Quaestiones Informaticae

Vol. 2 No. 2 May, 1983

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Circulation manager
Mr E Anderssen
Department of Computer Science
Rand Afrikaanse Universiteit
P O Box 524
Johannesburg 2000
Tel.: (011) 726-5000

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.

NOTE FROM THE EDITOR

Three points must be made by way of introduction to the second issue of Volume 2 of Quaestiones Informaticae.

Firstly, an apology is in order for the mistake in the date (November 1983 instead of 1982) at the foot of my note introducing the preceding issue. Lacking the services of a professional proof reader, printing errors are bound to show up from time to time, but it is hoped that their number will be kept to a minimum!

Secondly, it is a pleasure to announce that this journal will not only serve to publish papers of a scientific or technical nature on computing matters under the auspices of the Computer Society of South Africa. An agreement has been reached to share the facilities of Quaestiones Informaticae between the CSSA and SAICS, the South African Institute of Computer Scientists. Henceforth this journal will also be used to publish the Transactions of this Institute. This implies certain changes to the cover pages which will be implemented in future issues. I shall continue to serve as editor, but on behalf of SAICS Prof R. J. van den Heever will share some of my duties and act as co-editor.

Finally Mr Edwin Anderssen, of Rand Afrikaanse Universiteit, has agreed to serve as circulation manager for Quaestiones Informaticae. I am grateful indeed that he is willing to serve the journal in this capacity, and look forward to a long period of fruitful cooperation.

G WIECHERS

May, 1983

Die Operasionele Enkelbedienermodel

J.C. van Niekerk
Sperry, Johannesburg

Abstract
Die operasionele benadering tot die ontleding van die enkelbedienermodel word bespreek. Daar word gekyk na Buzen se operasionele ekwivalent vir die stogastiese geboorte/sterfte Markov-model, waarna veralgemenings van die model ondersoek word. Ten slotte word prestasiemate vir die modelle afgelei.

The operational approach to the analysis of the single server model is discusses. Firstly Buzen's operational equivalent to the stochastic birth/death Markov model is presented, whereafter generalisations of the model are examined. Lastly, the performance indicators are derived for these models.

Inleiding

In die laaste paar jare, in die besonder vanaf 1977, is 'n metode vir prestasie-ontleding ontwikkel wat grootliks ekwivalent is aan die stogastiese benadering, maar nie onder dieselfde nadele gebuk gaan nie. Die hoofdoel van hierdie metode is, naamlik, om van meetbare veranderlikes en bewysbare aannames gebruik te maak om die prestasie van 'n stelsel te ondersoek. Die groot klag teen die stogastiese metode is dan ook dat die gemiddelde prestasie-analis in die praktyk meestal nie oor die statistiese kennis beskik om dit ten volle te begryp nie. Daarby is die aannames wat gemaak moet word geensins in eindige tyd verifieerbaar nie. Alhoewel verifieeringseksperimente getoon het dat die meeste resultate wat bereken word wel baie goed vergelyk met die werklike gemete waardes, is daar geen manier waarop bewys kan word dat 'n nuwe stelsel onder beskouing ook op hierdie manier akkuraat ontleed kan word nie. Die prestasie-analis verkies dus dikwels om die meer bekende en makliker begrypbare metodes soos normtoetse te gebruik, alhoewel die metodes op sigself nie so akkuraat is nie [6].

Tot op hede is die twee belangrikste tipes operasionele modelle wat ontwikkel is die enkelbediener en die toustaannetwerkmodelle. Die enkelbedienermodel word gebruik om enkele komponente van 'n stelsel te ontleed. Die kompleksiteit van moderne stelsels maak dit egter feitlik onmoontlik om die hele stelsel as 'n enkelbediener te beskou. Sulke stelsels bestaan uit netwerke van enkelbedieners wat interafhanklik is en dus met meer gesofistikeerde metodes soos toustaannetwerkmodelle ontleed moet word [3,4,5]. 'n Deelversameling van hierdie toustaannetwerkmodelle wat aan 'n sekere aantal voorwaardes voldoen, reduseer egter na modelle waarin elke komponent as 'n aparte enkelbediener beskou word. Die resultate van die enkelbedienermodel is dan direk daarop van toepassