insert (i); integer i;
          begin integer j;
            for j:=1 step 1 until m do
              if A [j] =i then go to end insert;
              m:=m+1;
              A [m]:=i;

  end insert: end insert;

        procedure remove (i); integer i;
          begin integer j, k;
            for j:=1 step 1 until m do
              if A [j] =i then
                begin for k:=j+1 step 1 until m do A [k-1]:=A [k];
                  comment close the gap over the removed member;
                  m:=m-1;
                  go to end remove
                end;

end remove: end remove;

        Boolean procedure has (i); integer i;
          begin integer j;
              has:=false;
              for j:=1 step 1 until m do
                if A [j] =i then
                  begin has:=true; go to end contains end;

end contains: end contains;

    m:=0; comment initialise set to empty;

end smallintset;
```

Note: as in SIMULA 67, simple variable parameters are presumed to be called by value.

المنارة للاستشارات

4. Semantics and Implementation

The meaning of class declarations and calls on their constituent procedures may be readily explained by textual substitution; this also gives a useful clue to a practical and efficient method of implementation. A declaration:

$$\textbf{var}\,(T)\,t;$$

is regarded as equivalent to the unbracketed body of the class declaration with **begin** ... **end** brackets removed, after every occurrence of an identifier c_i or p_i declared in it has been prefixed by "$t\cdot$". If there are any initialising statements in the class declaration these are removed and inserted just in front of the compound tail of the block in which the declaration is made. Thus if T has the form displayed in (1), **var** $(T)t$ is equivalent to:

... declarations for $t\cdot c_1, t\cdot c_2, \ldots, t\cdot c_n \ldots$;
procedure $t\cdot p_1(\ldots); Q_1'$;
procedure $t\cdot p_2(\ldots); Q_2'$;
....
procedure $t\cdot p_m(\ldots); Q_m'$;

where $Q_1', Q_2', \ldots, Q_m', Q'$ are obtained from $Q_1, Q_2, \ldots, Q_m, Q$ by prefixing every occurrence of $c_1, c_2, \ldots, c_n, p_1, p_2, \ldots, p_m$ by "$t\cdot$". Furthermore, the initialising statement Q' will have been inserted just ahead of the statements of the block body.

If there are several variables of class T declared in the same block, the method described above can be applied to each of them. But in a practical implementation, only one copy of the procedure bodies will be translated. This would contain as an extra parameter an address to the block of $c_1, c_2, \ldots, c_n$ on which a particular call is to operate.

5. Criterion of Correctness

In an abstract program, an operation of the form

$$t_i \cdot p_j(a_1, a_2, \ldots, a_{n_j}) \tag{2}$$

will be expected to carry out some transformation on the variable t_i, in such a way that its resulting value is $f_j(t_i, a_1, a_2, \ldots, a_{n_j})$, where f_j is some primitive operation required by the abstract program. In other words the procedure statement is expected to be equivalent to the assignment

$$t_i := f_j(t_i, a_1, a_2, \ldots, a_{n_j});$$

When this equivalence holds, we say that p_j *models* f_j. A similar concept of modelling applies to functions. It is desired that the proof of the abstract program may be based on the equivalence, using the rule of assignment [8], so that for any propositional formula S, the abstract programmer may assume:

$$S^{t_i}_{f_j(t_i, a_1, a_2, \ldots, a_{n_j})}\{t_i \cdot p_j(a_1, a_2, \ldots, a_{n_j})\}S.^{1}$$

---

1 S^x_y stands for the result of replacing all free occurrences of x in S by y: if any free variables of y would become bound in S by this substitution, this is avoided by preliminary systematic alteration of bound variables in S.

المنارة للاستشارات

In addition, the abstract programmer will wish to assume that all declared variables are initialised to some designated value d_0 of the abstract space.

The criterion of correctness of a data representation is that every p_j models the intended f_j and that the initialisation statement "models" the desired initial value; and consequently, a program operating on abstract variables may validly be replaced by one carrying out equivalent operations on the concrete representation.

Thus in the case of smallintset, we require to prove that:

$$\begin{aligned} &\textbf{var}\,(i)\,t \text{ initialises } t \text{ to } \{\ \} \text{ (the empty set)} \\ &t \cdot \text{insert } (i) \equiv t := t \cup \{i\} \\ &t \cdot \text{remove}\,(i) \equiv t := t \cap \neg\{i\} \\ &t \cdot \text{has}\,(i) \equiv i \in t. \end{aligned} \tag{3}$$

6. Proof Method

The first requirement for the proof is to define the relationship between the abstract space in which the abstract program is written, and the space of the concrete representation. This can be accomplished by giving a function $\mathscr{A}(c_1, c_2, \ldots, c_n)$ which maps the concrete variables into the abstract object which they represent. For example, in the case of smallintset, the representation function can be defined as

$$\mathscr{A}(m, A) = \{i \colon \text{integer} \,|\, \exists k\, (1 \leqq k \leqq m \;\&\; A\,[k] = i)\} \tag{4}$$

or in words, "(m, A) represents the set of values of the first m elements of A". Note that in this and in many other cases $\mathscr{A}$ will be a many-one function. Thus there is no unique concrete value representing any abstract one.

Let t stand for the value of $\mathscr{A}(c_1, c_2, \ldots, c_m)$ before execution of the body Q_j of procedure p_j. Then what we must prove is that after execution of Q_j the following relation holds:

$$\mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{n_j})$$

where $v_1, v_2, \ldots, v_{n_j}$ are the formal parameters of p_j.

Using the notations of [8], the requirement for proof may be expressed:

$$t = \mathscr{A}(c_1, c_2, \ldots, c_n)\ \{Q_j\}\ \mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{n_j})$$

where t is a variable which does not occur in Q_j. On the basis of this we may say: $t \cdot p_j(a_1, a_2, \ldots, a_n) \equiv t := f_j(t, a_1, a_2, \ldots, a_n)$ with respect to $\mathscr{A}$. This deduction depends on the fact that no Q_j alters or accesses any variables other than $c_1, c_2, \ldots, c_n$; we shall in future assume that this constraint has been observed.

In fact for practical proofs we need a slightly stronger rule, which enables the programmer to give an invariant condition $I(c_1, c_2, \ldots, c_n)$, defining some relationship between the constituent concrete variables, and thus placing a constraint on the possible combinations of values which they may take. Each operation (except initialisation) may assume that I is true when it is first entered; and each operation must in return ensure that it is true on completion.

المنارة للاستشارات

In the case of smallintset, the correctness of all operations depends on the fact that m remains within the bounds of A, and the correctness of the remove operation is dependent on the fact that the values of $A[1], A[2], \ldots, A[m]$ are all different; a simple expression of this invariant is:

$$\text{size}\left(\mathscr{A}(m, A)\right) = m \leqq 100. \qquad (I)$$

One additional complexity will often be required; in general, a procedure body is not prepared to accept arbitrary combinations of values for its parameters, and its correctness therefore depends on satisfaction of some precondition $P(t, a_1, a_2, \ldots, a_n)$ before the procedure is entered. For example, the correctness of the insert procedure depends on the fact that the size of the resulting set is not greater than 100, that is

$$\text{size}\,(t \cup \{i\}) \leqq 100$$

This precondition (with t replaced by $\mathscr{A}$) may be assumed in the proof of the body of the procedure; but it must accordingly be proved to hold before every call of the procedure.

It is interesting to note that any of the p's that are functions may be permitted to change the values of the c's, on condition that it preserves the truth of the invariant, and also that it preserves unchanged the value of the abstract object $\mathscr{A}$. For example, the function "has" could reorder the elements of A; this might be an advantage if it is expected that membership of some of the members of the set will be tested much more frequently than others. The existence of such a concrete side-effect is wholly invisible to the abstract program. This seems to be a convincing explanation of the phenomenon of "benevolent side-effects", whose existence I was not prepared to admit in [8].

7. Proof of Smallintset

The proof may be split into four parts, corresponding to the four parts of the class declaration:

7.1. Initialisation

What we must prove is that after initialisation the abstract set is empty and that the invariant I is true:

$$\textbf{true}\ \{m := 0\}\ \{i \mid \exists k (1 \leqq k \leqq m \,\&\, A[k] = i)\} = \{\,\}$$
$$\&\ \text{size}\left(\mathscr{A}(m, a)\right) = m \leqq 100$$

Using the rule of assignment, this depends on the obvious truth of the lemma

$$\{i \mid \exists k (1 \leqq k \leqq 0 \,\&\, A[k] = i\} = \{\,\} \,\&\ \text{size}\,(\{\,\}) = 0 \leqq 100$$

7.2. Has

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I\, \{Q_{\text{has}}\}\, \mathscr{A}(m, A) = k \,\&\, I \,\&\ \text{has} = i \in \mathscr{A}(m, A)$$

المنارة للاستشارات

where Q_{has} is the body of has. Since Q_{has} does not change the value of m or A, the truth of the first two assertions on the right hand side follows directly from their truth beforehand. The invariant of the loop inside Q_{has} is:

$$j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j, A)$$

as may be verified by a proof of the lemma:

$$\begin{aligned} & j < m \,\&\, j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j, A) \\ & \supset \textbf{if } A[j+1] = i \textbf{ then } (\textbf{true} = i \in \mathscr{A}(m, A)) \\ & \qquad\qquad\qquad\qquad \textbf{else } \text{has} = i \in \mathscr{A}(j+1, A). \end{aligned}$$

Since the final value of j is m, the truth of the desired result follows directly from the invariant; and since the "initial" value of j is zero, we only need the obvious lemma

$$\textbf{false} = i \in \mathscr{A}(0, A)$$

7.3. Insert

What we must prove is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I \{Q_{\text{insert}}\} \mathscr{A}(m, A) = (k \cup \{i\}) \,\&\, I,$$

where $P \equiv_{\text{df}} \text{size}\,(\mathscr{A}(m, A) \cup \{i\}) \leqq 100$.

The invariant of the loop is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I \,\&\, i \tilde{\in} \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \qquad (6)$$

as may be verified by the proof of the lemma

$$\begin{aligned} & \mathscr{A}(m, A) = k \,\&\, i \tilde{\in} \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \,\&\, j < m \supset \\ & \qquad \textbf{if } A[j+1] = i \textbf{ then } \mathscr{A}(m, A) = (k \cup \{i\}) \\ & \qquad\qquad\qquad \textbf{else } 0 \leqq j+1 \leqq m \,\&\, i \tilde{\in} \mathscr{A}(j+1, A) \end{aligned}$$

(The invariance of $P \,\&\, \mathscr{A}(m, A) = k \,\&\, I$ follows from the fact that the loop does not change the values of m or A). That (6) is true before the loop follows from $i \tilde{\in} \mathscr{A}(0, A)$.

We must now prove that the truth of (6), together with $j = m$ at the end of the loop, is adequate to ensure the required final condition. This depends on proof of the lemma

$$j = m \,\&\, (6) \cup \mathscr{A}(m+1, A') = (k \cup \{i\}) \,\&\, \text{size}\,(\mathscr{A}(m+1, A')) = m+1 \leqq 100$$

where $A' = (A, m+1 : i)$ is the new value of A after assignment of i to $A[m+1]$.

7.4. Remove

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I \{Q_{\text{remove}}\} \mathscr{A}(m, A) = (k \cap \neg \{i\}) \,\&\, I.$$

The details of the proof are complex. Since they add nothing more to the purpose of this paper, they will be omitted.

المنارة للاستشارات

8. Formalities

Let T be a class declared as shown in Section 2, and let $\mathscr{A}$, I, P_j, f_j be formulae as explained in Section 6 (free variable lists are omitted where convenient). Suppose also that the following $m+1$ theorems have been proved:

$$\textbf{true}\ \{Q\} I \& \mathscr{A} = d_0 \tag{7}$$

$$\mathscr{A} = t \& I \& P_j(t) \{Q_j\} I \& \mathscr{A} = f_j(t) \quad \text{for procedure bodies } Q_j \tag{8}$$

$$\mathscr{A} = t \& I \& P_j(t) \{Q_j\} I \& \mathscr{A} = t \& p_j = f_j(t) \quad \text{for function bodies } Q_j. \tag{9}$$

In this section we show that the proof of these theorems is a sufficient condition for the correctness of the data representation, in the sense explained in Section 5.

Let X be a program beginning with a declaration of a variable t of an abstract type, and initialising it to d_0. The subsequent operations on this variable are of the form

(1) $t := f_j(t, a_1, a_2, \ldots, a_{n_j})$ if Q_j is a procedure

(2) $f_j(t, a_1, a_2, \ldots, a_{n_j})$ if Q_j is a function.

Suppose also that $P_j(t, a_1, a_2, \ldots, a_{n_j})$ has been proved true before each such operation.

Let X' be a program formed from X by replacements described in Section 4, as well as the following (see Section 5):

(1) initialisation $t := d_0$ replaced by Q'

(2) $t := f_j(t, a_1, a_2, \ldots, a_{n_j})$ replaced by $t \cdot p_j(a_1, a_2, \ldots, a_{n_j})$

(4) $f_j(t, a_1, a_2, \ldots, a_{n_j})$ by $t \cdot p_j(a_1, a_2, \ldots, a_{n_j})$.

Theorem. Under conditions described above, if X and X' both terminate, the value of t on termination of X will be $\mathscr{A}(c_1, c_2, \ldots, c_n)$, where $c_1, c_2, \ldots, c_n$ are the values of these variables on termination of X'.

Corollary. If $R(t)$ has been proved true on termination of X, $R(\mathscr{A})$ will be true on termination of X'.

Proof. Consider the sequence S of operations on t executed during the computation of X, and let S' be the sequence of subcomputations of X' arising from execution of the procedure calls which have replaced the corresponding operations on t in X. We will prove that there is a close elementwise correspondence between the two sequences, and that

(a) each item of S' is the very procedure statement which replaced the corresponding operation in S.

(b) the values of all variables (and hence also the actual parameters) which are common to both "programs" are the same after each operation.

(c) the invariant I is true between successive items of S'.

المنارة للاستشارات

(d) if the operations are function calls, their results in both sequences are the same.

(e) and if they are procedure calls (or the initialisation) the value of t immediately after the operation in S is given by $\mathscr{A}$, as applied to the values of c_1, $c_2, \ldots, c_n$ after the corresponding operation in S'.

It is this last fact, applied to the last item of the two sequences, that establishes the truth of the theorem.

The proof is by induction on the position of an item in S.

(1) *Basis*. Consider its first item of S, $t := d_0$. Since X and X' are identical up to this point, the first item of S' must be the subcomputation of the procedure Q which replaced it, proving (a). By (7), I is true after Q in S', and also $\mathscr{A} = d_0$, proving (c) and (e). (d) is not relevant. Q is not allowed to change any non-local variable, proving (b).

(2) *Induction step*. We may assume that conditions (a) to (e) hold immediately after the $(n-1)$-th item of S and S', and we establish that they are true after the n-th. Since the value of all other variables (and the result, if a function) were the same after the previous operation in both sequences, the subsequent course of the computation must also be the same until the very next point at which X' differs from X. This establishes (a) and (b). Since the only permitted changes to the values of $t \cdot c_1, t \cdot c_2, \ldots, t \cdot c_n$ occur in the subcomputations of S', and I contains no other variables, the truth of I after the previous subcomputation proves that it is true before the next. Since S contains *all* operations on t, the value of t is the same before the n-th as it was after the $(n-1)$-th operation, and it is still equal to $\mathscr{A}$. It is given as proved that the appropriate $P_j(t)$ is true before each call of f_j in S. Thus we have established that $\mathscr{A} = t \& I \& P_j(t)$ is true before the operation in S'. From (8) or (9) the truth of (c), (d), (e) follows immediately. (b) follows from the fact that the assignment in S changes the value of no other variable besides t; and similarly, Q_j is not permitted to change the value of any variable other than $t \cdot c_1, t \cdot c_2, \ldots, t \cdot c_n$.

This proof has been an informal demonstration of a fairly obvious theorem. Its main interest has been to show the necessity for certain restrictive conditions placed on class declarations. Fortunately these restrictions are formulated as scope rules, which can be rigorously checked at compile time.

9. Extensions

The exposition of the previous sections deals only with the simplest cases of the Simula 67 class concept; nevertheless, it would seem adequate to cover a wide range of practical data representations. In this section we consider the possibility of further extensions, roughly in order of sophistication.

9.1. Class Parameters

It is often useful to permit a class to have formal parameters which can be replaced by different actual parameters whenever the class is used in a declaration. These parameters may influence the method of representation, or the identity

المنارة للاستشارات

of the initial value, or both. In the case of smallintset, the usefulness of the definition could be enhanced if the maximum size of the set is a parameter, rather than being fixed at 100.

9.2. Dynamic Object Generation

In Simula 67, the value of a variable c of class C may be reinitialised by an assignment:

$$c := \textbf{new}\ C\ \langle\text{actual parameter part}\rangle;$$

This presents no extra difficulty for proofs.

9.3. Remote Identification

In many cases, a local concrete variable of a class has a meaningful interpretation in the abstract space. For example, the variable m of smallintset always stands for the size of the set. If the main program needs to test the size of the set, it would be possible to make this accessible by writing a function

$$\textbf{integer procedure}\ \text{size};\ \text{size} := m;$$

But it would be simpler and more convenient to make the variable more directly accessible by a compound identifier, perhaps by declaring it

$$\textbf{public integer}\ m;$$

The proof technique would specify that

$$m = \text{size}\,(\mathscr{A}(m, A))$$

is part of the invariant of the class.

9.4. Class Concatenation

The basic mechanism for representing sets by arrays can be applied to sets with members of type or class other than just integers. It would therefore be useful to have a method of defining a class "smallset", which can then be used to construct other classes such as "smallrealset" or "smallcarset", where "car" is another class. In SIMULA 67, this effect can be achieved by the class/subclass and virtual mechanisms.

9.5. Recursive Class Declaration

In Simula 67, the parameters of a class, or of a local procedure of the class, and even the local variables of a class, may be declared as belonging to that very same class. This permits the construction of lists and trees, and their processing by recursive procedure activation. In proving the correctness of such a class, it will be necessary to assume the correctness of all "recursive" operations in the proofs of the bodies of the procedures. In the implementation of recursive classes, it will be necessary to represent variables by a null pointer (**none**) or by the *address* of their value, rather than by direct inclusion of space for their

المنارة للاستشارات

values in block workspace of the block to which they are local. The reason for this is that the amount of space occupied by a value of recursively defined type cannot be determined at compile time.

It is worthy of note that the proof-technique recommended above is valid only if the data structure is "well-grounded" in the sense that it is a pure tree, without cycles and without convergence of branches. The restrictions suggested in this paper make it impossible for local variables of a class to be updated except by the body of a procedure local to that very same activation of the class; and I believe that this will effectively prevent the construction of structures which are not well-grounded, provided that assignment is implemented by copying the complete value, not just the address.

I am deeply indebted to Doug Ross and to all authors of referenced works. Indeed, the material of this paper represents little more than my belated understanding and formalisation of their original work.

References

1. Wirth, N.: The development of programs by stepwise refinement. Comm. ACM. **14**, 221–227 (1971).
2. Dijkstra, E. W.: Notes on structured programming. In Structured Programming. Academic Press (1972).
3. Hoare, C. A. R.: Notes on data structuring. *Ibid.*
4. Dahl, O.-J.: Hierachical program structures. *Ibid.*
5. Milner, R.: An algebraic definition of simulation between programs. CS 205 Stanford University, February 1971.
6. Dijkstra, E. W.: A constructive approach to the problem of program correctness. BIT. **8**, 174–186 (1968).
7. Dahl, O.-J., Myhrhaug, B., Nygaard, K.: The SIMULA 67 common base language. Norwegian Computing Center, Oslo, Publication No. S-22, 1970.
8. Hoare, C. A. R.: An axiomatic approach to computer programming. Comm. ACM. **12**, 576–580, 583 (1969).

Prof. C. A. R. Hoare
Computer Science
The Queen's University of Belfast
Belfast BT 71 NN
Northern Ireland

المنارة للاستشارات

Michael Jackson

Constructive Methods of Program Design

Lecture Notes in Computer Science, Vol 44,
ed. by G. Goos and J. Hartmann
Springer-Verlag, Berlin, Heidelberg, New York
pp. 236–262

المنارة للاستشارات

CONSTRUCTIVE METHODS OF PROGRAM DESIGN

M. A. Jackson
Michael Jackson Systems Limited
101 Hamilton Terrace, London NW8

Abstract

Correct programs cannot be obtained by attempts to test or to prove incorrect programs: the correctness of a program should be assured by the design procedure used to build it.

A suggestion for such a design procedure is presented and discussed. The procedure has been developed for use in data processing, and can be effectively taught to most practising programmers. It is based on correspondence between data and program structures, leading to a decomposition of the program into distinct processes. The model of a process is very simple, permitting use of simple techniques of communication, activation and suspension. Some wider implications and future possibilities are also mentioned.

1. Introduction

In this paper I would like to present and discuss what I believe to be a *more constructive method of program design*. The phrase itself is important; I am sure that no-one here will object if I use a LIFO discipline in briefly elucidating its intended meaning.

'Design' is primarily concerned with structure; the designer must say what parts there are to be and how they are to be arranged. The crucial importance of modular programming and structured programming (even in their narrowest and crudest manifestations) is that they provide some definition of what parts are permissible: a module is a separately compiled, parameterised subroutine; a structure component is a sequence, an iteration or a selection. With such definitions, inadequate though they may be, we can at least begin to think about design: what modules should make up that program, and how should they be arranged? should this program be an iteration of selections or a sequence of iterations? Without such definitions, design is meaningless. At the top level of a problem there are P^N possible designs, where P is the number of distinct types of permissible part and N is the number of parts needed to make up the whole. So, to preserve our sanity, both P and N must be small: modular programming, using tree or hierarchical structures, offers small values of N; structured programming offers, additionally, small values of P.

المنارة للاستشارات

'Program' or, rather, 'programming' I would use in a narrow sense. Modelling the problem is 'analysis'; 'programming' is putting the model on a computer. Thus, for example, if we are asked to find a prime number in the range 10^{50} to 10^{60}, we need a number theorist for the analysis; if we are asked to program discounted cash flow, the analysis calls for a financial expert. One of the major ills in data processing stems from uncertainty about this distinction. In mathematical circles the distinction is often ignored altogether, to the detriment, I believe, of our understanding of programming. Programming is about computer programs, not about number theory, or financial planning, or production control.

'Method' is defined in the Shorter OED as a 'procedure for attaining an object'. The crucial word here is 'procedure'. The ultimate method, and the ultimate is doubtless unattainable, is a procedure embodying a precise and correct algorithm. To follow the method we need only execute the algorithm faithfully, and we will be led infallibly to the desired result. To the extent that a putative method falls short of this ideal it is less of a method.

To be 'constructive', a method must itself be decomposed into distinct steps, and correct execution of each step must assure correct execution of the whole method and thus the correctness of its product. The key requirement here is that the correctness of the execution of a step should be largely verifiable without reference to steps not yet executed by the designer. This is the central difficulty in stepwise refinement: we can judge the correctness of a refinement step only by reference to what is yet to come, and hence only by exercising a degree of foresight to which few people can lay claim.

Finally, we must recognise that design methods today are intended for use by human beings: in spite of what was said above about constructive methods, we need, now and for some time to come, a substantial ingredient of intuition and subjectivity. So what is presented below doés not claim to be fully constructive - merely to be 'more constructive'. The reader must supply the other half of the comparison for himself, measuring the claim against the yardstick of his own favoured methods.

2. Basis of the Method

The basis of the method is described, in some detail, in (1). It is appropriate here only to illustrate it by a family of simple example problems.

Example 1

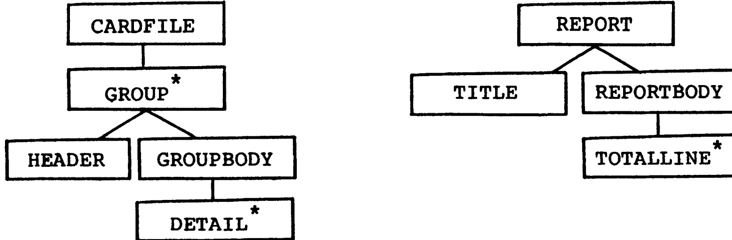
A cardfile of punched cards is sorted into ascending sequence of values of a key which appears in each card. Within this sequence, the first card for each group of cards with a common key value is a header card, while the others are detail cards. Each detail card carries an integer

المنارة للاستشارات

amount. It is required to produce a report showing the totals of amount for all keys.

Solution 1

The first step in applying the method is to describe the structure of the data. We use a graphic notation to represent the structures as trees:-



The above representations are equivalent to the following (in BNF with iteration instead of recursion):

<cardfile> ::= {<group>}$_0^\infty$

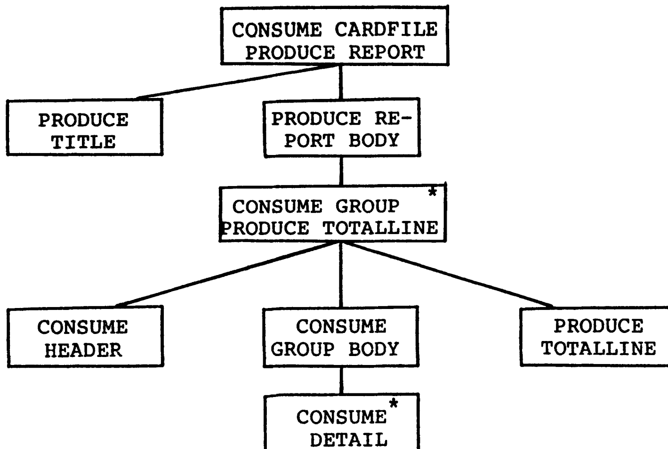
<group> ::= <header><groupbody>

<groupbody> ::= {<detail>}$_0^\infty$

<report> ::= <title><reportbody>

<reportbody> ::= {<totalline>}$_0^\infty$

The second step is to compose these data structures into a program structure:-



This structure has the following properties:

- It is related quite formally to each of the data structures. We may recover any one data structure from the program structure by first marking the leaves corresponding to leaves of

المنارة للاستشارات

the data structure, and then marking all nodes lying in a path from a marked node to the root.

- The correspondences (cardfile : report) and (group : totalline) are determined by the problem statement. One report is derivable from one cardfile; one totalline is derivable from one group, and the totallines are in the same order as the groups.
- The structure is vacuous, in the sense that it contains no executable statements: it is a program which does nothing; it is a tree without real leaves.

The third step in applying the method is to list the executable operations required and to allocate each to its right place in the program structure. The operations are elementary executable statements of the programming language, possibly after enhancement of the language by a bout of bottom-up design; they are enumerated, essentially, by working back from output to input along the obvious data-flow paths. Assuming a reasonably conventional machine and a line printer (rather than a character printer), we may obtain the list:

1. write title
2. write totalline (groupkey, total)
3. total := total + detail.amount
4. total := 0
5. groupkey := header.key
6. open cardfile
7. read cardfile
8. close cardfile

Note that every operation, or almost every operation, must have operands which are data objects. Allocation to a program structure is therefore a trivial task if the program structure is correctly based on the data structures. This triviality is a vital criterion of the success of the first two steps. The resulting program, in an obvious notation, is:

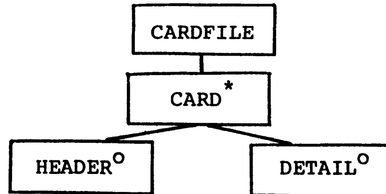
```
CARDFILE-REPORT  sequence
                 open cardfile; read cardfile; write title;
      REPORT-BODY  iteration until cardfile.eof
                   total := 0;  groupkey := header.key;
                   read cardfile;
         GROUP-BODY  iteration until cardfile.eof or
                                     detail.key ≠ groupkey
                     total := total + detail.amount;
                     read cardfile;
         GROUP-BODY  end
                   write totalline (groupkey, total);
      REPORT-BODY  end
                 close cardfile;
CARDFILE-REPORT  end
```

المنارة للاستشارات

Clearly, this program may be transcribed without difficulty into any procedural programming language.

Comment

The solution has proceeded in three steps: first, we defined the data structures; second, we formed them into a program structure; third, we listed and allocated the executable operations. At each step we have criteria for the correctness of the step itself and an implicit check on the correctness of the steps already taken. For example, if at the first step we had wrongly described the structure of cardfile as



(that is: <cardfile> ::= {<card>}$_0^\infty$
<card> ::= <header>|<detail>), we should have been able to see at the first step that we had failed to represent everything we knew about the cardfile. If nonetheless we had persisted in error, we would have discovered it at the second step, when we would have been unable to form a program structure in the absence of a cardfile component corresponding to totalline in report.

The design has throughout concentrated on what we may think of as a static rather than a dynamic view of the problem: on maps, not on itineraries, on structures, not on logic flow. The logic flow of the finished program is a by-product of the data structures and the correct allocation of the 'read' operation. There is an obvious connection between what we have done and the design of a very simple syntax analysis phase in a compiler: the grammar of the input file determines the structure of the program which parses it. We may observe that the 'true' grammar of the cardfile is not context-free: within one group, the header and detail cards must all carry the same key value. It is because the explicit grammar cannot show this that we are forced to introduce the variable groupkey to deal with this stipulation.

Note that there is no error-checking. If we wish to check for errors in the input we must elaborate the structure of the input file to accommodate those errors explicitly. By defining a structure for an input file we define the domain of the program: if we wish to extend the domain, we must extend the input file structure accordingly. In a practical data processing system, we would always define the structure of primary input (such as decks of cards, keyboard messages, etc) to encompass all physically possible files: it would be absurd to construct a program whose operation is unspecified (and therefore, in principle, unpredictable) in the event of a card deck being dropped or a wrong key depressed.

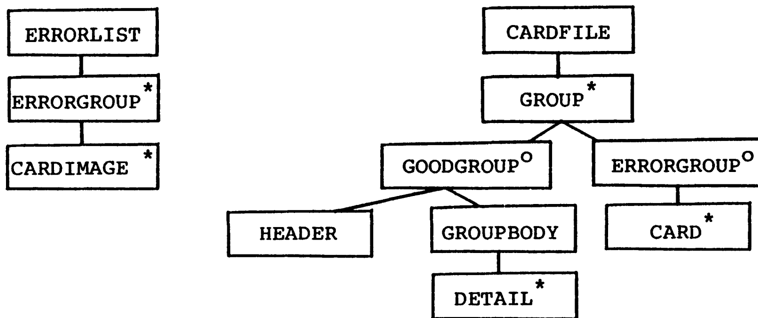
المنارة للاستشارات

Example 2

The cardfile of example 1 is modified so that each card contains a card-type indicator with possible values 'header', 'detail' and other. The program should take account of possible errors in the composition of a group: there may be no header card and/or there may be cards other than detail cards in the group body. Groups containing errors should be listed on an errorlist, but not totalled.

Solution 2

The structure of the report remains unchanged. The structure of the errorlist and of the new version of the cardfile are:



The structure of cardfile demands attention. Firstly, it is ambiguous: anything which is a goodgroup is also an errorgroup. We are forced into this ambiguity because it would be intolerably difficult - and quite unnecessary - to spell out all of the ways in which a group may be in error. The ambiguity is simply resolved by the conventions we use: the parts of a selection are considered to be ordered, and the first applicable part encountered in a left-to-right scan is chosen. So a group can be parsed as an errorgroup only if it has already been rejected as a goodgroup. Secondly, a goodgroup cannot be recognised by a left-to-right parse of the input file with any predetermined degree of lookahead. If we choose to read ahead R records, we may yet encounter a group containing an error only in the R+1'th card.

Recognition problems of this kind occur in many guises. Their essence is that we are forced to a choice during program execution at a time when we lack the evidence on which the choice must be based. Note that the difficulty is not structural but is confined to achieving a workable flow of control. We will call such problems 'backtracking' problems, and tackle them in three stages:-

a Ignore the recognition difficulty, imagining that a friendly demon will tell us infallibly which choice to make. In the present problem, he will tell us whether a group is a goodgroup or an errorgroup. Complete the design procedure in this blissful state of confidence, producing the full program text.

المنارة للاستشارات

b Replace our belief in the demon's infallibility by a sceptical determination to verify each 'landmark' in the data which might prove him wrong. Whenever he is proved wrong we will execute a 'quit' statement which branches to the second part of the selection. These 'quit' statements are introduced into the program text created in stage a.

c Modify the program text resulting from stage b to ensure that side-effects are repealed where necessary.

The.result of stage a, in accordance with the design procedure used for example 1, is:

```
CFILE-REPT-ERR  sequence
                open cardfile;  read cardfile;  write title;
    REPORT-BODY  iteration until cardfile.eof
                 groupkey := card.key;
       GROUP-OUTG  select goodgroup
                   total := 0;
                   read cardfile;
         GOOD-GROUP  iteration until cardfile.eof or
                                     detail.key ≠ groupkey
                     total := total + detail.amount;
                     read cardfile;
         GOOD-GROUP  end
                   write totalline (groupkey, total);
       GROUP-OUTG  or errorgroup
        ERROR-GROUP  iteration until cardfile.eof or
                                     card.key ≠ groupkey
                     write errorline (card);

                     read cardfile;
        ERROR-GROUP  end
       GROUP-OUTG  end
    REPORT-BODY  end
                close cardfile;
CFILE-REPT-ERR  end
```

Note that we cannot completely transcribe this program into any programming language, because we cannot code an evaluable expression for the predicate goodgroup. However, we can readily verify the correctness of the program (assuming the infallibility of the demon). Indeed, if we are prepared to exert ourselves to punch an identifying character into the header card of each goodgroup - thus acting as our own demon - we can code and run the program as an informal demonstration of its acceptability.

We are now ready to proceed to stage b, in which we insert 'quit' statements into the first part of the selection GROUP-OUTG. Also, since quit statements are not present in a normal selection, we will replace the

المنارة للاستشارات

words 'select' and 'or' by 'posit' and 'admit' respectively, thus indicating the tentative nature of the initial choice. Clearly, the landmarks to be checked are the card-type indicators in the header and detail cards. We thus obtain the following program:

```
CFILE-REPT-ERR  sequence
                open cardfile; read cardfile; write title;
    REPORT-BODY  iteration until cardfile.eof
                 groupkey := card.key;
      GROUP-OUTG  posit goodgroup
                  total := 0;
                  quit GROUP-OUTG if card.type ≠ header;
                  read cardfile;
        GOOD-GROUP  iteration until cardfile.eof or
                                    card.key ≠ groupkey
                    quit GROUP-OUTG if card.type ≠ detail;
                    total := total + detail.amount;
                    read cardfile;
        GOOD-GROUP  end
                  write totalline (groupkey, total);
      GROUP-OUTG  admit errorgrou
       ERROR-GROUP  iteration until cardfile.eof or
                                    card.key ≠ groupkey;
                    write errorline (card);
                    read cardfile;
       ERROR-GROUP  end
      GROUP-OUTG  end
    REPORT-BODY  end
                close cardfile;
CFILE-REPT-ERR  end
```

The third stage, stage c, deals with the side-effects of partial execution of the first part of the selection. In this trivial example, the only significant side-effect is the reading of cardfile. In general, it will be found that the only troublesome side-effects are the reading and writing of serial files; the best and easiest way to handle them is to equip ourselves with input and output procedures capable of 'noting' and 'restoring' the state of the file and its associated buffers. Given the availability of such procedures, stage c can be completed by inserting a 'note' statement immediately following the 'posit' statement and a 'restore' statement immediately following the 'admit'. Sometimes side-effects will demand a more ad hoc treatment: when 'note' and 'restore' are unavailable there is no alternative to such cumbersome expedients as explicitly storing each record on disk or in main storage.

المنارة للاستشارات

Comment

By breaking our treatment of the backtracking difficulty into three distinct stages, we are able to isolate distinct aspects of the problem. In stage a we ignore the backtracking difficulty entirely, and concentrate our efforts on obtaining a correct solution to the reduced problem. This solution is carried through the three main design steps, producing a completely specific program text: we are able to satisfy ourselves of the correctness of that text before going on to modify it in the second and third stages. In the second stage we deal only with the recognition difficulty: the difficulty is one of logic flow, and we handle it, appropriately, by modifying the logic flow of the program with quit statements. Each quit statement says, in effect, 'It is supposed (posited) that this is a goodgroup; but if, in fact, this card is not what it ought to be then this is not, after all, a goodgroup'. The required quit statements can be easily seen from the data structure definition, and their place is readily found in the program text because the program structure perfectly matches the data structure. The side-effects arise to be dealt with in stage 3 because of the quit statements inserted in stage b: the quit statements are truly 'go to' statements, producing discontinuities in the context of the computation and hence side-effects. The side-effects are readily identified from the program text resulting from stage b.

Note that it would be quite wrong to distort the data structures and the program structure in an attempt to avoid the dreaded four-letter word 'goto'. The data structures shown, and hence the program structure, are self-evidently the correct structures for the problem as stated: they must not be abandoned because of difficulties with the logic flow.

3. Simple Programs and Complex Programs

The design method, as described above, is severely constrained: it applies to a narrow class of serial file-processing programs. We may go further, and say that it defines such a class - the class of 'simple programs'. A 'simple program' has the following attributes:-

- The program has a fixed initial state; nothing is remembered from one execution to the next.
- Program inputs and outputs are serial files, which we may conveniently suppose to be held on magnetic tapes. There may be more than one input and more than one output file.
- Associated with the program is an explicit definition of the structure of each input and output file. These structures are tree structures, defined in the grammar used above. This grammar permits recursion in addition to the features shown above; it is not very different from a grammar of regular expressions.

المنارة للاستشارات

- The input data structures define the domain of the program, the output data structures its range. Nothing is introduced into the program text which is not associated with the defined data structures.
- The data structures are compatible, in the sense that they can be combined into a program structure in the manner shown above.
- The program structure thus derived from the data structures is sufficient for a workable program. Elementary operations of the program language (possibly supplemented by more powerful or suitable operations resulting from bottom-up design) are allocated to components of the program structure without introducing any further 'program logic'.

A simple program may be designed and constructed with the minimum of difficulty, provided that we adhere rigorously to the design principles adumbrated here and eschew any temptation to pursue efficiency at the cost of distorting the structure. In fact, we should usually discount the benefits of efficiency, reminding ourselves of the mass of error-ridden programs which attest to its dangers.

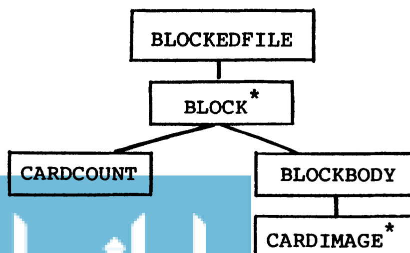
Evidently, not all programs are simple programs. Sometimes we are presented with the task of constructing a program which operates on direct-access rather than on serial files, or which processes a single record at each execution, starting from a varying internal state. As we shall see later, a simple program may be clothed in various disguises which give it a misleading appearance without affecting its underlying nature. More significantly, we may find that the design procedure suggested cannot be applied to the problem given because the data structures are not compatible: that is, we are unable at the second step of the design procedure to form the program structure from the data structures.

Example 3

The input cardfile of example 1 is presented to the program in the form of a blocked file. Each block of this file contains a card count and a number of card images.

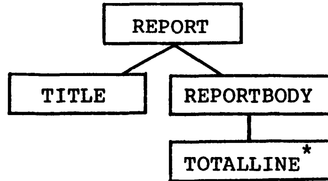
Solution 3

The structure of blockedfile is:



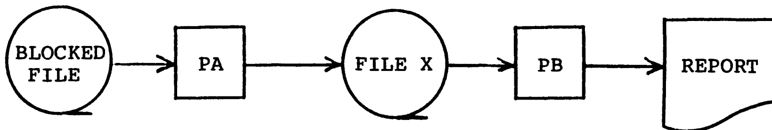
المنارة للاستشارات

This structure does not, of course, show the arrangement of the cards in groups. It is impossible to show, in a single structure, both the arrangement in groups and the arrangement in blocks. But the structure of the report is still:



We cannot fit together the structures of report and blockedfile to form a program structure; nor would we be in better case if we were to ignore the arrangement in blocks. The essence of our difficulty is this: the program must contain operations to be executed once per block, and these must be allocated to a 'process block' component; it must also contain operations to be executed once per group, and these must be allocated to a 'process group' component; but it is impossible to form a single program structure containing both a 'process block' and a 'process group' component. We will call this difficulty a 'structure clash'.

The solution to the structure clash in the present example is obvious: more so because of the order in which the examples have been taken and because everyone knows about blocking and deblocking. But the solution can be derived more formally from the data structures. The clash is of a type we will call 'boundary clash': the boundaries of the blocks are not synchronised with the boundaries of the groups. The standard solution for a structure clash is to abandon the attempt to form a single program structure and instead decompose the problem into two or more simple programs. For a boundary clash the required decomposition is always of the form:



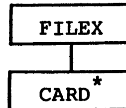
The intermediate file, file X, must be composed of records each of which is a cardimage, because cardimage is the highest common factor of the structures blockedfile and cardfile. The program PB is the program produced as a solution to example 1; the program PA is:

```
PA   sequence
     open blockedfile; open fileX; read blockedfile;
PABODY  iteration until blockedfile.eof
        cardpointer := 1;
```

المنارة للاستشارات

```
      PBLOCK  iteration until cardpointer > block.cardcount
              write cardimage (cardpointer);
              cardpointer := cardpointer + 1;
      PBLOCK  end
            read blockedfile;
    PABODY  end
        close fileX; close blockedfile;
    PA  end
```

The program PB sees file X as having the structure of cardfile in example 1, while program PA sees its structure as:



Comment

The decomposition into two simple programs achieves a perfect solution. Only the program PA is cognisant of the arrangement of cardimages in blocks; only the program PB of their arrangement in groups. The tape containing file X acts as a cordon sanitaire between the two, ensuring that no undesired interactions can occur: we need not concern ourselves at all with such questions as 'what if the header record of a group is the first cardimage in a block with only one cardimage?', or 'what if a group has no detail records and its header is the last cardimage in a block?' in this respect our design is known to be correct.

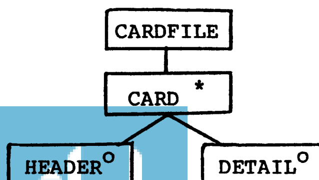
There is an obvious inefficiency in our solution. By introducing the intermediate magnetic tape file we have, to a first approximation, doubled the elapsed time for program execution and increased the program's deman for backing store devices.

Example 4

The input cardfile of example 1 is incompletely sorted. The cards are partially ordered so that the header card of each group precedes any detail cards of that group, but no other ordering is imposed. The report has no title, and the totals may be produced in any order.

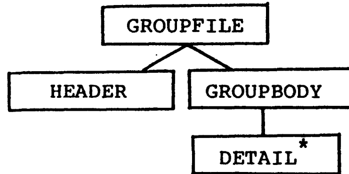
Solution 4

The best we can do for the structure of cardfile is:

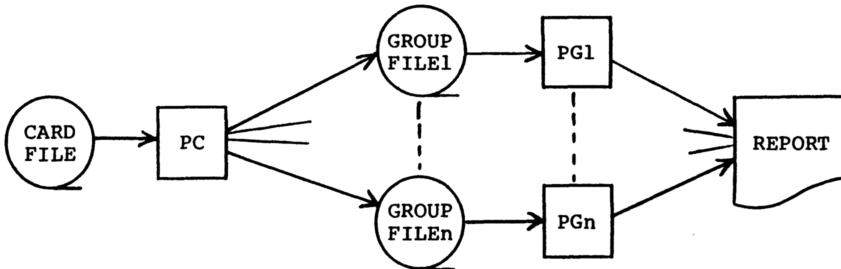


المنارة للاستشارات

which is clearly incompatible with the structure of the report, since there is no component of cardfile corresponding to totalline in the report. Once again we have a structure clash, but this time of a different type. The cardfile consists of a number of groupfiles, each one of which has the form:



The cardfile is an arbitrary interleaving of these groupfiles. To resolve the clash (an 'interleaving clash') we must resolve cardfile into its constituent groupfiles:



Allowing, for purposes of exposition, that a single report may be produced by the n programs PG1, ... PGn (each contributing one totalline), we have decomposed the problem into n+1 simple programs; of these, n are identical programs processing the n distinct groupfiles groupfile1, ... groupfilen; while the other, PC, resolves cardfile into its constituents.

Two possible versions of PC are:

```
PC1  sequence
     open cardfile; read cardfile;
     open all possible groupfiles;
PC1BODY  iteration until cardfile.eof
         write record to groupfile (record.key);
         read cardfile;
PC1BODY  end
     close all possible groupfiles;
     close cardfile;
PC1  end
```

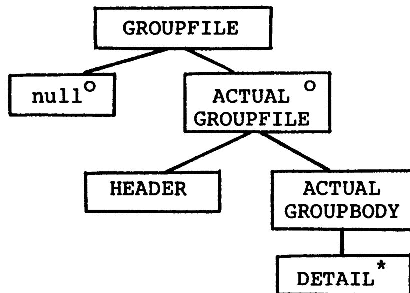
and

```
PC2  sequence
     open cardfile; read cardfile;
```

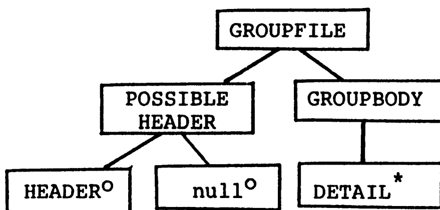
المنارة للاستشارات

```
PC2BODY  iteration until cardfile.eof
 REC-INIT  select new groupfile
           open groupfile (record.key);
 REC-INIT  end
          write record to groupfile (record.key);
          read cardfile;
PC2BODY  end
      close all opened groupfiles;
      close cardfile;
PC2  end
```

Both PC1 and PC2 present difficulties. In PC1 we must provide a groupfile for every possible key value, whether or not cardfile contains records for that key. Also, the programs PG1, ... PGn must be elaborated to handle the null groupfile:



In PC2 we must provide a means of determining whether a groupfile already exists for a given key value. Note that it would be quite wrong to base the determination on the fact that a header must be the first record for a group: such a solution takes impermissible advantage of the structure of groupfile which, in principle, is unknown in the program PC; we would then have to make a drastic change to PC if, for example, the header card were made optional:



Further, in PC2 we must be able to run through all the actual key values in order to close all the groupfiles actually opened. This would still be necessary even if each group had a recognisable trailer record, for reasons similar to those given above concerning the header records.

المنارة للاستشارات

Comment

The inefficiency of our solution to example 4 far outstrips the inefficiency of our solution to example 3. Indeed, our solution to example 4 is entirely impractical. Practical implementation of the designs will be considered below in the next section. For the moment, we may observe that the use of magnetic tapes for communication between simple programs enforces a very healthy discipline. We are led to use a very simple protocol: every serial file must be opened and closed. The physical medium encourages a complete decoupling of the programs: it is easy to imagine one program being run today, the tapes held overnight in a library, and a subsequent program being run tomorrow; the whole of the communication is visible in the defined structure of the files. Finally, we are strengthened in our resolve to think in terms of static structures, avoiding the notoriously error-prone activity of thinking about dynamic flow and execution-time events.

Taking a more global view of the design procedure, we may say that the simple program is a satisfactory high level component. It is a larger object than a sequence, iteration or selection; it has a more precise definition than a module; it is subject to restrictions which reveal to us clearly when we are trying to make a single program out of what should be two or more.

4. Programs, Procedures and Processes

Although from the design point of view we regard magnetic tapes as the canonical medium of communication between simple programs, they will not usually provide a practical implementation.

An obvious possibility for implementation in some environments is to replace each magnetic tape by a limited number of buffers in main storage, with a suitable regime for ensuring that the consumer program does not run ahead of the producer. Each simple program can then be treated as a distinct task or process, using whatever facilities are provided for the management of multiple concurrent tasks.

However, something more like coroutines seems more attractive (2). The standard procedure call mechanism offers a simple implementation of great flexibility and power. Consider the program PA, in our solution to example 3, which writes the intermediate file X. We can readily convert this program into a procedure PAX which has the characteristics of an input procedure for file X. That is, invocations of the procedure PAX will satisfactorily implement the operations 'open file X for reading', 'read file X' and 'close file X after reading'.

We will call this conversion of PA into PAX 'inversion of PA with respect to file X'. (Note that the situation in solution 3 is symmetrical: we could equally well decide to invert PB with respect to file X, obtaining

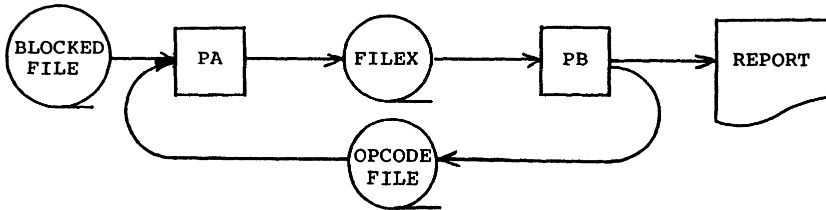
المنارة للاستشارات

an output procedure for file X.) The mechanics of inversion are a mere matter of generating the appropriate object coding from the text of the simple program: there is no need for any modification to that text. PA and PAX are the same program, not two different programs. Most practising programmers seem to be unaware of this identity of PA and PAX, and even those who are familiar with coroutines often program as if they supposed that PA and PAX were distinct things. This is partly due to the baleful influence of the stack as a storage allocation device: we cannot jump out of an inner block of PAX, return to the invoking procedure, and subsequently resume where we left off when we are next invoked. So we must either modify our compiler or modify our coding style, adopting the use of labels and go to statements as a standard in place of the now conventional compound statement of structured programming. It is common to find PAX, or an analogous program, designed as a selection or case statement: the mistake is on all fours with that of the kindergarten child who has been led to believe that the question 'what is 5 multiplied by 3?' is quite different from the question 'what is 3 multiplied by 5?'. At a stroke the poor child has doubled the difficulty of learning the multiplication tables.

The procedure PAX is, of course, a variable state procedure. The value of its state is held in a 'state vector' (or activation record), of which a vital part is the text pointer; the values of special significance are those associated with the suspension of PAX for operations on file X - open, write and close. The state vector is an 'own variable' par excellence, and should be clearly seen as such.

The minimum interface needed between PB and PAX is two parameters: a record of file X, and an additional bit to indicate whether the record is or is not the eof marker. This minimum interface suffices for example 3: there is no need for PB to pass an operation code to PAX (open read or close). It is important to understand that this minimum interface will not suffice for the general case. It is sufficient for example 3 only because the operation code is implicit in the ordering of operations. From the point of view of PAX, the first invocation must be 'open', and subsequent invocations must be 'read' until PAX has returned the eof marker to PB, after which the final invocation must be 'close'. This felicitous harmony is destroyed if, for example, PB is permitted to stop reading and close file X before reaching the eof marker. In such a case the interface must be elaborated with an operation code. Worse, the sequence of values of this operation code now constitutes a file in its own right: the solution becomes:

المنارة للاستشارات



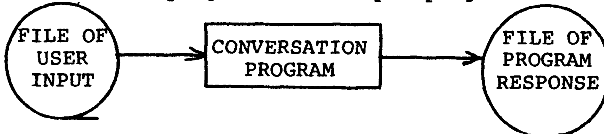
The design of PA is, potentially, considerably more complicated. The benefit we will obtain from treating this complication conscientiously is well worth the price: by making explicit the structure of the opcode file we define the problem exactly and simplify its solution. Failure to recognise the existence of the opcode file, or, just as culpable, failure to make its structure explicit, lies at the root of the errors and obscurities for which manufacturers' input-output software is deservedly infamous.

In solution 4 we created an intolerable multiplicity of files - groupfile1, ... groupfilen. We can rid ourselves of these by inverting the programs PG1, ... PGn with respect to their respective groupfiles: that is, we convert each of the programs PGi to an output procedure PGFi, which can be invoked by PC to execute operations on groupfilei. But we still have an intolerable multiplicity of output procedures, so a further step is required. The procedures are identical except for their names and the current values of their state vectors. So we separate out the pure procedure part - PGF - of which we need keep only one copy, and the named state vectors SVPGF1, ... SVPGFn. We must now provide a mechanism for storing and retrieving these state vectors and for associating the appropriate state vector with each invocation of PGF; many mechanisms are possible, from a fully-fledged direct-access file with serial read facilities to a simple arrangement of the state vectors in an array in main storage.

5. Design and Implementation

The model of a simple program and the decomposition of a problem into simple programs provides some unity of viewpoint. In particular, we may be able to see what is common to programs with widely different implementations. Some illustrations follow.

a A conversational program is a simple program of the form:



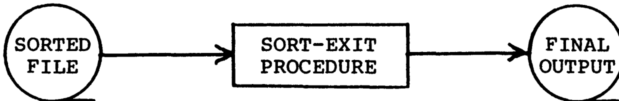
The user provides a serial input file of messages, ordered in time; the conversation program produces a serial file of responses. Inversion of the program with respect to the user in-

المنارة للاستشارات

put file gives an output procedure 'dispose of one message in a conversation'. The state vector of the inverted program must be preserved for the duration of the conversation: IBM's IMS provides the SPA (Scratchpad Area) for precisely this purpose. The conversation program must, of course, be designed and written as a single program: implementation restrictions may dictate segmentation of the object code.

b A 'sort-exit' allows the user of a generalised sorting program to introduce his own procedure at the point where each record is about to be written to the final output file. An interface is provided which permits 'insertion' and 'deletion' of records as well as 'updating'.

We should view the sort-exit procedure as a simple program:



To fit it in with the sorting program we must invert it with respect to both the sortedfile and the finaloutput. The interface must provide an implementation of the basic operations: open sortedfile for reading; read sortedfile (distinguishing the eof marker); close sortedfile after reading; open finaloutput for writing; write finaloutput record; close finaloutput file after writing (including writing the eof marker).

Such concepts as 'insertion' and 'deletion' of records are pointless: at best, they serve the cause of efficiency, traducing clarity; at worst, they create difficulty and confusion where none need exist.

c Our solution to example 1 can be seen as an optimisation of the solution to the more general example 4. By sorting the cardfile we ensure that the groups do not overlap in time: the state vectors of the inverted programs PGF1, ... PGFn can therefore share a single area in main storage. The state vector consists only of the variable total; the variable groupkey is the name of the currently active group and hence of the current state vector. Because the records of a group are contiguous, the end of a group is recognisable at cardfile.eof or at the start of another group. The individual groupfile may therefore be closed, and the totalline written, at the earliest possible moment.

We may, perhaps, generalise so far as to say that an identifier is stored by a program only in order to give a unique name to the state vector of some process.

المنارة للاستشارات

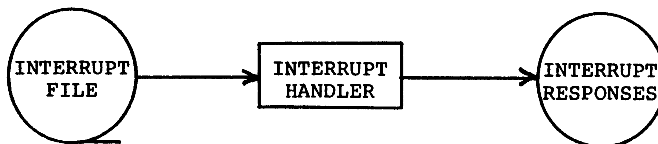
d A data processing system may be viewed as consisting of many simple programs, one for each independent entity in the real world model. By arranging the entities in sets we arrange the corresponding simple programs in equivalence classes. The 'master record' corresponding to an entity is the state vector of the simple program modelling that entity.

The serial files of the system are files of transactions ordered in time: some are primary transactions, communicating with the real world, some are secondary, passing between simple programs of the system. In general, the real world must be modelled as a network of entities or of entity sets; the data processing system is therefore a network of simple programs and transaction files.

Implementation of the system demands decisions in two major areas. First a scheduling algorithm must be decided; second, the representation and handling of state vectors. The extreme cases of the first are 'real-time' and 'serial batch'. In a pure 'real-time' system every primary transaction is dealt with as soon as it arrives, followed immediately by all of the secondary and consequent transactions, until the system as a whole becomes quiet. In a pure 'serial batch' system, each class (identifier set) of primary transactions is accumulated for a period (usually a day, week or month). Each simple program of that class is then activated (if there is a transaction present for it), giving rise to secondary transactions of various classes. These are then treated similarly, and so on until no more transactions remain to be processed.

Choosing a good implementation for a data processing system is difficult, because the network is usually large and many possible choices present themselves. This difficulty is compounded by the long-term nature of the simple programs: a typical entity, and hence a typical program, has a lifetime measured in years or even decades. During such a lifetime the system will inevitably undergo change: in effect, the programs are being rewritten while they are in course of execution.

e An interrupt handler is a program which processes a serial file of interrupts, ordered in time:

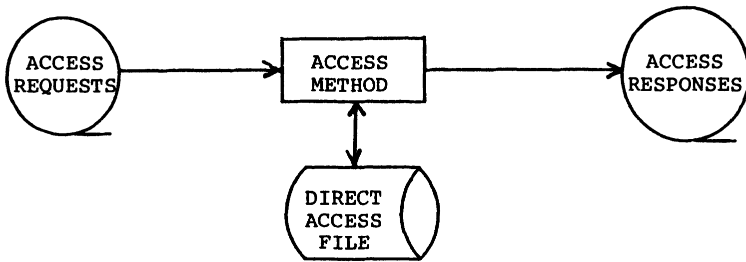


Inversion of the interrupt handler with respect to the interrupt file gives the required procedure 'dispose of one interrupt'. In general, the interrupt file will be composed of in-

المنارة للاستشارات

terleaved files for individual processes, devices, etc. Implementation is further complicated by the special nature of the invocation mechanism, by the fact that the records of the interrupt file are distributed in main storage, special registers and other places, and by the essentially recursive structure of the main interrupt file (unless the interrupt handler is permitted to mask off secondary interrupts).

f An input-output procedure (what IBM literature calls an 'access method') is a simple program which processes an input file of access requests and produces an output file of access responses. An access request consists of an operation code and, sometimes, a data record; an access response consists of a result code and, sometimes, a data record. For example, a direct-access method has the form:

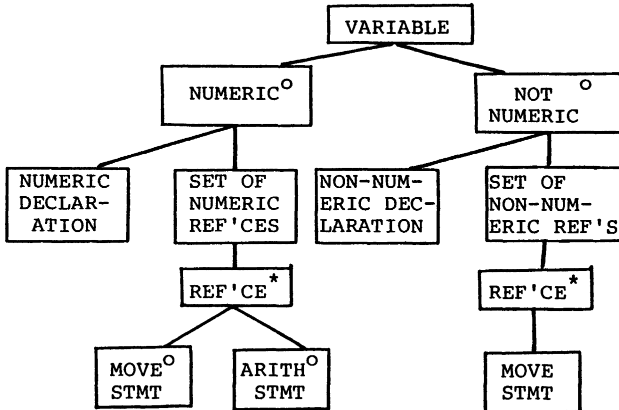


By inverting this simple program with respect to both the file of access requests and the file of access responses we obtain the desired procedure. This double inversion is always possible without difficulty, because each request must produce a response and that response must be calculable before the next request is presented.

The chief crime of access method designers is to conceal from their customers (and, doubtless, from themselves) the structure of the file of access requests. The user of the method is thus unable to determine what sequences of operations are permitted by the access method, and what their effect will be.

g Some aspects of a context-sensitive grammar may be regarded as interleaved context-free grammars. For example, in a grossly simplified version of the COBOL language we may wish to stipulate that any variable may appear as an operand of a MOVE statement, while only a variable declared as numeric may appear as an operand of an arithmetic (ADD, SUBTRACT, MULTIPLY or DIVIDE) statement. We may represent this stipulation as follows:

المنارة للاستشارات



The syntax-checking part of the compiler consists, partly, of a simple program for each declared variable. The symbol table is the set of state vectors for these simple programs. The algorithm for activating and suspending these and other programs will determine the way in which one error interacts with another both for diagnosis and correction.

6. A Modest Proposal

It is one thing to propose a model to illuminate what has already been done, to clarify the sources of existing success or failure. It is quite another to show that the model is of practical value, and that it leads to the construction of acceptable programs. An excessive zeal in decomposition produces cumbersome interfaces and pointlessly redundant code. The "Shanley Principle" in civil engineering (3) requires that several functions be implemented in a single part; this is necessary for economy both in manufacturing and in operating the products of engineering design. It appears that a design approach which depends on decomposition runs counter to this principle: its main impetus is the separation of functions for implementation in distinct parts of the program.

But programs do not have the intractable nature of the physical objects which civil, mechanical or electrical engineers produce. They can be manipulated and transformed (for example, by compilers) in ways which preserve their vital qualities of correctness and modifiability while improving their efficiency both generally and in the specialised environment of a prticular machine. The extent to which a program can be manipulated and transformed is critically affected by two factors: the variety of forms it can take, and the semantic clarity of the text. Programs written using today's conventional techniques score poorly on both factors. There is a distressingly large variety of forms, and intelligibility is compromised or even destroyed by the introduction of

المنارة للاستشارات

implementation-orientated features. The justification for these techniques is, of course, efficiency. But in pursuing efficiency in this way we become caught in a vicious circle: because our languages are rich the compilers cannot understand, and hence cannot optimise, our programs so we need rich languages to allow us to obtain the efficiency which the compilers do not offer.

Decomposition into simple programs, as discussed above, seems to offer some hope of separating the considerations of correctness and modifiability from the considerations of efficiency. Ultimately, the objective is that the first should become largely trivial and the second largely automatic.

The first phase of design would produce the following documents:-

- a definition of each serial file structure for each simple program (including files of operation codes!);
- the text of each simple program;
- a statement of the communication between simple programs, perhaps in the form of identities such as

$$\text{output } (p_i, f_r) \equiv \text{input } (p_j, f_s).$$

It may then be possible to carry out some automatic checking of self-consistency in the design - for instance, to check that the inputs to a program are within its domain. We may observe, incidentally, that the 'inner' feature of Simula 67 (4) is a way of enforcing consistency of a file of operation codes between the consumer and producer processes in a very limited case. More ambitiously, it may be possible, if file-handling protocol is exactly observed, and read and write operations are allocated with a scrupulous regard to principle, to check the correctness of the simple programs in relation to the defined data structures.

In the second phase of design, the designer would specify, in greater or lesser detail:-

- the synchronisation of the simple programs;
- the handling of state vectors;
- the dissection and recombining of programs and state vectors to reduce interface overheads.

Synchronisation is already loosely constrained by the statements of program communication made in the first phase: the consumer can never run ahead of the producer. Within this constraint the designer may choose to impose additional constraints at compile time and/or at execution time. The weakest local constraint is to provide unlimited dynamic buffering at execution time, the consumer being allowed to lag behind the producer by anything from a single record to the whole file, depending on resource allocation elsewhere in the system. The strongest local con-

straints are use of coroutines or program inversion (enforcing a single record lag) and use of a physical magnetic tape (enforcing a whole file lag).

Dissection and recombining of programs becomes possible with coroutines or program inversion; its purpose is to reduce interface overheads by moving code between the invoking and invoked programs, thus avoiding some of the time and space costs of procedure calls and also, under certain circumstances, avoiding replication of program structure and hence of coding for sequencing control. It depends on being able to associate code in one program with code in another through the medium of the communicating data structure.

A trivial illustration is provided by solution 3, in which we chose to invert PA with respect to file X, giving an input procedure PAX for the file of cardimages. We may decide that the procedure call overhead is intolerable, and that we wish to dissect PAX and combine it with PB. This is achieved by taking the invocations of PAX in PB (that is, the statements 'open fileX', 'read fileX' and 'close fileX') and replacing those invocations by the code which PAX would execute in response to them. For example, in response to 'open fileX', PAX would execute the code 'open blockedfile'; therefore the 'open fileX' statement in PB can be replaced by the statement 'open blockedfile'.

A more substantial illustration is provided by the common practice of designers of 'real-time' data processing systems. Suppose that a primary transaction for a product gives rise to a secondary transaction for each open order item for that product, and that each of those in turn gives rise to a transaction for the open order of which it is a part, which then gives rise to a transaction for the customer who placed the order. Instead of having separate simple programs for the product, order item, order and customer, the designer will usually specify a 'transaction processing module': this consists of coding from each of those simple programs, the coding being that required to handle the relevant primary or secondary transaction.

Some interesting program transformations of a possibly relevant kind are discussed in a paper by Burstall and Darlington (5). I cannot end this paper better than by quoting from them:

> "The overall aim of our investigation has been to help people to write correct programs which are easy to alter. To produce such programs it seems advisable to adopt a lucid, mathematical and abstract programming style. If one takes this really seriously, attempting to free one's mind from considerations of computational efficiency, there may be a heavy penalty in program running time; in practice it is often necessary to adopt a more intricate version of the program, sacrificing comprehensibility for speed.

المنارة للاستشارات

The question then arises as to how a lucid program can be transformed into a more intricate but efficient one in a systematic way, or indeed in a way which could be mechanised.

" ... We are interested in starting with programs having an extremely simple structure and only later introducing the complications which we usually take for granted even in high level language programs. These complications arise by introducing useful interactions between what were originally separate parts of the program, benefiting by what might be called 'economies of interaction'."

References

(1) Principles of Program Design; M A Jackson; Academic Press 1975.

(2) Hierarchical Program Structures; O-J Dahl; in Structured Programming; Academic Press 1972.

(3) Structured Programming with go to Statements; Donald E Knuth; in ACM Computing Surveys Vol 6 No 4 December 1974.

(4) A Structural Approach to Protection; C A R Hoare; 1975.

(5) Some Transformations for Developing Recursive Programs; R M Burstall & John Darlington; in Proceedings of 1975 Conference on Reliable Software; Sigplan Notices Vol 10 No 6 June 1975.

المنارة للاستشارات

David L. Parnas

On the Criteria to Be Used in Decomposing Systems into Modules

Communications of the ACM, Vol. 15 (12), 1972
pp. 1053–1058

المنارة للاستشارات

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will

---

Copyright © 1972, Association for Computing Machinery, Inc.
General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.
Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

المنارة للاستشارات

be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design

CR Categories: 4.0

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

المنارة للاستشارات

A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

Expected Benefits of Modular Programming

The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication: (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

What Is Modularization?

Below are several partial system descriptions called *modularizations*. In this context "module" is considered to be a responsibility assignment rather than a sub-

المنارة للاستشارات

program. The *modularizations* include the design decisions which must be made *before* the work on independent modules can begin. Quite different decisions are included for each alternative, but in all cases the intention is to describe all "system level" decisions (i.e. decisions which affect more than one module).

Example System 1: A KWIC Index Production System

The following description of a KWIC index will suffice for this paper. The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

This is a small system. Except under extreme circumstances (huge data base, no supporting software), such a system could be produced by a good programmer within a week or two. Consequently, none of the difficulties motivating modular programming are important for this system. Because it is impractical to treat a large system thoroughly, we must go through the exercise of treating this problem as if it were a large project. We give one modularization which typifies current approaches, and another which has been used successfully in undergraduate class projects.

Modularization 1

We see the following modules:

Module 1: Input. This module reads the data lines from the input medium and stores them in core for

المنارة للاستشارات

processing by the remaining modules. The characters are packed four to a word, and an otherwise unused character is used to indicate the end of a word. An index is kept to show the starting address of each line.

Module 2: Circular Shift. This module is called after the input module has completed its work. It prepares an index which gives the address of the first character of each circular shift, and the original index of the line in the array made up by module 1. It leaves its output in core with words in pairs (original line number, starting address).

Module 3: Alphabetizing. This module takes as input the arrays produced by modules 1 and 2. It produces an array in the same format as that produced by module 2. In this case, however, the circular shifts are listed in another order (alphabetically).

Module 4: Output. Using the arrays produced by module 3 and module 1, this module produces a nicely formatted output listing all of the circular shifts. In a sophisticated system the actual start of each line will be marked, pointers to further information may be inserted, and the start of the circular shift may actually not be the first word in the line, etc.

Module 5: Master Control. This module does little more than control the sequencing among the other four modules. It may also handle error messages, space allocation, etc.

It should be clear that the above does not constitute a definitive document. Much more information would have to be supplied before work could start. The defining documents would include a number of pictures showing core formats, pointer conventions, calling

المنارة للاستشارات

conventions, etc. All of the interfaces between the four modules must be specified before work could begin.

This is a modularization in the sense meant by all proponents of modular programming. The system is divided into a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition which would be proposed by most programmers for the task specified.

Modularization 2

We see the following modules:

Module 1: Line Storage. This module consists of a number of functions or subroutines which provide the means by which the user of the module may call on it. The function call *CHAR*(*r*,*w*,*c*) will have as value an integer representing the *c*th character in the *r*th line, *w*th word. A call such as *SETCHAR*(*r*,*w*,*c*,*d*) will cause the *c*th character in the *w*th word of the *r*th line to be the character represented by *d* (i.e. *CHAR*(*r*,*w*,*c*) = *d*). *WORDS*(*r*) returns as value the number of words in line *r*. There are certain restrictions in the way that these routines may be called; if these restrictions are violated the routines "trap" to an error-handling subroutine which is to be provided by the users of the routine. Additional routines are available which reveal to the caller the number of words in any line, the number of lines currently stored, and the number of characters in any word. Functions *DELINE* and *DELWRD* are provided to delete portions of lines which have already been stored. A precise specification of a similar module

المنارة للاستشارات

has been given in [3] and [8] and we will not repeat it here.

Module 2: INPUT. This module reads the original lines from the input media and calls the line storage module to have them stored internally.

Module 3: Circular Shifter. The principal functions provided by this module are analogs of functions provided in module 1. The module creates the impression that we have created a line holder containing not all of the lines but all of the circular shifts of the lines. Thus the function call *CSCHAR*(l,w,c) provides the value representing the cth character in the wth word of the lth circular shift. It is specified that (1) if $i < j$ then the shifts of line i precede the shifts of line j, and (2) for each line the first shift is the original line, the second shift is obtained by making a one-word rotation to the first shift, etc. A function *CSSETUP* is provided which must be called before the other functions have their specified values. For a more precise specification of such a module see [8].

Module 4: Alphabetizer. This module consists principally of two functions. One, *ALPH*, must be called before the other will have a defined value. The second, *ITH*, will serve as an index. *ITH*(i) will give the index of the circular shift which comes ith in the alphabetical ordering. Formal definitions of these functions are given [8].

Module 5: Output. This module will give the desired printing of set of lines or circular shifts.

Module 6: Master Control. Similar in function to the modularization above.

المنارة للاستشارات

Comparison of the Two Modularizations

General. Both schemes will work. The first is quite conventional; the second has been used successfully in a class project [7]. Both will reduce the programming to the relatively independent programming of a number of small, manageable, programs.

Note first that the two decompositions may share all data representations and access methods. Our discussion is about two different ways of cutting up what *may* be the same object. A system built according to decomposition 1 could conceivably be identical *after assembly* to one built according to decomposition 2. The differences between the two alternatives are in the way that they are divided into the work assignments, and the interfaces between modules. The algorithms used in both cases *might* be identical. The systems are substantially different even if identical in the runnable representation. This is possible because the runnable representation need only be used for running; other representations are used for changing, documenting, understanding, etc. The two systems will not be identical in those other representations.

Changeability. There are a number of design decisions which are questionable and likely to change under many circumstances. This is a partial list.

1. Input format.
2. The decision to have all lines stored in core. For large jobs it may prove inconvenient or impractical to keep all of the lines in core at any one time.
3. The decision to pack the characters four to a word. In cases where we are working with small amounts of data it may prove undesirable to pack the characters;

المنارة للاستشارات

time will be saved by a character per word layout. In other cases we may pack, but in different formats.

4. The decision to make an index for the circular shifts rather that actually store them as such. Again, for a small index or a large core, writing them out may be the preferable approach. Alternatively, we may choose to prepare nothing during *CSSETUP*. All computation could be done during the calls on the other functions such as *CSCHAR*.

5. The decision to alphabetize the list once, rather than either (a) search for each item when needed, or (b) partially alphabetize as is done in Hoare's FIND [2]. In a number of circumstances it would be advantageous to distribute the computation involved in alphabetization over the time required to produce the index.

By looking at these changes we can see the differences between the two modularizations. The first change is confined to one module in both decompositions. For the first decomposition the second change would result in changes in every module! The same is true of the third change. In the first decomposition the format of the line storage in core must be used by all of the programs. In the second decomposition the story is entirely different. Knowledge of the exact way that the lines are stored is entirely hidden from all but module 1. Any change in the manner of storage can be confined to that module!

In some versions of this system there was an additional module in the decomposition. A symbol table module (as specified in [3]) was used within the line storage module. This fact was completely invisible to the rest of the system.

المنارة للاستشارات

The fourth change is confined to the circular shift module in the second decomposition, but in the first decomposition the alphabetizer and the output routines will also know of the change.

The fifth change will also prove difficult in the first decomposition. The output module will expect the index to have been completed before it began. The alphabetizer module in the second decomposition was designed so that a user could not detect when the alphabetization was actually done. No other module need be changed.

Independent Development. In the first modularization the interfaces between the modules are the fairly complex formats and table organizations described above. These represent design decisions which cannot be taken lightly. The table structure and organization are essential to the efficiency of the various modules and must be designed carefully. The development of those formats will be a major part of the module development and that part must be a joint effort among the several development groups. In the second modularization the interfaces are more abstract; they consist primarily in the function names and the numbers and types of the parameters. These are relatively simple decisions and the independent development of modules should begin much earlier.

Comprehensibility. To understand the output module in the first modularization, it will be necessary to understand something of the alphabetizer, the circular shifter, and the input module. There will be aspects of the tables used by output which will only make sense because of the way that the other modules work. There will be constraints on the structure of the tables due to

المنارة للاستشارات

the algorithms used in the other modules. The system will only be comprehensible as a whole. It is my subjective judgment that this is not true in the second modularization.

The Criteria

Many readers will now see what criteria were used in each decomposition. In the first decomposition the criterion used was to make each major step in the processing a module. One might say that to get the first decomposition one makes a flowchart. This is the most common approach to decomposition or modularization. It is an outgrowth of all programmer training which teaches us that we should begin with a rough flowchart and move from there to a detailed implementation. The flowchart was a useful abstraction for systems with on the order of 5,000–10,000 instructions, but as we move beyond that it does not appear to be sufficient; something additional is needed.

The second decomposition was made using "information hiding" [4] as a criterion. The modules no longer correspond to steps in the processing. The line storage module, for example, is used in almost every action by the system. Alphabetization may or may not correspond to a phase in the processing according to the method used. Similarly, circular shift might, in some circumstances, not make any table at all but calculate each character as demanded. Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

المنارة للاستشارات

Improvement in Circular Shift Module

To illustrate the impact of such a criterion let us take a closer look at the design of the circular shift module from the second decomposition. Hindsight now suggests that this definition reveals more information than necessary. While we carefully hid the method of storing or calculating the list of circular shifts, we specified an order to that list. Programs could be effectively written if we specified only (1) that the lines indicated in circular shift's current definition will all exist in the table, (2) that no one of them would be included twice, and (3) that an additional function existed which would allow us to identify the original line given the shift. By prescribing the order for the shifts we have given more information than necessary and so unnecessarily restricted the class of systems that we can build without changing the definitions. For example, we have not allowed for a system in which the circular shifts were produced in alphabetical order, *ALPH* is empty, and *ITH* simply returns its argument as a value. Our failure to do this in constructing the systems with the second decomposition must clearly be classified as a design error.

In addition to the general criteria that each module hides some design decision from the rest of the system, we can mention some specific examples of decompositions which seem advisable.

1. A *data structure*, its internal linkings, *accessing procedures and modifying procedures* are part of a single module. They are not shared by many modules as is conventionally done. This notion is perhaps just an elaboration of the assumptions behind the papers of

المنارة للاستشارات

Balzer [9] and Mealy [10]. Design with this in mind is clearly behind the design of BLISS [11].

2. *The sequence of instructions necessary to call a given routine and the routine itself are part of the same module.* This rule was not relevant in the Fortran systems used for experimentation but it becomes essential for systems constructed in an assembly language. There are no perfect general calling sequences for real machines and consequently they tend to vary as we continue our search for the ideal sequence. By assigning responsibility for generating the call to the person responsible for the routine we make such improvements easier and also make it more feasible to have several distinct sequences in the same software structure.

3. The *formats of control blocks* used in queues in operating systems and similar programs *must be hidden* within a "control block module." It is conventional to make such formats the interfaces between various modules. Because design evolution forces frequent changes on control block formats such a decision often proves extremely costly.

4. *Character codes, alphabetic orderings, and similar data should be hidden* in a module for greatest flexibility.

5. The sequence in which certain items will be processed should (as far as practical) be hidden within a single module. Various changes ranging from equipment additions to unavailability of certain resources in an operating system make sequencing extremely variable.

Efficiency and Implementation

If we are not careful the second decomposition will prove to be much less efficient than the first. If each of

المنارة للاستشارات

the functions is actually implemented as a procedure with an elaborate calling sequence there will be a great deal of such calling due to the repeated switching between modules. The first decomposition will not suffer from this problem because there is relatively infrequent transfer of control between modules.

To save the procedure call overhead, yet gain the advantages that we have seen above, we must implement these modules in an unusual way. In many cases the routines will be best inserted into the code by an assembler; in other cases, highly specialized and efficient transfers would be inserted. To successfully and efficiently make use of the second type of decomposition will require a tool by means of which programs may be written as if the functions were subroutines, but assembled by whatever implementation is appropriate. If such a technique is used, the separation between modules may not be clear in the final code. For that reason additional program modification features would also be useful. In other words, the several representations of the program (which were mentioned earlier) must be maintained in the machine together with a program performing mapping between them.

A Decomposition Common to a Compiler and Interpretor for the Same Language

In an earlier attempt to apply these decomposition rules to a design project we constructed a translator for a Markov algorithm expressed in the notation described in [6]. Although it was not our intention to investigate the relation between compiling and interpretive translators of a langugage, we discovered that our decomposition was valid for a pure compiler and several

المنارة للاستشارات

varieties of interpretors for the language. Although there would be deep and substantial differences in the final running representations of each type of compiler, we found that the decisions implicit in the early decomposition held for all.

This would not have been true if we had divided responsibilities along the classical lines for either a compiler or interpretor (e.g. syntax recognizer, code generator, run time routines for a compiler). Instead the decomposition was based upon the hiding of various decisions as in the example above. Thus register representation, search algorithm, rule interpretation etc. were modules and these problems existed in both compiling and interpretive translators. Not only was the decomposition valid in all cases, but many of the routines could be used with only slight changes in any sort of translator.

This example provides additional support for the statement that the order in time in which processing is expected to take place should not be used in making the decomposition into modules. It further provides evidence that a careful job of decomposition can result in considerable carryover of work from one project to another.

A more detailed discussion of this example was contained in [8].

Hierarchical Structure

We can find a program hierarchy in the sense illustrated by Dijkstra [5] in the system defined according to decomposition 2. If a symbol table exists, it functions

المنارة للاستشارات

without any of the other modules, hence it is on level 1. Line storage is on level 1 if no symbol table is used or it is on level 2 otherwise. Input and Circular Shifter require line storage for their functioning. Output and Alphabetizer will require Circular Shifter, but since Circular Shifter and line holder are in some sense compatible, it would be easy to build a parameterized version of those routines which could be used to alphabetize or print out either the original lines or the circular shifts. In the first usage they would not require Circular Shifter; in the second they would. In other words, our design has allowed us to have a single representation for programs which may run at either of two levels in the hierarchy.

In discussions of system structure it is easy to confuse the benefits of a good decomposition with those of a hierarchical structure. We have a hierarchical structure if a certain relation may be defined between the modules or programs and that relation is a partial ordering. The relation we are concerned with is "uses" or "depends upon." It is better to use a relation between programs since in many cases one module depends upon only part of another module (e.g. Circular Shifter depends only on the output parts of the line holder and not on the correct working of *SETWORD*). It is conceivable that we could obtain the benefits that we have been discussing without such a partial ordering, e.g. if all the modules were on the same level. The partial ordering gives us two additional benefits. First, parts of the system are benefited (simplified) because they use the services of lower[2] levels. Second, we are able to cut off

[2] Here "lower" means "lower numbered."

المنارة للاستشارات

the upper levels and still have a usable and useful product. For example, the symbol table can be used in other applications; the line holder could be the basis of a question answering system. The existence of the hierarchical structure assures us that we can "prune" off the upper levels of the tree and start a new tree on the old trunk. If we had designed a system in which the "low level" modules made some use of the "high level" modules, we would not have the hierarchy, we would find it much harder to remove portions of the system, and "level" would not have much meaning in the system.

Since it is conceivable that we could have a system with the type of decomposition shown in version 1 (important design decisions in the interfaces) but retaining a hierarchical structure, we must conclude that hierarchical structure and "clean" decomposition are two desirable but *independent* properties of a system structure.

Conclusion

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more sub-

المنارة للاستشارات

routines, and instead allow subroutines and programs to be assembled collections of code from various modules.

Received August 1971; revised November 1971

References

1. Gauthier, Richard, and Pont, Stephen. *Designing Systems Programs*, (*C*), Prentice-Hall, Englewood Cliffs, N.J., 1970.
2. Hoare, C. A. R. Proof of a program, FIND. *Comm. ACM 14*, 1 (Jan. 1971), 39–45.
3. Parnas, D. L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May, 1972), 330–336.
4. Parnas, D. L. Information distribution aspects of design methodology. Tech. Rept., Depart. Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1971. Also presented at the IFIP Congress 1971, Ljubljana, Yugoslavia.
5. Dijkstra, E. W. The structure of "THE"-multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
6. Galler, B., and Perlis, A. J. *A View of Programming Languages*, Addison-Wesley, Reading, Mass., 1970.
7. Parnas, D. L. A course on software engineering. Proc. SIGCSE Technical Symposium, Mar. 1972.
8. Parnas, D. L. On the criteria to be used in decomposing systems into modules. Tech. Rept., Depart. Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1971.
9. Balzer, R. M. Dataless programming. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 535–544.
10. Mealy, G. H. Another look at data. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 525–534.
11. Wulf, W. A., Russell, D. B., and Habermann, A. N. BLISS, A language for systems programming. *Comm. ACM 14*, 12 (Dec. 1971), 780–790.

المنارة للاستشارات

David L. Parnas

On a ‘Buzzword’: Hierarchical Structure

Information Processing 74, IFIP Congress 74, North Holland, 1974
pp. 336–339

المنارة للاستشارات

ON A 'BUZZWORD': HIERARCHICAL STRUCTURE

David PARNAS

Technische Hochschule Darmstadt, Fachbereich Informatik
Research Group on Operating Systems I
Steubenplatz 12, 61 Darmstadt, West Germany

This paper discusses the use of the term "hierarchically structured" to describe the design of operating systems. Although the various uses of this term are often considered to be closely related, close examination of the use of the term shows that it has a number of quite different meanings. For example, one can find two different senses of "hierarchy" in a single operating system [3] and [6]. An understanding of the different meanings of the term is essential, if a designer wishes to apply recent work in Software Engineering and Design Methodology. This paper attempts to provide such an understanding.

INTRODUCTION

The phrase "hierarchical structure" has become a buzzword in the computer field. For many it has acquired a connotation so positive that it is akin to the quality of being a good mother. Others have rejected it as being an unrealistic restriction on the system [1]. This paper attempts to give some meaning to the term by reviewing some of the ways that the term has been used in various operating systems (e.g. T.H.E. [3], MULTICS [12], and the RC4ooo [8]) and providing some better definitions. Uses of the term, which had been considered equivalent or closely related, are shown to be independant. Discussions of the advantages and disadvantages of the various hierarchical restrictions are included.

GENERAL PROPERTIES OF ALL USES OF THE PHRASE "HIERARCHICAL STRUCTURE"

As discussed earlier [2], the word "structure" refers to a partial description of a system showing it as a collection of parts and showing some relations between the parts. We can term such a structure hierarchical, if a relation or predicate on pairs of the parts ($R(\alpha,\beta)$) allows us to define levels by saying that

المنارة للاستشارات

1. Level 0 is the set of parts α such that there does not exist a β such that $R(\alpha,\beta)$, and
2. Level i is the set of parts α such that
   a) there exists a β on level i-1 such that $R(\alpha,\beta)$ and
   b) if $R(\alpha,\gamma)$ then γ is on level i-1 _or lower_.

This is possible with a relation R only if the directed graph representing R has no loops.

The above definition is the most precise reasonably simple definition, which encompasses all uses of the word in the computer literature. This suggests that the statement "our Operating System has a hierarchical structure" carries no information at all. _Any_ system can be represented as a hierarchical system with one level and one part; more importantly, it is possible to divide _any_ system into parts and contrive a relation such that the system has a hierarchical structure. Before such a statement can carry any information at all, the way that the system is divided into parts and the nature of the relation must be specified.

The decision to produce a hierarchically structured system may restrict the class of possible systems, and may, therefore, introduce disadvantages as well as the desired advantages. In the remainder of this paper we shall introduce a variety of definitions for "hierarchical structure", and mention some advantages and disadvantages of the restriction imposed by these definitions.

1. THE PROGRAM HIERARCHY

Prof. E.W. Dijkstra in his paper on the T.H.E. system and in later papers on structured programming [3] and [4] has demonstrated the value of programming using layers of abstract machines. We venture the following definition for this program hierarchy. The parts of the system are subprograms, which may be called as if they were procedures.* We assume that

* They may be expanded as MACROS.

المنارة للاستشارات

each such program has a specified purpose (e.g. FNO :: = find next odd number in sequence or invoke DONE if there is none). The relation "uses" may be defined by USES(p_i,p_j)=*iff* p_i calls p_j *and* p_i will be considered incorrect if p_j does not function properly.

With the last clause we intend to imply that, our example, FNO does *not* "use" DONE in the sense defined here. The task of FNO is to invoke DONE; the purpose and "correctness" of DONE is irrelevant to FNO. Without excepting such calls, we could not consider a program to be higher in the hierarchy than the machine, which it uses. Most machines have "trap" facilities, and invoke software routines, when trap conditions occur.

A program divided into a set of subprograms may be said to be hierarchically structured, when the relation "uses" defines levels as described above. The term "abstract machine" is commonly used, because the relation between the lower level programs and the higher level programs is analoguous to the relation between hardware and software.

A few remarks are necessary here. First, *we* do not claim that the only good programs are hierarchically structured programs. Second, we point out that the way that the program is divided into subprograms can be rather arbitrary. For *any* program, some decompositions into subprograms may reveal a hierarchical structure, while other decompositions may show a graph with loops in it. As demonstrated in the simple example above, the specification of each program's purpose is critical!

The purpose of the restriction on program structure implied by this definition, is twofold. First, the calling program should be able to ignore the internal workings of the called program; the called program should make no assumptions about the internal structure of the calling program. Allowing the called program to call its user, might make this more difficult since each would have to be designed

المنارة للاستشارات

to work properly in the situations where it could be called by the other.

The second purpose might be termed "ease of subsetting". When a program has this "program hierarchy", the lower levels may always be used without the higher levels, when the higher levels are not ready or their services are not needed. An example of non-hierarchical systems would be one in which the "lower level" scheduling programs made use of the "high level" file system for storage of information about the tasks that it schedules. Assuming that nothing useful could be done without the scheduler, no subset of the system that did not include the file system could exist. The file system (usually a complex and "buggy" piece of software) could not be developped using the remainder of the system as a "virtual machine".

For those who argue that the hierarchical structuring proposed in this section prevents the use of recursive programming techniques, we remind them of the freedom available in choosing a decomposition into subprograms. If there exists a subset of the programs, which call each other recursively, we can view the group as a single program for this analysis and then consider the remaining structure to see, if it is hierarchical. In looking for possible subsets of a system, we must either include or exclude this group of programs as a single program.

One more remark: please, note that the division of the program into levels by the above discussed relation has no <u>necessary</u> connection with the division of the programs into modules as discussed in [5]. This is discussed further later (section 6).

2. THE "HABERMANN" HIERARCHY IN THE T.H.E. SYSTEM

The T.H.E. system was also hierarchical in another sense. In order to make the system relatively insensitive to the number of processors and their relative speeds, the system was designed as a set of "parallel sequential processes". The activities in

المنارة للاستشارات

the system were organized into "processes" such that the sequence of events within a process was relatively easy to predict, but the sequencing of events in different processes were considered unpredictable (the relative speeds of the processes were considered unknown). Resource allocation was done in terms of the processes and the processes exchanged work assignments and information. In carrying out a task, a process could assign part of the task to another process in the system.

One important relation between the processes in such a system is the relation "gives work to". In his thesis [6] Habermann assumed that "gives work to" defined a hierarchy to prove "harmonious cooperation". If we have an Operating System we want to show that a request of the system will generate only a finite (and reasonably small) number of requests to individual processes before the original request is satisfied. If the relation "gives work to" defines a hierarchy, we can prove our result by examining each process seperately to make sure that every request to it results in only a finite number of requests to other processes. If the relation is not hierarchical, a more difficult, "global", analysis would be required.

Restricting "gives work to" so that it defines a hierarchy helps in the establishment of the "well-behavedness", but it is certainly not a necessary condition for "harmonious cooperation".*

*This restriction is also valuable in human organizations. Where requests for administrative work flow only in one direction things go relatively smoothly, but in departments where the "leader" constantly refers requests "downward" to committees (which can themselves send requests to the "leader") we often find the system filling up with uncompleted tasks and a correspondingly large increase in overhead.

المنارة للاستشارات

In the T.H.E. system the two hierarchies described above coincided. Every level of abstraction was achieved by the introduction of parallel processes and these processes only gave work to those written to implement lower levels in the program hierarchy. One should not draw general conclusions about system structure on the basis of this coincidence. For example, the remark that "building a system with more levels than were found in the T.H.E. system is undesirable, because it introduces more queues" is often heard because of this coincidence. The later work by Dijkstra on structured programming [21] shows that the levels of abstraction are useful when there is only one process. Further, the "Habermann hierarchy" is useful, when the processes are controlled by badly structured programs. Adding levels in the program hierarchy need not introduce new processes or queues. Adding processes can be done without writing new programs.

The "program hierarchy" is only significant at times when humans are working with the program (e.g. when the program is being constructed or changed). If the programs were all implemented as macros, there would be no trace of this hierarchy in the running system. The "Habermann hierarchy" is a restriction on the run time behavior of the system. The theorems proven by Habermann would hold even if a process that is controlled by a program written at a low level in the program hierarchy "gave work to" a process which was controlled by a program originally written at a higher level in the program hierarchy. There are also no detrimental effects on the program hierarchy provided that the programs written at the lower level are not written in terms of programs at the higher level. Readers are referred to "Flatland" [7].

3. HIERARCHICAL STRUCTURES RELATING TO RESOURCE OWNERSHIP AND ALLOCATION

The RC4000 system [8] and [9] enforced a hierarchical relation based upon the ownership of memory. A generalization of that hierarchical structure has

المنارة للاستشارات

been proposed by Varney [1o] and similar hierarchical relationships are to be found in various commercial operating systems, though they are not often formally described.

In the RC4ooo system the objects were processes and the relation was "allocated a memory region to". Varney proposes extending the relation so that the hierarchical structure controlled the allocation of other resources as well. (In the RC4ooo systems specific areas of memory were allocated, but that was primarily a result of the lack of virtual memory hardware; in most systems of interest now, we can allocate quantities of a resource without allocating the specific physical resources until they are actually used). In many commercial systems we also find that resources are not allocated directly to the processes which use them. They are allocated to administrative units, who, in turn, may allocate them to other processes. In these systems we do not find any loops in the graph of "allocates resources to", and the relation defines a hierarchy, which is closely related to the RC4ooo structure.

This relation was not a significant one in the T.H.E. system, where allocating was done by a central allocator called a BANKER. Again this sense of hierarchy is not strongly related to the others, and if it is present with one or more of the others, they need not coincide.

The disadvantage of a non-trivial hierarchy (the hierarchy is present in a trivial form even in the T.H.E. system) of this sort are (1) poor resource utilization that may occur when some processes in the system are short of resources while other processes, under a different allocator in the hierarchy, have an excess; (2) high overhead that occurs when resources are tight. Requests for more resources must always go up all the levels of the hierarchy before being denied or granted. The central "banker" does not have these disadvantages. A central resource allocator, however, becomes complicated in situations where groups of related processes wish to dynamically share

المنارة للاستشارات

resources without influence by other such groups. Such situations can arise in systems that are used in real time by independant groups of users. The T.H.E. system did not have such problems and as a result, centralized resource allocation was quite natural.

It is this particular hierarchical relation which the Hydra group rejected. They did not mean to reject the general notion of hierarchical structure as suggested in the original report [1] and [11].

4. PROTECTION HIERARCHIES A LA MULTICS

Still another hierarchy can be found in the MULTICS system. The conventional two level approach to operating systems (low level called the supervisor, next level the users) has been generalized to a sequence of levels in the supervisor called "rings". The set of programs within a MULTICS process is organized in a hierarchical structure, the lower levels being known as the inner rings, and the higher levels being known as outer rings. Although the objects are programs, this relation is not the program hierarchy discussed in section 1. Calls occur in both directions and lower level programs may use higher level ones to get their work done [12].

Noting that certain data are much more crucial to operation of the system than other data, and that certain procedures are much more critical to the overall operation of the system than others, the designers have used this as the basis of their hierarchy. The data to which the system is most sensitive are controlled by the inner ring procedures, and transfers to those programs are very carefully controlled. Inner ring procedures have unrestricted access to programs and data in the outer rings. The outer rings contain data and procedures that effect a relatively small number of users and hence are less "sensitive". The hierarchy is most easily defined in terms of a relation "can be accessed by" since "sensitivity" in the sense used above is difficult to define. Low levels have unrestricted access to higher levels, but not vice versa.

المنارة للاستشارات

It is clear that placing restrictions on the relation "can be accessed by" is important to system reliability and security.

It has, however, been suggested that by insisting that the relation "can be accessed by" be a hierarchy, we prevent certain accessibility patterns that might be desired. We might have three segments in which A requires access to B, B to C, and C to A. No other access rights are needed or desirable. If we insist that "can be accessed by" define a hierarchy, we must (in this case) use the trivial hierarchy in which A, B, C are considered one part.

In the view of the author, the member of pairs in the relation "can be accessed by" should be minimized, but he sees no advantage in insisting that it define a hierarchy [13] and [14].

The actual MULTICS restriction is even stronger than requiring a hierarchy. Within a process, the relation must be a complete ordering.

5. HIERARCHIES AND "TOP DOWN" DESIGN METHODOLOGY

About the time that the T.H.E. system work appeared, it became popular to discuss design methods using such terms as "top down" and "outside in"[15], [16], and [17]. The simultaneous appearance of papers suggesting how to design well and a well designed system led to the unfounded assumption that the T.H.E. system had been the result of a "top down" design process. Even in more recent work [18] top down design and structured programming are considered almost synonymous.

Actually "outside in" was a much better term for what was intended, than was "top down"! The intention was to begin with a description of the system's user interface, and work in small, verifiable steps towards the implementation. The "top" in that hierarchy consisted of those parts of the system that were visible to the user. In a system designed according to the "program hierarchy", the lower level

المنارة للاستشارات

functions will be used by the higher level functions, but some of them may also be visible to the user (store and load, for example). Some functions on higher levels may not be available to him (Restart system). Those participants in the design of the T.H.E. system with whom I have discussed the question [19], report that they did not proceed with the design of the higher levels first.

6. HIERARCHICAL STRUCTURE AND DECOMPOSITION INTO MODULES

Often one wants to view a system as divided into "modules" (e.g. with the purpose outlined in [5] and [2o]). This division defines a relation "part of". A group of sub-programs is collected into a module, groups of modules collected into bigger modules, etc. This process defines a relation "part of" whose graph is clearly loop-free. It remains loop-free even if we allow programs or modules to be part of several modules - the part never includes the whole.

Note that we may allow programs in one module to call programs in another module, so that the module hierarchy just defined need not have any connection with the program hierarchy. Even allowing recursive calls between modules does not defeat the purpose of the modular decomposition (e.g. flexibility) [5], provided that programs in one module do not assume much about the programs in another.

7. LEVELS OF LANGUAGE

It is so common to hear phrases such as "high level language", "low level language" and "linguistic level" that it is necessary to comment on the relation between the implied language hierarchy and the hierarchies discussed in the earlier sections of this paper. It would be nice, if, for example, the higher level languages were the languages of the higher level "abstract machines" in the program hierarchy. Unfortunately, this author can find no such relation and cannot define the hierarchy

المنارة للاستشارات

that is implied in the use of those phrases. In moments of scepticism one might suggest that the relation is "less efficient than" or "has a bigger grammar than" or "has a bigger compiler than", however, none of those phrases suggests an ordering, which is completely consistant with the use of the term. It would be nice, if the next person to use the phrase "higher level language" in a paper would define the hierarchy to which he refers.

SUMMARY

The computer system design literature now contains quite a number of valuable suggestions for improving the comprehensibility and predictability of computer systems by imposing a hierarchical structure on the programs. This paper has tried to demonstrate that, although these suggestions have been described in quite similar terms, the structures implied by those suggestions are not necessarily closely related. Each of the suggestions must be understood and evaluated (for its applicability to a particular system design problem) independantly. Further, we have tried to show that, while each of the suggestions offers some advantages over an "unstructured" design, there are also disadvantages, which must be considered. The main purpose of this paper has been to provide some guidance for those reading earlier literature and to suggest a way for future authors to include more precise definitions in their papers on design methods.

ACKNOWLEDGMENT

The Author acknowledges the valuable suggestions of Mr. W. Bartussek (Technische Hochschule Darmstadt) and Mr. John Shore (Naval Research Laboratory, Washington, D.C.). Both of these gentlemen have made substantial contributions to the more precise formulation of many of the concepts in this paper; neither should be held responsible for the fuzziness, which unfortunately remains.

المنارة للاستشارات

REFERENCES

[1] Wulf, Cohen, Coowin, Jones, Levin, Pierson, Pollach, Hydra: The Kernel of a Multiprogramming System, Technical Report, Computer Science Department, Carnegie-Mellon University.

[2] David L. Parnas, Information Distribution Aspects of Design Methodology, Proceedings of the 1971 IFIP Congress, Booklet TA/3, 26-3o.

[3] E.W. Dijkstra, The Structure of the T.H.E. Multiprogramming System, Communications of the ACM, vol 11, no. 5, May 1968, 341-346.

[4] E.W. Dijkstra, Complexity controlled by Hierarchical Ordering of Function and Variability, Software Engineering, NATO.

[5] David L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, vol. 15, no. 12, December 1972, 1o53-1o58.

[6] A.N. Habermann, On the Harmonious Cooperation of Abstract Machines, Doctoral Dissertation, Technische Hogeschool Eindhoven, The Netherlands.

[7] Edwin A. Abbott, Flatland, the Romance of Many Dimensions, Dover Publications, Inc., New York, 1952.

[8] Per Brinch Hansen, The Nucleus of a Multiprogramming System, Communications of the ACM, vol. 13, no. 4, April 197o, 238-25o.

[9] RC4000 Reference Manuals for the Operating System, Regnecentralen Denmark.

[1o] R.C. Varney, Process Selection in a Hierarchical Operating System, SIGOPS Operating Review, June 1972.

[11] W. Wulf, C. Pierson, Private Discussions.

[12] R.W. Graham, Protection in an Information Processing Utility, Communications of the ACM, May 1968.

[13] W.R. Price and David L. Parnas, The Design of the Virtual Memory Aspects of a Virtual Machine, Proceedings of the SIGARCH-SIGOPS Workshop on Virtual Machines, March 1973.

[14] W.R. Price, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., U.S.A.

المنارة للاستشارات

[15] David L. Parnas and Darringer, SODAS and Methodology for System Design, Proceedings of 1967 FJCC.

[16] David L. Parnas, More on Design Methodology and Simulation, Proceedings of the 1969 SJCC.

[17] Zurcher and Randell, Iterative Multi-Level Modeling, Proceedings of the 1968 IFIP Congress.

[18] F.T. Baker, System Quality through Structured Programming, Proceedings of the 1972 FJCC.

[19] E.W. Dijkstra, A.N. Habermann, Private Discussions

[2o] David L. Parnas, Some Conclusions from an Experiment in Software Engineering, Proceedings of the 1972 FJCC.

[21] E.W. Dijkstra, A Short Introduction to the Art of Programming, in O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, New York, 1972.

المنارة للاستشارات

Niklaus Wirth

The Programming Language Pascal

Acta Informatica, Vol. 1, Fasc. 1, 1971
pp. 35–63

المنارة للاستشارات

Acta Informatica 1, 35–63 (1971)
© by Springer-Verlag 1971

The Programming Language Pascal

N. Wirth*

Received October 30, 1970

Summary. A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60, its range of applicability is considerably increased due to a variety of data structuring facilities. In view of its intended usage both as a convenient basis to teach programming and as an efficient tool to write large programs, emphasis was placed on keeping the number of fundamental concepts reasonably small, on a simple and systematic language structure, and on efficient implementability. A one-pass compiler has been constructed for the CDC 6000 computer family; it is expressed entirely in terms of Pascal itself.

1. Introduction

The development of the language *Pascal* is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. The second is to develop implementations of this language which are both reliable and efficient on presently available computers, dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any nontrivial system is bound to contain mistakes forever.

There is of course plenty of reason to be cautious with the introduction of yet another programming language, and the objection against teaching programming in a language which is not widely used and accepted has undoubtedly some justification—at least based on short-term commercial reasoning. However, the choice of a language for teaching based on its widespread acceptance and availability, together with the fact that the language most widely taught is thereafter going to be the one most widely used, forms the safest recipe for stagnation in a subject of such profound paedagogical influence. I consider it therefore well worth-while to make an effort to break this vicious circle.

Of course a new language should not be developed just for the sake of novelty; existing languages should be used as a basis for development wherever they meet the chosen objectives, such as a systematic structure, flexibility of program and data structuring, and efficient implementability. In that sense Algol 60 was used as a basis for Pascal, since it meets most of these demands to a much higher degree than any other standard language [1]. Thus the principles of structuring, and in fact the form of expressions, are copied from Algol 60. It was, however, not deemed appropriate to adopt Algol 60 as a subset of Pascal; certain construction principles, particularly those of declarations, would have been incom-

* Fachgruppe Computer-Wissenschaften, Eidg. Technische Hochschule, Zürich, Schweiz.

patible with those allowing a natural and convenient representation of the additional features of Pascal. However, conversion of ALGOL 60 programs to Pascal can be considered as a negligible effort of transcription, particularly if they obey the rules of the IFIP ALGOL Subset [2].

The main extensions relative to ALGOL 60 lie in the domain of data structuring facilities, since their lack in ALGOL 60 was considered as the prime cause for its relatively narrow range of applicability. The introduction of record and file structures should make it possible to solve commercial type problems with Pascal, or at least to employ it successfully to demonstrate such problems in a programming course. This should help erase the mystical belief in the segregation between scientific and commercial programming methods. A first step in extending the data definition facilities of ALGOL 60 was undertaken in an effort to define a successor to ALGOL in 1965 [3]. This language is a direct predecessor of Pascal, and was the source of many features such as e.g. the while and case statements and of record structures.

Pascal has been implemented on the CDC 6000 computers. The compiler is written in Pascal itself as a one-pass system which will be the subject of a subsequent report. The "dialect" processed by this implementation is described by a few amendments to the general description of Pascal. They are included here as a separate chapter to demonstrate the brevity of a manual necessary to characterise a particular implementation. Moreover, they show how facilities are introduced into this high-level, machine independent programming language, which permit the programmer to take advantage of the characteristics of a particular machine.

The syntax of Pascal has been kept as simple as possible. Most statements and declarations begin with a unique key word. This property facilitates both the understanding of programs by human readers and the processing by computers. In fact, the syntax has been devised so that Pascal texts can be scanned by the simplest techniques of syntactic analysis. This textual simplicity is particularly desirable, if the compiler is required to possess the capability to detect and diagnose errors and to proceed thereafter in a sensible manner.

2. Summary of the Language

An algorithm or computer program consists of two essential parts, a description of *actions* which are to be performed, and a description of the *data* which are manipulated by these actions. Actions are described in Pascal by so-called *statements*, and data are described by so-called *declarations* and *definitions*.

The data are represented by values of *variables*. Every variable occuring in a statement must be introduced by a *variable declaration* which associates an identifier and a data type with that variable. The *data type* essentially defines the set of values which may be assumed by that variable. A data type may in Pascal be either directly described in the variable declaration, or it may be referenced by a type identifier, in which case this identifier must be described by an explicit *type definition*.

The basic data types are the *scalar* types. Their definition indicates an ordered set of values, i.e. introduces an identifier as a constant standing for each value

المنارة للاستشارات

in the set. Apart from the definable scalar types, there exist in Pascal four *standard scalar types* whose values are not denoted by identifiers, but instead by numbers and quotations respectively, which are syntactically distinct from identifiers. These types are: *integer*, *real*, *char*, and *alfa*.

The set of values of type *char* is the character set available on the printers of a particular installation. *Alfa* type values consist of sequences of characters whose length again is implementation dependent, i.e. is the number of characters packed per word. Individual characters are not directly accessible, but *alfa* quantities can be unpacked into a character array (and vice-versa) by a standard procedure.

A scalar type may also be defined as a *subrange* of another scalar type by indicating the smallest and the largest value of the subrange.

Structured types are defined by describing the types of their components and by indicating a *structuring method*. The various structuring methods differ in the selection mechanism serving to select the components of a variable of the structured type. In Pascal, there are five structuring methods available: array structure, record structure, powerset structure, file structure, and class structure.

In an *array structure*, all components are of the same type. A component is selected by an array selector, or computable *index*, whose type is indicated in the array type definition and which must be scalar. It is usually a programmer-defined scalar type, or a subrange of the type *integer*.

In a *record structure*, the components (called *fields*) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing the program), a record selector does not contain a computable value, but instead consists of an identifier uniquely denoting the component to be selected. These component identifiers are defined in the record type definition.

A record type may be specified as consisting of several *variants*. This implies that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The difference may consist of a different number and different types of components. The variant which is assumed by the current value of a record variable is indicated by a component field which is common to all variants and is called the *tag field*. Usually, the part common to all variants will consist of several components, including the tag field.

A *powerset structure* defines a set of values which is the powerset of its base type, i.e. the set of all subsets of values of the base type. The base type must be a scalar type, and will usually be a programmer-defined scalar type or a subrange of the type *integer*.

A *file structure* is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any instance, only one component is directly accessible. The other components are made accessible through execution of standard file positioning procedures. A file is at any time in one of the three modes called *input*, *output*, and *neutral*. According to the mode, a file can be read sequentially, or it can be written by appending components to the existing sequence of components. File positioning procedures may influence the mode. The file type definition does not determine the number of components, and this number is variable during execution of the program.

المنارة للاستشارات

The *class structure* defines a class of components of the same type whose number may alter during execution of a program. Each declaration of a variable with class structure introduces a set of variables of the component type. The set is initially empty. Every activation of the standard procedure *alloc* (with the class as implied parameter) will generate (or allocate) a new component variable in the class and yield a value through which this new component variable may be accessed. This value is called a *pointer*, and may be assigned to variables of type pointer. Every pointer variable, however, is through its declaration bound to a fixed class variable, and because of this *binding* may only assume values pointing to components of that class. There exists a pointer value **nil** which points to no component whatsoever, and may be assumed by any pointer variable irrespective of its binding. Through the use of class structures it is possible to construct data corresponding to any finite graph with pointers representing edges and component variables representing nodes.

The most fundamental statement is the *assignment statement*. It specifies that a newly computed value be assigned to a variable (or component of a variable). The value is obtained by evaluating an *expression*. Pascal defines a fixed set of operators, each of which can be regarded as describing a mapping from the operand types into the result type. The set of operators is subdivided into groups of

1. arithmetic operators of addition, subtraction, sign inversion, multiplication, division, and computing the remainder. The operand and result types are the types *integer* and *real*, or subrange types of *integer*.

2. Boolean operators of negation, union (or), and conjunction (and). The operand and result types are *Boolean* (which is a standard type).

3. set operators of union, intersection, and difference. The operands and results are of any powerset type.

4. relational operators of equality, inequality, ordering and set membership. The result of relational operations is of type *Boolean*. Any two operands may be compared for equality as long as they are of the same type. The ordering relations apply only to scalar types.

The assignment statement is a so-called *simple statement*, since it does not contain any other statement within itself. Another kind of simple statement is the *procedure statement*, which causes the execution of the designated procedure (see below). Simple statements are the components or building blocks of *structured statements*, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by the *compound statement*, conditional or selective execution by the *if statement* and the *case statement*, and repeated execution by the *repeat statement*, the *while statement*, and the *for statement*. The if statement serves to make the execution of a statement dependent on the value of a *Boolean* expression, and the case statement allows for the selection among many statements according to the value of a selector. The for statement is used when the number of iterations is known beforehand, and the repeat and while statements are used otherwise.

A statement can be given a name (identifier), and be referenced through that identifier. The statement is then called a *procedure*, and its declaration a *procedure*

المنارة للاستشارات

declaration. Such a declaration may additionally contain a set of variable declarations, type definitions and further procedure declarations. The variables, types and procedures thus defined can be referenced only within the procedure itself, and are therefore called *local* to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the *scope* of these identifiers. Since procedures may be declared local to other procedures, scopes may be nested.

A procedure may have a fixed number of parameters, which are classified into constant-, variable-, procedure-, and function parameters. In the case of a variable parameter, its type has to be specified in the declaration of the formal parameter. If the actual variable parameter contains a (computable) selector, this selector is evaluated before the procedure is activated in order to designate the selected component variable.

Functions are declared analogously to procedures. In order to eliminate side-effects, assignments to non-local variables are not allowed to occur within the function.

3. Notation, Terminology, and Vocabulary

According to traditional Backus-Naur form, syntactic constructs are denoted by English words enclosed between the angular brackets ⟨and⟩. These words also describe the nature or meaning of the construct, and are used in the accompanying description of semantics. Possible repetition of a construct is indicated by an asterisk (0 or more repetitions) or a circled plus sign (1 or more repetitions). If a sequence of constructs to be repeated consists of more than one element, it is enclosed by the meta-brackets **{**and**}**.

The basic *vocabulary* consists of basic symbols classified into letters, digits, and special symbols.

⟨letter⟩ ::= *A*|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*|*N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*|
a|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨special symbol⟩ ::= + | − | * | / | ∨ | ∧ | ¬ | = | ≠ | < | > | ≤ | ≥ | (|) | [|] | { | } | := |
$_{10}$ | . | , | ; | : | ' | ↑ | **div** | **mod** | **nil** | **in** |
if | **then** | **else** | **case** | **of** | **repeat** | **until** | **while** | **do** |
for | **to** | **downto** | **begin** | **end** | **with** | **goto** |
var | **type** | **array** | **record** | **powerset** | **file** | **class** |
function | **procedure** | **const**

The construct

{⟨any sequence of symbols not containing "}"⟩}

may be inserted between any two identifiers, numbers (cf. 4), or special symbols. It is called a *comment* and may be removed from the program text without altering its meaning.

المنارة للاستشارات

4. Identifiers and Numbers

Identifiers serve to denote constants, types, variables, procedures and functions. Their association must be unique within their scope of validity, i.e. within the procedure or function in which they are declared (cf. 10 and 11).

⟨identifier⟩ ::= ⟨letter⟩ ⟨letter or digit⟩*

⟨letter or digit⟩ ::= ⟨letter⟩ | ⟨digit⟩

The decimal notation is used for numbers, which are the constants of the data types *integer* and *real*. The symbol $_{10}$ preceding the scale factor is pronounced as "times 10 to the power of".

⟨number⟩ ::= ⟨integer⟩ | ⟨real number⟩
⟨integer⟩ ::= ⟨digit⟩$^{\oplus}$
⟨real number⟩ ::= ⟨digit⟩$^{\oplus}$. ⟨digit⟩$^{\oplus}$ |
    ⟨digit⟩$^{\oplus}$. ⟨digit⟩$^{\oplus}{}_{10}$ ⟨scale factor⟩ | ⟨integer⟩$_{10}$ ⟨scale factor⟩
⟨scale factor⟩ ::= ⟨digit⟩$^{\oplus}$ | ⟨sign⟩ ⟨digit⟩$^{\oplus}$
⟨sign⟩ ::= + | −

Examples:

1   100   0.1 $5_{10}-3$ $87.35_{10}+8$

5. Constant Definitions

A constant definition introduces an identifier as a synonym to a constant.

⟨unsigned constant⟩ ::= ⟨number⟩ | '⟨character⟩$^{\oplus}$' | ⟨identifier⟩ | **nil**
⟨constant⟩ ::= ⟨unsigned constant⟩ | ⟨sign⟩ ⟨number⟩
⟨constant definition⟩ ::= ⟨identifier⟩ = ⟨constant⟩

6. Data Type Definitions

A data type determines the set of values which variables of that type may assume and associates an identifier with the type. In the case of structured types, it also defines their structuring method.

⟨type⟩ ::= ⟨scalar type⟩ | ⟨subrange type⟩ | ⟨array type⟩ | ⟨record type⟩ |
    ⟨powerset type⟩ | ⟨file type⟩ | ⟨class type⟩ | ⟨pointer type⟩ |
    ⟨type identifier⟩
⟨type identifier⟩ ::= ⟨identifier⟩
⟨type definition⟩ ::= ⟨identifier⟩ = ⟨type⟩

6.1. Scalar Types

A scalar type defines an ordered set of values by enumeration of the identifiers which denote these values.

⟨scalar type⟩ ::= (⟨identifier⟩ {,⟨identifier⟩}*)

المنارة للاستشارات

Examples:

(*red, orange, yellow, green, blue*)
(*club, diamond, heart, spade*)
(*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*)

Functions applying to all scalar types are:

succ the succeeding value (in the enumeration)
pred the preceding value (in the enumeration)

6.1.1. Standard Scalar Types

The following types are standard in Pascal, i.e. the identifier denoting them is predefined:

integer the values are the integers within a range depending on the particular implementation. The values are denoted by integers (cf. 4) and not by identifiers.

real the values are a subset of the real numbers depending on the particular implementation. The values are denoted by real numbers as defined in paragraph 4.

Boolean (*false, true*)

char the values are a set of characters depending on a particular implementation. They are denoted by the characters themselves enclosed within quotes.

alfa the values are sequences of n characters, where n is an implementation dependent parameter. If α and β are values of type alfa

$$\alpha = a_1 \ldots a_k \ldots a_n$$
$$\beta = b_1 \ldots b_k \ldots b_n,$$

then

$\alpha = \beta$, if and only if $a_i = b_i$ for $i = 1 \ldots n$,

$\alpha < \beta$, if and only if $a_i = b_i$ for $i = 1 \ldots k-1$ and $a_k < b_k$,

$\alpha > \beta$, if and only if $a_i = b_i$ for $i = 1 \ldots k-1$ and $a_k > b_k$.

Alfa values are denoted by sequences of (at most) n characters enclosed in quotes. Trailing blanks may be omitted. Alfa quantities may be regarded as a packed representation of short character arrays (cf. also 10.1.3.).

6.1.2. Subrange Types

A type may be defined as a subrange of another scalar type by indication of the least and the highest value in the subrange. The first constant specifies the lower bound, and must not be greater than the upper bound.

⟨subrange type⟩ ::= ⟨constant⟩ . . ⟨constant⟩

المنارة للاستشارات

Examples:

1..100
−10..+10
Monday..Friday

6.2. Structured Types

6.2.1. Array Types

An array type is a structure consisting of a fixed number of components which are all of the same type, called the *component type*. The elements of the array are designated by indices, values belonging to the so-called *index type*. The array type definition specifies the component type as well as the index type.

⟨array type⟩ ::= **array** [⟨index type⟩ **{,** ⟨index type⟩**}***] **of** ⟨component type⟩
⟨index type⟩ ::= ⟨scalar type⟩ | ⟨subrange type⟩ | ⟨type identifier⟩
⟨component type⟩ ::= ⟨type⟩

If *n* index types are specified, the array type is called *n-dimensional*, and a component is designated by *n* indices.

Examples:

array [1..100] **of** *real*
array [1..10, 1..20] **of** 0..99
array [−10..+10] **of** *Boolean*
array [*Boolean*] **of** *Color*

6.2.2. Record Types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called *field*, its type and an identifier which denotes it. The scope of these so-called *field identifiers* is the record definition itself, and they are also accessible within a field designator (cf. 7.2) refering to a record variable of this type.

A record type may have several *variants*, in which case a certain field is designated as the *tag field*, whose value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a case label which is a constant of the type of the tag field.

⟨record type⟩ ::= **record** ⟨field list⟩ **end**
⟨field list⟩ ::= ⟨fixed part⟩ | ⟨fixed part⟩; ⟨variant part⟩ | ⟨variant part⟩
⟨fixed part⟩ ::= ⟨record section⟩ **{**; ⟨record section⟩**}***
⟨record section⟩ ::= ⟨field identifier⟩ **{,** ⟨field identifier⟩**}***: ⟨type⟩
⟨variant part⟩ ::= **case** ⟨tag field⟩ : ⟨type identifier⟩ **of** ⟨variant⟩ **{**; ⟨variant⟩**}***
⟨variant⟩ ::= **{**⟨case label⟩ :**}**$^{\oplus}$ (⟨field list⟩) | **{**⟨case label⟩**}**$^{\oplus}$
⟨case label⟩ ::= ⟨unsigned constant⟩
⟨tag field⟩ ::= ⟨identifier⟩

المنارة للاستشارات

Examples:

```
record day: 1..31;
        month: 1..12;
        year: 0..2000
end

record name, firstname: alfa;
        age: 0..99;
end

record x, y: real;
        area: real;
case s: Shape of
triangle:  (side: real;
            inclination, angle1  angle2: Angle);
rectangle: (side1, side2: real;
            skew, angle3: Angle);
circle:    (diameter: real)
end
```

6.2.3. Powerset Types

A powerset type defines a range of values as the powerset of another scalar type, the so-called *base type*. Operators applicable to all powerset types are:

- **∨** union
- **∧** intersection
- **—** set difference
- **in** membership

⟨powerset type⟩ ::= **powerset** ⟨type identifier⟩ | **powerset** ⟨subrange type⟩

6.2.4. File Types

A file type definition specifies a structure consisting of a sequence of components, all of the same type. The number of components, called the *length* of the file, is not fixed by the file type definition, i.e. each variable of that type may have a value with a different, varying length.

Associated with each variable of file type is a *file position* or *file pointer* denoting a specific element. The file position or the file pointer can be moved by certain standard procedures, some of which are only applicable when the file is in one of the three *modes*: input (being read), output (being written), or neutral (passive). Initially, a file variable is in the neutral mode.

⟨file type⟩ ::= **file of** ⟨type⟩

6.2.5. Class Types

A class type definition specifies a structure consisting of a class of components, all of the same type. The number of components is variable; the initial number

المنارة للاستشارات

upon declaration of a variable of class type is zero. Components are created (allocated) during execution of the program through the standard procedure *alloc*. The maximum number of components which can thus be created, however, is specified in the type definition.

⟨class type⟩ ::= **class** ⟨maxnum⟩ **of** ⟨type⟩

⟨maxnum⟩ ::= ⟨integer⟩

6.2.6. Pointer Types

A pointer type is associated with every variable of class type. Its values are the potential pointers to the components of that class variable (cf. 7.5), and the pointer constant **nil**, designating no component. A pointer type is said to be *bound* to its class variable.

⟨pointer type⟩ ::= ↑⟨class variable⟩

⟨class variable⟩ ::= ⟨variable⟩

Examples of type definitions:

```
Color   = (red, yellow, green, blue)
Sex     = (male, female)
Charfile = file of char
Shape   = (triangle, rectangle, circle)
Card    = array [1..80] of char
Complex = record realpart, imagpart: real end
Person  = record name, firstname: alfa;
                 age: integer;
                 married: Boolean;
                 father, youngestchild, eldersibling: ↑family;
          case s: Sex of
          male: (enlisted, bold: Boolean);
          female: (pregnant: Boolean;
                 size: array [1..3] of integer)
          end
```

7. Declarations and Denotations of Variables

Variable declarations consist of a list of identifiers denoting the new variables, followed by their type.

⟨variable declaration⟩ ::= ⟨identifier⟩ {,⟨identifier⟩}*:⟨type⟩

Two *standard file variables* can be assumed to be predeclared as

input, *output*: **file of** *char*

The file *input* is restricted to input mode (reading only), and the file *output* is restricted to output mode (writing only). A Pascal program should be regarded as a procedure with these two variables as formal parameters. The corresponding

المنارة للاستشارات

actual parameters are expected either to be the standard input and output media of the computer installation, or to be specifyable in the system command activating the Pascal system.

Examples:

x, y, z: *real*
u, v: *Complex*
i, j: *integer*
k: 0..9
p, q: *Boolean*
operator: (*plus, times, absval*)
a: **array** [0..63] **of** *real*
b: **array** [*Color, Boolean*] **of**
  record *occurrence*: *integer*;
    appeal: *real*
  end
c: *Color*
f: **file of** *Card*
hue1, hue2: **powerset** *Color*
family: **class** 100 **of** *Person*
p1, p2: ↑*family*

Denotations of variables either denote an entire variable or a component of a variable.

⟨variable⟩ ::= ⟨entire variable⟩ | ⟨component variable⟩

7.1. *Entire Variables*

An entire variable is denoted by its identifier.

⟨entire variable⟩ ::= ⟨variable identifier⟩
⟨variable identifier⟩ ::= ⟨identifier⟩

7.2. *Component Variables*

A component of a variable is denoted by the denotation for the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

⟨component variable⟩ ::= ⟨indexed variable⟩ | ⟨field designator⟩ |
⟨current file component⟩ | ⟨referenced component⟩

7.2.1. Indexed Variables

A component of an *n*-dimensional array variable is denoted by the denotation of the variable followed by *n* index expressions.

⟨indexed variable⟩ ::= ⟨array variable⟩ [⟨expression⟩ {,⟨expression⟩}*]
⟨array variable⟩ ::= ⟨variable⟩

المنارة للاستشارات

The types of the index expressions must correspond with the index types declared in the definition of the array type.

Examples:

a [12]
a [*i* + *j*]
b [*red*, *true*]
b [*succ* (*c*), *p* ∧ *q*]
f ↑ [1]

7.2.2. Field Designators

A component of a record variable is denoted by the denotation of the record variable followed by the field identifier of the component.

⟨field designator⟩ ::= ⟨record variable⟩ . ⟨field identifier⟩
⟨record variable⟩ ::= ⟨variable⟩
⟨field identifier⟩ ::= ⟨identifier⟩

Examples:

u . realpart
v . realpart
b [*red*, *true*] . *appeal*
p2 ↑ . *size*

7.2.3. Current File Components

At any time, only the one component determined by the current file position (or file pointer) is directly accessible.

⟨current file component⟩ ::= ⟨file variable⟩ ↑
⟨file variable⟩ ::= ⟨variable⟩

7.2.4. Referenced Components

Components of class variables are referenced by pointers.

⟨referenced component⟩ ::= ⟨pointer variable⟩ ↑
⟨pointer variable⟩ ::= ⟨variable⟩

Thus, if *p1* is a pointer variable which is bound to a class variable *v*, *p1* denotes that variable and its pointer value, whereas *p1* ↑ denotes the component of *v* referenced by *p1*.

Examples:

p1 ↑ . *father*
p1 ↑ . *eldersibling* ↑ . *youngestchild*

المنارة للاستشارات

8. Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands, i.e. variables and constants, operators, and functions.

The rules of composition specify operator *precedences* according to four classes of operators. The operator ¬ has the highest precedence, followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators. Sequences of operators of the same precedence are executed from left to right. These rules of precedence are reflected by the following syntax:

⟨factor⟩ ::= ⟨variable⟩ | ⟨unsigned constant⟩ | ⟨function designator⟩ |
  ⟨set⟩ | (⟨expression⟩) | ¬ ⟨factor⟩

⟨set⟩ ::= [⟨expression⟩ {,⟨expression⟩}*] | [ ]

⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨multiplying operator⟩⟨factor⟩

⟨simple expression⟩ ::= ⟨term⟩ |
  ⟨simple expression⟩⟨adding operator⟩⟨term⟩ |
  ⟨adding operator⟩⟨term⟩

⟨expression⟩ ::= ⟨simple expression⟩ |
  ⟨simple expression⟩⟨relational operator⟩
  ⟨simple expression⟩

Expressions which are members of a set must all be of the same type, which is the base type of the set. [ ] denotes the empty set.

Examples:

Factors:
x
15
$(x+y+z)$
$sin(x+y)$
$[red, c, green]$
$\neg p$

Terms:
$x*y$
$i/(1-i)$
$p \wedge q$
$(x \leq y) \wedge (y < z)$

Simple expressions:
$x+y$
$-x$
$hue1 \vee hue2$
$i*j+1$

المنارة للاستشارات

9.1.1. Assignment Statements

The assignment statement serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator symbol is :=, pronounced as "becomes".

⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩ |
⟨function identifier⟩ := ⟨expression⟩

The variable (or the function) and the expression must be must be of identical type (but neither class nor file type), with the following exceptions permitted:

1. the type of the variable is *real*, and the type of the expression is *integer* or a subrange thereof.

2. the type of the expression is a subrange of the type of the variable.

Examples:

$x := y + 2.5$
$p := (1 \leqq i) \wedge (i < 100)$
$i := sqr(k) - (i * j)$
$hue := [blue, succ(c)]$

9.1.2. Procedure Statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of *actual parameters* which are substituted in place of their corresponding *formal parameters* defined in the procedure declaration (cf. 10). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist four kinds of parameters: variable-, constant-, procedure parameters (the actual parameter is a procedure identifier), and function parameters (the actual parameter is a function identifier).

In the case of variable parameters, the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated when the substitution takes place, i.e. before the execution of the procedure. If the parameter is a constant parameter, then the corresponding actual parameter must be an expression.

⟨procedure statement⟩ ::= ⟨procedure identifier⟩ |
⟨procedure identifier⟩ (⟨actual parameter⟩
{, ⟨actual parameter⟩}*)

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨variable⟩ |
⟨procedure identifier⟩ | ⟨function identifier⟩

Examples:

next
Transpose (a, n, m)
Bisect $(sin, -1, +2, x, q)$

المنارة للاستشارات

9.1.3. Goto Statements

A goto statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label. Labels can be placed in front of statements being part of a compound statement (cf. 9.2.1.).

⟨goto statement⟩ ::= **goto** ⟨label⟩

⟨label⟩ ::= ⟨integer⟩

The following restriction holds concerning the applicability of labels:

The scope (cf. 10) of a label is the procedure declaration within which it is defined. It is therefore not possible to jump into a procedure.

9.2. Structured Statements

Structured statements are constructs composed of other statements which have to be executed either in sequence (compound statement), conditionally (conditional statements), or repeatedly (repetitive statements).

⟨structured statement⟩ ::= ⟨compound statement⟩ |
⟨conditional statement⟩ | ⟨repetitive statement⟩ |
⟨with statement⟩

9.2.1. Compound Statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. Each statement may be preceded by a label which can be referenced by a goto statement (cf. 9.1.3.).

⟨compound statement⟩ ::=
begin ⟨component statement⟩ **{**;⟨component statement⟩**}*** **end**

⟨component statement⟩ ::=
⟨statement⟩ | ⟨label definition⟩ ⟨statement⟩

⟨label definition⟩ ::= ⟨label⟩ :

Example:

begin *z* := *x*; *x* := *y*; *y* := *z* **end**

9.2.2. Conditional Statements

A conditional statement selects for execution a single one of its component statements.

⟨conditional statement⟩ ::= ⟨if statement⟩ | ⟨case statement⟩

9.2.2.1. If Statements

The if statement specifies that a statement be executed only if a certain condition (*Boolean* expression) is *true*. If it is *false*, then either no statement is to be executed, or the statement following the symbol **else** is to be executed.

⟨if statement⟩ ::= **if** ⟨expression⟩ **then** ⟨statement⟩ |
if ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩

The expression between the symbols **if** and **then** must be of type *Boolean*.

المنارة للاستشارات

Note: The syntactic ambiguity arising from the construct

if ⟨expression—1⟩ **then if** ⟨expression—2⟩ **then** ⟨statement—1⟩
  else ⟨statement—2⟩

is resolved by interpreting the construct as equivalent to

if ⟨expression—1⟩ **then**
  begin if ⟨expression—2⟩ **then** ⟨statement—1⟩ **else** ⟨statement—2⟩
  end

Examples:

if $x < 1.5$ **then** $z := x + y$ **else** $z := 1.5$

if $p \neq$ **nil then** $p := p\uparrow.$*father*

9.2.2.2. Case Statements

The case statement consists of an expression (the selector) and a list of statements, each being labeled by a constant of the type of the selector. It specifies that the one statement be executed whose label is equal to the current value of the selector.

⟨case statement⟩ ::= **case** ⟨expression⟩ **of**
  ⟨case list element⟩ {; ⟨case list element⟩}* **end**

⟨case list element⟩ ::= {⟨case label⟩:}$^{\oplus}$ ⟨statement⟩ | {⟨case label⟩:}$^{\oplus}$

Example:

case *operator* **of**
plus: $x := x + y$;
times: $x := x * y$;
absval: **if** $x < 0$ **then** $x := -x$
end

9.2.3. Repetitive Statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known beforehand, i.e. before the repetitions are started, the for statement is the appropriate construct to express this situation; otherwise the while or repeat statement should be used.

⟨repetitive statement⟩ ::= ⟨while statement⟩ |
  ⟨repeat statement⟩ | ⟨for statement⟩

9.2.3.1. While Statements

⟨while statement⟩ ::= **while** ⟨expression⟩ **do** ⟨statement⟩

The expression controlling repetition must be of type *Boolean*. The statement is repeatedly executed until the expression becomes *false*. If its value is *false* at the beginning, the statement is not executed at all. The while statement

while e **do** S

المنارة للاستشارات

is equivalent to

```
if e then
    begin S;
        while e do S
    end
```

Examples:

```
while (a[i] ≠ x) ∧ (i < n) do i := i + 1
while i > 0 do
begin if odd(i) then z := z * x;
        i := i div 2;
        x := sqr(x)
end
```

9.2.3.2. Repeat Statements

⟨repeat statement⟩ ::=
    repeat ⟨statement⟩ {; ⟨statement⟩}* **until** ⟨expression⟩

The expression controlling repetition must be of type *Boolean*. The sequence of statements between the symbols **repeat** and **until** is repeatedly (and at least once) executed until the expression becomes *true*. The repeat statement

```
repeat S until e
```

is equivalent to

```
begin S;
    if ¬e then
        repeat S until e
end
```

Examples:

```
repeat k := i mod j;
        i := j;
        j := k
until j = 0

repeat get(f)
until (f↑ = a) ∨ eof(f)
```

9.2.3.3. For Statements

The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the *control variable* of the for statement.

⟨for statement⟩ ::= **for** ⟨control variable⟩ := ⟨for list⟩ **do** ⟨statement⟩
⟨for list⟩ ::= ⟨initial value⟩ **to** ⟨final value⟩ |
    ⟨initial value⟩ **downto** ⟨final value⟩

المنارة للاستشارات

⟨control variable⟩ ::= ⟨identifier⟩

⟨initial value⟩ ::= ⟨expression⟩

⟨final value⟩ ::= ⟨expression⟩

The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof).

A for statement of the form

```
for v := e1 to e2 do S
```

is equivalent to the statement

```
if e1 ≤ e2 then
begin v := e1; S;
   for v := succ(v) to e2 do S
end
```

and a for statement of the form

```
for v := e1 downto e2 do S
```

is equivalent to the statement

```
if e1 ≥ e2 then
begin v := e1; S;
   for v := pred(v) downto e2 do S
end
```

Note: The repeated statement S must alter neither the value of the control variable nor the final value.

Examples:

```
for i := 2 to 100 do if a[i] > max then max := a[i]
for i := 1 to n do
for j := 1 to n do
begin x := 0;
   for k := 1 to n do x := x + a[i, k]*b[k, j];
   c[i, j] := x
end
for c := red to blue do try(c)
```

9.2.4. With Statements

⟨with statement⟩ ::= **with** ⟨record variable⟩ **do** ⟨statement⟩

Within the component statement of the with statement, the components (fields) of the record variable specified by the with clause can be denoted by their field identifier only, i.e. without preceding them with the denotation of the entire record variable. The with clause effectively opens the scope containing the field identifiers of the specified record variable, so that the field identifiers may occur as variable identifiers.

المنارة للاستشارات

Example:

```
with date do
begin
   if month = 12 then
   begin month := 1; year := year + 1
   end else month := month + 1
end
```

This statement is equivalent to

```
begin
   if date.month = 12 then
   begin date.month := 1; date.year := date.year + 1
   end else date.month := date.month + 1
end
```

10. Procedure Declarations

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements. A procedure declaration consists of the following parts, any of which, except the first and the last, may be empty:

⟨procedure declaration⟩ ::=
  ⟨procedure heading⟩
  ⟨constant definition part⟩ ⟨type definition part⟩
  ⟨variable declaration part⟩
  ⟨procedure and function declaration part⟩ ⟨statement part⟩

The procedure heading specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either constant-, variable, procedure-, or function parameters (cf. also 9.1.2.).

⟨procedure heading⟩ ::= **procedure** ⟨identifier⟩ ; |
  procedure ⟨identifier⟩ (⟨formal parameter section⟩
   {;⟨formal parameter section⟩}*) ;
⟨formal parameter section⟩ ::=
  ⟨parameter group⟩ |
  const ⟨parameter group⟩ {;⟨parameter group⟩}* |
  var ⟨parameter group⟩ {;⟨parameter group⟩}* |
  function ⟨parameter group⟩ |
  procedure ⟨identifier⟩ {,⟨identifier⟩}*
⟨parameter group⟩ ::= ⟨identifier⟩ {,⟨identifier⟩}*: ⟨type identifier⟩

A parameter group without preceding specifier implies constant parameters.

The constant definition part contains all constant synonym definitions local to the procedure.

⟨constant definition part⟩ ::= ⟨empty⟩ |
  const ⟨constant definition⟩ {,⟨constant definition⟩}*;

المنارة للاستشارات

The type definition part contains all type definitions which are local to the procedure declaration.

```
⟨type definition part⟩ ::= ⟨empty⟩ |
    type ⟨type definition⟩ {;⟨type definition⟩}*;
```

The variable declaration part contains all variable declarations local to the procedure declaration.

```
⟨variable declaration part⟩ ::= ⟨empty⟩ |
    var ⟨variable declaration⟩ {;⟨variable declaration⟩}*;
```

The procedure and function declaration part contains all procedure and function declarations local to the procedure declaration.

```
⟨procedure and function declaration part⟩ ::=
    {⟨procedure or function declaration⟩ ;}*
⟨procedure or function declaration⟩ ::=
    ⟨procedure declaration⟩ | ⟨function declaration⟩
```

The statement part specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

```
⟨statement part⟩ ::= ⟨compound statement⟩
```

All identifiers introduced in the formal parameter part, the constant definition part, the type definition part, the variable-, procedure or function declaration parts are *local* to the procedure declaration which is called the *scope* of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

```
procedure readinteger (var x: integer);
    var i, j: integer;
begin i := 0;
    while (input↑ ≧ '0') ∧ (input↑ ≦ '9') do
    begin j := int(input↑) − int('0');
          i := i * 10 + j;
          get(input)
    end;
    x := i
end

procedure Bisect (function f: real; const low, high: real;
    var, zero: real; p: Boolean);
    var a, b, m: real;
begin a := low; b := high;
    if (f(a) ≧ 0) ∨ (f(b) ≦ 0) then p := false else
```

المنارة للاستشارات

```
    begin p := true;
        while abs(a - b) > eps do
        begin m := (a + b)/2;
            if f(m) > 0 then b := m else a := m
        end;
        zero := a
    end
end
```

```
procedure GCD(m, n: integer; var, x, y, z: integer); {m ≧ 0, n > 0}
var a1, a2, b1, b2, c, d, q, r: integer;
begin{Greatest Common Divisor x of m and n,
      Extended Euclid's Algorithm, cf. [4], p. 14}
    c := m; d := n;
    a1 := 0;  a2 := 1;  b1 := 1;  b2 := 0;
    while d ≠ 0 do
    begin{a1*m + b1*n = d, a2*m + n2*n = c,
          gcd(c, d) = gcd(m, n)}
          q := c div d; r := c mod d;
          {c = q*d + r, gcd(d, r) = gcd(m, n)}
          a2 := a2 - q*a1; b2 := b2 - q*b1;
          {a2*m + b2*n = r, a1*m + b1*n = d}
          c := d;  d := r;
          r := a1;  a1 := a2;  a2 := r;
          r := b1;  b1 := b2;  b2 := r;
          {a1*m + b1*n = d, a2*m + b2*n = c,
          gcd(c, d) = gcd(m, n)}
    end;
    {gcd(c, 0) = c = gcd(m, n)}
    x := c;  y := a2;  z := b2
    {x = gcd(m, n), y*m + z*n = gcd(m, n)}
end
```

10.1. Standard Procedures

Standard procedures are supposed to be predeclared in every implementation of Pascal. Any implementation may feature additional predeclared procedures. Since they are, as all standard quantities, assumed as declared in a scope surrounding the Pascal program, no conflict arises form a declaration redefining the same identifier within the program. The standard procedures are listed and explained below.

10.1.1. File Positioning Procedures

put(*f*) advances the file pointer of file *f* to the next file component. It is only applicable, if the file is either in the output or in the neutral mode. The file is put into the output mode.

المنارة للاستشارات

get(*f*) advances the file pointer of file *f* to the next file component. It is only applicable, if the file is either in the input or in the neutral mode. If there does not exist a next file component, the end-of-file condition arises, the value of the variable denoted by $f\uparrow$ becomes undefined, and the file is put into the neutral mode.

reset(*f*) the file pointer of file *f* is reset to its beginning, and the file is put into the neutral mode.

10.1.2. Class Component Allocation Procedure

alloc(*p*) allocates a new component in the class to which the pointer variable *p* is bound, and assigns the pointer designating the new component to *p*. If the component type is a record type with variants, the form

alloc(*p*, *t*) can be used to allocate a component of the variant whose tag field value is *t*. However, this allocation does not imply an assignment to the tag field. If the class is already compleately allocated, the value **nil** will be assigned to *p*.

10.1.3. Data Transfer Procedures

Assuming that *a* is a character array variable, *z* is an alfa variable, and *i* is an integer expression, then

pack(*a*, *i*, *z*) packs the *n* characters $a[i] \dots a[i+n-1]$ into the alfa variable *z* (for *n* cf. 6.1.1.), and

unpack(*z*, *a*, *i*) unpacks the alfa value *z* into the variables $a[i] \dots a[i+n-1]$.

11. Function Declarations

Function declarations serve to define parts of the program which compute a scalar value or a pointer value. Functions are activated by the evaluation of a function designator (cf. 8.2) which is a constituent of an expression. A function declaration consists of the following parts, any of which, except the first and the last, may be empty (cf. also 10.).

⟨function declaration⟩ ::=
    ⟨function heading⟩
    ⟨constant definition part⟩ ⟨type definition part⟩
    ⟨variable declaration part⟩
    ⟨procedure and function declaration part⟩ ⟨statement part⟩

The function heading specifies the identifier naming the function, the formal parameters of the function (note that there must be at least one parameter), and the type of the (result of the) function.

⟨function heading ::= **function** ⟨identifier⟩ (⟨formal parameter section⟩
    {; ⟨formal parameter section⟩}*) : ⟨result type⟩ ;

⟨result type⟩ ::= ⟨type identifier⟩

المنارة للاستشارات

The type of the function must be a scalar or a subrange type or a pointer type. Within the function declaration there must be at least one assignment statement assigning a value to the function identifier. This assignment determines the result of the function. Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function. Within the statement part no assignment must occur to any variable which is not local to the function. This rule also excludes assignments to parameters.

Examples:

```
function Sqrt(x: real): real;
    var x0, x1: real;
begin x1 := x;  {x > 1, Newton's method}
    repeat x0 := x1;  x1 := (x0 + x/x0)*0.5
        {x0² − 2*x1*x0 + x = 0}
    until abs(x1 − x0) ≤ eps;
    {(x0 − eps) ≤ x1 ≤ (x0 + eps),
    (x − 2*eps*x0) ≤ x0² ≤ (x + 2*eps*x0)}
    Sqrt := x0
end

function Max(a: vector; n: integer): real;
    var x: real;  i: integer;
begin x := a[1];
    for i := 2 to n do
    begin {x = max(a₁ ... aᵢ₋₁)}
        if x < a[i] then x := a[i]
        {x = max(a₁ ... aᵢ)}
    end;
    {x = max(a₁ ... aₙ)}
    Max := x
end

function GCD(m, n: integer): integer;
begin if n = 0 then GCD := m else GCD := GCD(n, m mod n)
end

function Power(x: real; y: integer): real;  {y ≥ 0}
    var w, z: real;  i: integer;
begin w := x;  z := 1;  i := y;
    while i ≠ 0 do
    begin {z*w^i = x^y}
        if odd(i) then z := z*w;
        i := i div 2;  {z*w^2i = x^y}
        w := sqr(w)  {z*w^i = x^y}
    end;
    {i = 0, z = x^y}
    Power := z
end
```

المنارة للاستشارات

11.1. Standard Functions

Standard functions are supposed to be predeclared in every implementation of Pascal. Any implementation may feature additional predeclared functions (cf. also 10.1.).

The standard functions are listed and explained below:

11.1.1. Arithmetic Functions

abs (x) computes the absolute value of x. The type of x must be either *real* or *integer*, and the type of the result is the type of x.

sqr (x) computes x^2. The type of x must be either *real* or *integer*, and the type of the result is the type of x.

sin (x)
cos (x)
exp (x)
ln (x)
sqrt (x)
arctan (x) — the type of x must be either *real* or *integer*, and the type of the result is *real*

11.1.2. Predicates

odd (x) the type of x must be *integer*, and the result is x **mod** $2 = 1$

eof (f) indicates, whether the file f is in the end-of-file status.

11.1.3. Transfer Functions

trunc (x) x must be of type *real*, and the result is of type *integer*, such that $abs(x) - 1 < trunc(abs(x)) \leqq abs(x)$

int (x) x must be of type *char*, and the result (of type *integer*) is the ordinal number of the character x in the defined character set.

chr (x) x must be of type *integer*, and the result (of *type char*) is the character whose ordinal number is x.

11.1.4. Further Standard Functions

succ (x) x is of any scalar or subrange type, and the result is the successor value of x (if it exists).

pred (x) x is of any scalar or subrange type, and the result is the predecessor value of x (if it exists).

12. Programs

A Pascal program has the form of a procedure declaration without heading (cf. also 7.4.).

⟨program⟩ ::= ⟨constant definition part⟩⟨type definition part⟩
⟨variable declaration part⟩
⟨procedure and function declaration part⟩⟨statement part⟩.

المنارة للاستشارات

13. Pascal 6000

The version of the language Pascal which is processed by its implementation on the CDC 6000 series of computers is described by a number of amendments to the preceding Pascal language definition. The amendments specify extensions and restrictions and give precise definitions of certain standard data types. The section numbers used hereafter refer to the corresponding sections of the language definition.

3. Vocabulary

Only capital letters are available in the basic vocabulary of symbols. The symbol **eol** is added to the vocabulary. Symbols which consist of a sequence of underlined letters are called *word-delimiters*. They are written in Pascal 6000 without underlining and without any surrounding escape characters. Blanks or end-of-lines may be inserted anywhere except within :=, word-delimiters, identifiers, and numbers. The symbol $_{10}$ is written as '.

4. Identifiers

Only the 10 first symbols of an identifier are significant. Identifiers not differing in the 10 first symbols are considered as equal. Word-delimiters must not be used as identifiers. At least one blank space must be inserted between any two word-delimiters or between a word-delimiter and an adjacent identifier.

6. Data Types

6.1.1. Standard Scalar Types

integer is defined as
type *integer* $= -2^{48}+1 \,..\, 2^{48}-1$

real is defined according to the CDC 6000 floating point format specifications. Arithmetic operations on real type values imply rounding.

char is defined by the CDC 6000 display code character set. This set is incremented by the character denoted by **eol**, signifying end-of-line.

The ordered set is:

eol	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
J	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	*S*
T	*U*	*V*	*W*	*X*	*Y*	*Z*	0	1	2
3	4	5	6	7	8	9	+	−	*
/	(	)	$	=	␣	,	.	'	[
]	:	≠	{	∨	∧	↑	}	<	>
≤	≥	¬	;						

(Note that the characters ' { } are special features on the printers of the ETH installation, and correspond to the characters ≡ ⌈ ↓ at standard CDC systems.)

alfa the number *n* of characters packed into an alfa value is 10 (cf. 6.1.1.).

المنارة للاستشارات

6.2.3. Powerset Types

The base type of a powerset type must be either

1. a scalar type with less than 60 values, or

2. a subrange of the type *integer*, with a minimum element $min(T) \geqq 0$ and a maximum element $max(T) < 59$, or

3. a subrange of the type char with the maximum element $max(T) <$ '>'.

6.2.4. and 6.2.5. File and Class Types

No component of any structured type can be of a file type or of a class type.

7. Variable Declarations

File variables declared in the main program may be restricted to either input or output mode by appending the specifiers

[*in*] or [*out*]

to the file identifier in its declaration. Files restricted to input mode (input files) are expected to be Permanent Files attached to the job by the SCOPE Attach command, and files restricted to output mode may be catalogued as Permanent Files by the SCOPE Catalog command. In both commands, the file identifier is to be used as the Logical File Name [5].

10. and 11. Procedure and Function Declarations

A procedure or a function which contains local file declarations must not be activated recursively.

14. Glossary

actual parameter	9.1.2.
adding operator	8.1.3.
array type	6.2.1.
array variable	7.2.1.
assignment statement	9.1.1.
case label	6.2.2.
case list element	9.2.2.2.
case statement	9.2.2.2.
class type	6.2.5.
class variable	6.2.6.
component statement	9.2.1.
component type	6.2.1.
component variable	7.2.
compound statement	9.2.1.
conditional statement	9.2.2.
constant	5.
constant definition	5.
constant definition part	10.
control variable	9.2.3.3.
current file component	7.2.3.
digit	3.
entire variable	7.1.
expression	8.
factor	8.
field designator	7.2.2.
field identifier	7.2.2.
field list	6.2.2.
file type	6.2.4.
file variable	7.2.3.
final value	9.2.3.3.
fixed part	6.2.2.
for list	9.2.3.3.
for statement	9.2.3.3.
formal parameter section	10.
function declaration	11.
function designator	8.2.
function heading	11.
function identifier	8.2.
goto statement	9.1.3.
identifier	4.
if statement	9.2.2.1.
index type	6.2.1.
indexed variable	7.2.1.
initial value	9.2.3.3.
integer	4.
label	9.1.3.
label definition	9.2.1.
letter	3.
letter or digit	4.

المنارة للاستشارات

maxnum 6.2.5.
multiplying operator 8.1.2.
number 4.
parameter group 10.
pointer type 6.2.6.
pointer variable 7.2.4.
powerset type 6.2.3.
procedure and function declaration part 10.
procedure declaration 10.
procedure heading 10.
procedure identifier 9.1.2.
procedure or function declaration 10.
procedure statement 9.1.2.
program 12.
real number 4.
record section 6.2.2.
record type 6.2.2.
record variable 7.2.2.
referenced component 7.2.4.
relational operator 8.1.4.
repeat statement 9.2.3.2.
repetitive statement 9.2.3.
result type 11.
scalar type 6.1.
scale factor 4.
set 8.
sign 4.
simple expression 8.
simple statement 9.1.
special symbol 3.
statement 9.
statement part 10.
structured statement 9.2.
tag field 6.2.2.
term 8.
type 6.
type definition 6.
type definition part 10.
type identifier 6.
unsigned constant 5.
variable 7.
variable declaration 7.
variable declaration part 10.
variable identifier 7.1.
variant 6.2.2.
variant part 6.2.2.
with statement 9.2.4.
while statement 9.2.3.1.

The author gratefully acknowledges his indeptedness to C. A. R. Hoare for his many valuable suggestions concerning overall design strategy as well as details, and for his critical scrutiny of this paper.

References

1. Naur, P.: Report on the algorithmic language ALGOL 60. Comm ACM 3, 299–314 (1960).
2. Report on Subset ALGOL 60 (IFIP): Comm. ACM 7, 626–628 (1964).
3. Wirth, N., Hoare, C. A. R.: A contribution to the development of ALGOL. Comm. ACM 9, 413–432 (1966).
4. Knuth, D. E.: The art of computer programming, Vol. 1. Addison-Wesley 1968.
5. Control Data 6000 Computer Systems, SCOPE Reference Manual, Pub. No. 60189400.

Prof. Dr. N. Wirth
Eidgenössische Technische Hochschule
Fachgruppe Computer-Wissenschaften
Clausiusstraße 55
CH-8006 Zürich
Schweiz

المنارة للاستشارات

Niklaus Wirth

Program Development by Stepwise Refinement

Communications of the ACM, Vol. 14 (4), 1971
pp. 221–227

المنارة للاستشارات

Program Development by Stepwise Refinement

Niklaus Wirth
Eidgenössische Technische Hochschule
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction

CR Categories: 1.50, 4.0

1. Introduction

Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples. Unfortunately, they are too often selected with the prime intent to demonstrate what a computer can do. Instead, a main criterion for selection should be their suitability

to exhibit certain widely applicable *techniques*. Furthermore, examples of programs are commonly presented as finished "products" followed by explanations of their purpose and their linguistic details. But active programming consists of the design of *new* programs, rather than contemplation of old programs. As a consequence of these teaching methods, the student obtains the impression that programming consists mainly of mastering a language (with all the peculiarities and intricacies so abundant in modern PL's) and relying on one's intuition to somehow transform ideas into finished programs. Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual *development* can be nicely demonstrated.

This paper deals with a single example chosen with these two purposes in mind. Some well-known techniques are briefly demonstrated and motivated (strategy of preselection, stepwise construction of trial solutions, introduction of auxiliary data, recursion), and the program is gradually developed in a sequence of *refinement steps*.

In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. The result of the execution of a program is expressed in terms of data, and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, so the data may

المنارة للاستشارات

have to be refined, decomposed, or structured, and it is natural to *refine program and data specifications in parallel*.

Every refinement step implies some design decisions. It is important that these decision be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision. Subtrees may be considered as *families of solutions* with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This will result in programs which are easier to adapt to different environments (languages and computers), where different representations may be required.

The chosen sample problem is formulated at the beginning of section 3. The reader is strongly urged to try to find a solution by himself before embarking on the paper which—of course—presents only one of many possible solutions.

2. Notation

For the description of programs, a slightly augmented *Algol 60* notation will be used. In order to express

المنارة للاستشارات

repetition of statements in a more lucid way than by use of labels and jumps, a statement of the form

```
repeat ⟨statement sequence⟩
until ⟨Boolean expression⟩
```

is introduced, meaning that the statement sequence is to be repeated until the Boolean expression has obtained the value **true**.

3. The 8-Queens Problem and an Approach to Its Solution[1]

Given are an 8×8 chessboard and 8 queens which are hostile to each other. Find a position for each queen (a configuration) such that no queen may be taken by any other queen (i.e. such that every row, column, and diagonal contains at most one queen).

This problem is characteristic for the rather frequent situation where an analytical solution is not known, and where one has to resort to the method of trial and error. Typically, there exists a set A of candidates for solutions, among which one is to be selected which satisfies a certain condition p. Thus a solution is characterized as an x such that $(x \in A) \wedge p(x)$.

A straightforward program to find a solution is:

```
repeat Generate the next element of A and call it x
until p(x) ∨ (no more elements in A);
if p(x) then x = solution
```

The difficulty with this sort of problem usually is the sheer size of A, which forbids an exhaustive generation of candidates on the grounds of efficiency considera-

[1] This problem was investigated by C. F. Gauss in 1850.

المنارة للاستشارات

tions. In the present example, A consists of $64!/(56! \times 8!) \doteq 2^{32}$ elements (board configurations). Under the assumption that generation and test of each configuration consumes 100 μs, it would roughly take 7 hours to find a solution. It is obviously necessary to invent a "shortcut," a method which eliminates a large number of "obviously" disqualified contenders. This *strategy of preselection* is characterized as follows: Find a representation of p in the form $p = q \land r$. Then let $B_r = \{x \mid (x \in A) \land r(x)\}$. Obviously $B_r \subseteq A$. Instead of generating elements of A, only elements of B are produced and tested on condition q instead of p. Suitable candidates for a condition r are those which satisfy the following requirements:

1. B_r is much smaller than A.
2. Elements of B_r are easily generated.
3. Condition q is easier to test than condition p.

The corresponding program then is:

repeat Generate the next element of B and call it x
until $q(x) \lor$ (no more elements in B);
if $q(x)$ **then** x = solution

A suitable condition r in the 8-queens problem is the rule that in every column of the board there must be exactly one queen. Condition q then merely specifies that there be at most one queen in every row and in every diagonal, which is evidently somewhat easier to test than p. The set B_r (configurations with one queen in every column) contains "only" $8^8 = 2^{24}$ elements. They are generated by restricting the movement of queens to columns. Thus all of the above conditions are satisfied.

المنارة للاستشارات

Assuming again a time of 100 μs for the generation and test of a potential solution, finding a solution would now consume only 100 seconds. Having a powerful computer at one's disposal, one might easily be content with this gain in performance. If one is less fortunate and is forced to, say, solve the problem by hand, it would take 280 hours of generating and testing configurations at the rate of one per second. In this case it might pay to spend some time finding further shortcuts. Instead of applying the same method as before, another one is advocated here which is characterized as follows: Find a representation of trial solutions x of the form $[x_1, x_2, \cdots, x_n]$, such that every trial solution can be generated in steps which produce $[x_1]$, $[x_1, x_2]$, $\cdots$, $[x_1, x_2, \cdots, x_n]$ respectively. The decomposition must be such that:

1. Every step (generating x_j) must be considerably simpler to compute than the entire candidate x.
2. $q(x) \supset q(x_1 \cdots x_j)$ for all $j \leq n$.

Thus a full solution can never be obtained by extending a partial trial solution which does not satisfy the predicate q. On the other hand, however, a partial trial solution satisfying q may not be extensible into a complete solution. This method of *stepwise construction of trial solutions* therefore requires that trial solutions failing at step j may have to be "shortened" again in order to try different extensions. This technique is called *backtracking* and may generally be characterized by the program:

```
j := 1;
repeat trystep j;
  if successful then advance else regress
until (j < 1) ∨ (j > n)
```

المنارة للاستشارات

In the 8-queens example, a solution can be constructed by positioning queens in successive columns starting with column 1 and adding a queen in the next column in each step. Obviously, a partial configuration not satisfying the mutual nonaggression condition may never be extended by this method into a full solution. Also, since during the jth step only j queens have to be considered and tested for mutual nonaggression, finding a partial solution at step j requires less effort of inspection than finding a complete solution under the condition that all 8 queens are on the board all the time. Both stated criteria are therefore satisfied by the decomposition in which step j consists of finding a safe position for the queen in the jth column.

The program subsequently to be developed is based on this method; it generates and tests 876 partial configurations before finding a complete solution. Assuming again that each generation and test (which is now more easily accomplished than before) consumes one second, the solution is found in 15 minutes, and with the computer taking 100 μs per step, in 0.09 seconds.

4. Development of the Program

We now formulate the stepwise generation of partial solutions to the 8-queens problem by the following first version of a program:

```
variable board, pointer, safe;
considerfirstcolumn;
repeat trycolumn;
   if safe then
   begin setqueen; considernextcolumn
   end else regress
until lastcoldone ∨ regressoutoffirstcol
```

المنارة للاستشارات

This program is composed of a set of more primitive instructions (or procedures) whose actions may be described as follows:

considerfirstcolumn. The problem essentially consists of inspecting the safety of squares. A pointer variable designates the currently inspected square. The column in which this square lies is called the currently inspected column. This procedure initializes the pointer to denote the first column.

trycolumn. Starting at the current square of inspection in the currently considered column, move down the column either until a safe square is found, in which case the Boolean variable *safe* is set to **true**, or until the last square is reached and is also unsafe, in which case the variable *safe* is set to **false.**

setqueen. A queen is positioned onto the last inspected square.

considernextcolumn. Advance to the next column and initialize its pointer of inspection.

regress. Regress to a column where it is possible to move the positioned queen further down, and remove the queens positioned in the columns over which regression takes place. (Note that we may have to regress over at most two columns. Why?)

The next step of program development was chosen to refine the descriptions of the instructions *trycolumn* and *regress* as follows:

```
procedure trycolumn;
repeat advancepointer; testsquare
until safe ∨ lastsquare

procedure regress;
  begin reconsiderpriorcolumn
    if ¬ regressoutoffirstcol then
    begin removequeen;
      if lastsquare then
      begin reconsiderpriorcolumn;
        if ¬ regressoutoffirstcol then
          removequeen
      end
    end
  end
```

المنارة للاستشارات

The program is expressed in terms of the instructions:

considerfirstcolumn
considernextcolumn
reconsiderpriorcolumn
advancepointer
testsquare (sets the variable *safe*)
setqueen
removequeen

and of the predicates:

lastsquare
lastcoldone
regressoutoffirstcol

In order to refine these instructions and predicates further in the direction of instructions and predicates available in common programming languages, it becomes necessary to express them in terms of data representable in those languages. A decision on how to represent the relevant facts in terms of data can therefore no longer be postponed. First priority in decision making is given to the problem of how to represent the positions of the queens and of the square being currently inspected.

The most straightforward solution (i.e. the one most closely reflecting a wooden chessboard occupied by marble pieces) is to introduce a Boolean square matrix with $B[i, j] =$ **true** denoting that square (i, j) is occupied. The success of an algorithm, however, depends almost always on a suitable choice of its data representation in the light of the ease in which this representation allows the necessary operations to be expressed. Apart from this, consideration regarding storage requirements may be of prime importance (although hardly in this case). A common difficulty in program design lies in the unfor-

المنارة للاستشارات

tunate fact that at the stage where decisions about data representations have to be made, it often is still difficult to foresee the details of the necessary instructions operating on the data, and often quite impossible to estimate the advantages of one possible representation over another. In general, it is therefore advisable to delay decisions about data representation as long as possible (but not until it becomes obvious that no realizable solution will suit the chosen algorithm).

In the problem presented here, it is fairly evident even at this stage that the following choice is more suitable than a Boolean matrix in terms of simplicity of later instructions as well as of storage economy.

j is the index of the currently inspected column; (x_j, j) is the coordinate of the last inspected square; and the position of the queen in column $k < j$ is given by the coordinate pair (x_k, k) of the board. Now the variable declarations for pointer and board are refined into:

integer j $(0 \leq j \leq 9)$
integer array $x[1:8]$ $(0 \leq x_i \leq 8)$

and the further refinements of some of the above instructions and predicates are expressed as:

procedure *considerfirstcolumn*;
  begin $j := 1$; $x[1] := 0$ **end**
procedure *considernextcolumn*;
  begin— $j := j+1$; $x[j] := 0$ **end**
procedure *reconsiderpriorcolumn*; $j := j-1$
procedure *advancepointer*;
  $x[j] := x[j] + 1$
Boolean procedure *lastsquare*;
  lastsquare $:= x[j] = 8$

المنارة للاستشارات

Boolean procedure *lastcoldone*;
  lastcoldone := $j > 8$
Boolean procedure *regressoutoffirstcol*;
  regressoutoffirstcol := $j < 1$

At this stage, the program is expressed in terms of the instructions:

testsquare
setqueen
removequeen

As a matter of fact, the instructions *setqueen* and *removequeen* may be regarded as vacuous, if we decide that the procedure *testsquare* is to determine the value of the variable *safe* solely on the grounds of the values $x_1 \cdots x_{j-1}$ which completely represent the positions of the $j - 1$ queens so far on the board. But unfortunately the instruction *testsquare* is the one most frequently executed, and it is therefore the one instruction where considerations of efficiency are not only justified but essential for a good solution of the problem. Evidently a version of *testsquare* expressed only in terms of $x_1 \cdots x_{j-1}$ is inefficient at best. It should be obvious that *testsquare* is executed far more often than *setqueen* and *removequeen*. The latter procedures are executed whenever the column (j) is changed (say m times), the former whenever a move to the next square is undertaken (i.e. x_j is changed, say n times). However, *setqueen* and *removequeen* are the only procedures which affect the chessboard. Efficiency may therefore be gained by the method of *introducing auxiliary variables* $V(x_1 \cdots x_j)$ such that:

1. Whether a square is safe can be computed more easily from $V(x)$ than from x directly (say in u

المنارة للاستشارات

units of computation instead of ku units of computation).

2. The computation of $V(x)$ from x (whenever x changes) is not too complicated (say of v units of computation).

The introduction of V is advantageous (apart from considerations of storage economy), if

$$n(k - 1)u > mu \qquad \text{or} \qquad \frac{n}{m}(k - 1) > \frac{v}{u},$$

i.e. if the gain is greater than the loss in computation units.

A most straightforward solution to obtain a simple version of *testsquare* is to introduce a Boolean matrix B such that $B[i, j] =$ **true** signifies that square (i, j) is not taken by another queen. But unfortunately, its recomputation whenever a new queen is removed (v) is prohibitive (why?) and will more than outweigh the gain.

The realization that the relevant condition for safety of a square is that the square must lie neither in a row nor in a diagonal already occupied by another queen, leads to a much more economic choice of V. We introduce Boolean arrays a, b, c with the meanings:

$a_k =$ **true** : no queen is positioned in row k
$b_k =$ **true** : no queen is positioned in the /-diagonal k
$c_k =$ **true** : no queen is positioned in the \-diagonal k

The choice of the index ranges of these arrays is made in view of the fact that squares with equal sum of their coordinates lie on the same /-diagonal, and those with equal difference lie on the same \-diagonal.

المنارة للاستشارات

With row and column indices from 1 to 8, we obtain:

Boolean array $a[1:8]$, $b[2:16]$, $c[-7:7]$

Upon every introduction of auxiliary data, care has to be taken of their *correct initialization.* Since our algorithm starts with an empty chessboard, this fact must be represented by initially assigning the value **true** to all components of the arrays a, b, and c. We can now write:

```
procedure testsquare;
  safe := a[x[j]] ∧ b[j+x[j]] ∧ c[j−x[j]]
procedure setqueen;
  a[x[j]] := b[j+x[j]] := x[j−x[j]] := false
procedure removequeen;
  a[x[j]] := b[j+x[j]] := c[j−x[j]] := true
```

The correctness of the latter procedure is based on the fact that each queen currently on the board had been positioned on a safe square, and that all queens positioned after the one to be removed now had already been removed. Thus the square to be vacated becomes safe again.

A critical examination of the program obtained so far reveals that the variable $x[j]$ occurs very often, and is not taken by another queen. But unfortunately, its recomputation whenever a new queen is removed (v) is prohibitive (why?) and will more than outweigh the gain.

The realization that the relevant condition for safety of a square is that the square must lie neither in a row nor in a diagonal already occupied by another queen, leads to a much more economic choice of V. We introduce Boolean arrays a, b, c with the meanings:

المنـــارة للاستشارات

a_k = **true** : no queen is positioned in row k
b_k = **true** : no queen is positioned in the /-diagonal k
c_k = **true** : no queen is positioned in the \-diagonal k

The choice of the index ranges of these arrays is made in view of the fact that squares with equal sum of their coordinates lie on the same /-diagonal, and those with equal difference lie on the same \-diagonal. With row and column indices from 1 to 8, we obtain:

```
Boolean array a[1:8], b[2:16], c[-7:7]
```

Upon every introduction of auxiliary data, care has to be taken of their *correct initialization*. Since our algorithm starts with an empty chessboard, this fact must be represented by initially assigning the value **true** to all components of the arrays a, b, and c. We can now write:

```
procedure testsquare;
  safe := a[x[j]] ∧ b[j+x[j]] ∧ c[j−x[j]]
procedure setqueen;
  a[x[j]] := b[j+x[j]] := x[j−x[j]] := false
procedure removequeen;
  a[x[j]] := b[j+x[j]] := c[j−x[j]] := true
```

The correctness of the latter procedure is based on the fact that each queen currently on the board had been positioned on a safe square, and that all queens positioned after the one to be removed now had already been removed. Thus the square to be vacated becomes safe again.

A critical examination of the program obtained so far reveals that the variable $x[j]$ occurs very often, and in particular at those places of the program which are also executed most often. Moreover, examination of

المنارة للاستشارات

$x[j]$ occurs much more frequently than reassignment of values to j. As a consequence, the principle of introduction of auxiliary data can again be applied to increase efficiency: a new variable

integer *i*

is used to represent the value so far denoted by $x[j]$. Consequently $x[j] := i$ must always be executed before j is increased, and $i := x[j]$ after j is decreased. This final step of program development leads to the reformulation of some of the above procedures as follows:

```
procedure testsquare;
  safe := a[i] ∧ b[i+j] ∧ c[i−j]
procedure setqueen;
  a[i] := b[i+j] := c[i−j] := false
procedure removequeen;
  a[i] := b[i+j] := c[i−j] := true
procedure considerfirstcolumn;
  begin j := 1;   i := 0 end
procedure advancepointer;   i := i+1
procedure considernextcolumn;
  begin x[j] := i;   j := j+1;   i := 0 end
Boolean procedure lastsquare;
  lastsquare := i = 8
```

The final program, using the procedures

testsquare
setqueen
regress
removequeen

and with the other procedures directly substituted, now has the form

```
j := 1;   i := 0;
repeat
  repeat i := i+1;   testsquare
```

المنارة للاستشارات

```
      until safe ∨ (i=8);
      if safe then
      begin setqueen;   x[j] := i;   j := j+1;   i := 0
      end else regress
   until (j > 8) ∨ (j < 1);
   if j > 8 then PRINT(x) else FAILURE
```

It is noteworthy that this program still displays the structure of the version designed in the first step. Naturally other, equally valid solutions can be suggested and be developed by the same method of stepwise program refinement. It is particularly essential to demonstrate this fact to students. One alternative solution was suggested to the author by E. W. Dijkstra. It is based on the view that the problem consists of a stepwise extension of the board by one column containing a safely positioned queen, starting with a null-board and terminating with 8 columns. The process of extending the board is formulated as a procedure, and the natural method to obtain a complete board is by *recursion* of this procedure. It can easily be composed of the same set of more primitive instructions which were used in the first solution.

```
procedure Trycolumn(j);
   begin integer i;   i := 0;
      repeat i := i+1;   testsquare;
         if safe then
         begin setqueen;   x[j] := i;
            if j < 8 then Trycolumn (j+1);
            if ¬ safe then removequeen
         end
      until safe ∨ (i=8)
   end
```

The program using this procedure then is

```
Trycolumn(1);
if safe then PRINT(x) else FAILURE
```

المنارة للاستشارات

(Note that due to the introduction of the variable *i* local to the recursive procedure, every column has its own pointer of inspection *i*. As a consequence, the procedures

testsquare
setqueen
removequeen

must be declared locally within *Trycolumn* too, because they refer to the *i* designating the scanned square in the *current* column.)

5. The Generalized 8-Queens Problem

In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever. Usually its users discover sooner or later that their program does not deliver all the desired results, or worse, that the results requested were not the ones really needed. Then either an extension or a change of the program is called for, and it is in this case where the method of stepwise program design and systematic structuring is most valuable and advantageous. If the structure and the program components were well chosen, then often many of the constituent instructions can be adopted unchanged. Thereby the effort of redesign and reverification may be drastically reduced. As a matter of fact, the *adaptability* of a program to changes in its objectives (often called maintainability) and to changes in its environment (nowadays called portability) can be measured primarily in terms of the degree to which it is neatly structured.

المنارة للاستشارات

It is the purpose of the subsequent section to demonstrate this advantage in view of a generalization of the original 8-queens problem and its solution through an extension of the program components introduced before.

The generalized problem is formulated as follows:

> Find *all* possible configurations of 8 hostile queens on an 8 × 8 chessboard, such that no queen may be taken by any other queen.

The new problem essentially consists of two parts:

1. Finding a method to generate further solutions.
2. Determining whether all solutions were generated or not.

It is evidently necessary to generate and test candidates for solutions in some *systematic manner.* A common technique is to find an *ordering of candidates* and a condition to identify the last candidate. If an ordering is found, the solutions can be mapped onto the integers. A condition limiting the numeric values associated with the solutions then yields a criterion for termination of the algorithm, if the chosen method generates solutions strictly in increasing order.

It is easy to find orderings of solutions for the present problem. We choose for convenience the mapping

$$M(x) = \sum_{j=1}^{8} x_j 10^{j-1}$$

An upper bound for possible solutions is then

$$M(x_{\max}) = 88888888$$

and the "convenience" lies in the circumstance that our earlier program generating one solution generates the

المنارة للاستشارات

minimum solution which can be regarded as the starting point from which to proceed to the next solution. This is due to the chosen method of testing squares strictly proceeding in increasing order of $M(x)$ starting with 00000000. The method for generating further solutions must now be chosen such that starting with the configuration of a given solution, scanning proceeds in the same order of increasing M, until either the next higher solution is found or the limit is reached.

6. The Extended Program

The technique of extending the two given programs finding a solution to the simple 8-queens problem is based on the idea of modification of the global structure only, and of using the same building blocks. The global structure must be changed such that upon finding a solution the algorithm will produce an appropriate indication—e.g. by printing the solution—and then proceed to find the next solution until it is found or the limit is reached. A simple condition for reaching the limit is the event when the first queen is moved beyond row 8, in which case regression out of the first column will take place. These deliberations lead to the following modified version of the nonrecursive program:

```
considerfirstcolumn;
  repeat trycolumn;
    if safe then
    begin setqueen;   considernextcolumn;
      if lastcoldone then
      begin PRINT(x); regress
      end
    end else regress
  until regressoutoffirstcol
```

Indication of a solution being found by printing it now occurs directly at the level of detection, i.e. before leaving the repetition clause. Then the algorithm proceeds to find a next solution whereby a shortcut is used by directly regressing to the prior column; since a solution places one queen in each row, there is no point in further moving the last queen within the eighth column.

The recursive program is extended with even greater ease following the same considerations:

```
procedure Trycolumn(j);
begin integer i;
  ⟨declarations of procedures testsquare, advancequeen,
  setqueen, removequeen, lastsquare⟩
  i := 0;
  repeat advancequeen;   testsquare;
    if safe then
    begin setqueen;   x[j] := i;
      if ¬ lastcoldone then Trycolumn(j+1) else PRINT(x);
      removequeen
    end
  until lastsquare
end
```

The main program starting the algorithm then consists (apart from initialization of a, b, and c) of the single statement *Trycolumn*(1).

In concluding, it should be noted that both programs represent the same algorithm. Both determine 92 solutions in the *same* order by testing squares 15720 times. This yields an average of 171 tests per solution; the maximum is 876 tests for finding a next solution (the first one), and the minimum is 8. (Both programs coded in the language Pascal were executed by a CDC 6400 computer in less than one second.)

المنارة للاستشارات

7. Conclusions

The lessons which the described example was supposed to illustrate can be summarized by the following points.

1. Program construction consists of a sequence of *refinement steps*. In each step a given task is broken up into a number of subtasks. Each refinement in the description of a task may be accompanied by a refinement of the description of the data which constitute the means of communication between the subtasks. Refinement of the description of program and data structures should proceed in parallel.

2. The degree of *modularity* obtained in this way will determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is executed.

3. During the process of stepwise refinement, a *notation* which is natural to the problem in hand should be used as long as possible. The direction in which the notation develops during the process of refinement is determined by the language in which the program must ultimately be specified, i.e. with which the notation ultimately becomes identical. This language should therefore allow us to express as naturally and clearly as possible the structures of program and data which emerge during the design process. At the same time, it must give guidance in the refinement process by exhibiting those basic features and structuring principles which are natural to the machine by which programs are supposed to be executed. It is remarkable that it would be difficult to find a language that would meet these important require-

المنارة للاستشارات

ments to a lesser degree than the one language still used most widely in teaching programming: Fortran.

4. Each refinement implies a number of *design decisions* based upon a set of design criteria. Among these criteria are efficiency, storage economy, clarity, and regularity of structure. Students must be taught to be conscious of the involved decisions and to critically examine and to reject solutions, sometimes even if they are correct as far as the result is concerned; they must learn to weigh the various aspects of design alternatives in the light of these criteria. In particular, they must be taught to revoke earlier decisions, and to back up, if necessary even to the top. Relatively short sample problems will often suffice to illustrate this important point; it is not necessary to construct an operating system for this purpose.

5. The detailed elaborations on the development of even a short program form a long story, indicating that careful programming is not a trivial subject. If this paper has helped to dispel the widespread belief that programming is easy as long as the programming language is powerful enough and the available computer is fast enough, then it has achieved one of its purposes.

Acknowledgments. The author gratefully acknowledges the helpful and stimulating influence of many discussions with C.A.R. Hoare and E. W. Dijkstra.

المنارة للاستشارات

References

The following articles are listed for further reference on the subject of programming.

1. Dijkstra, E. W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
2. Dijkstra, E. W. Notes on structured programming. EWD 249, Technical U. Eindhoven, The Netherlands, 1969.
3. Naur, P. Programming by action clusters. *BIT 9* (1969) 250–258.
4. Wirth, N. Programming and programming languages. Proc. Internat. Comput. Symp., Bonn, Germany, May 1970.

المنارة للاستشارات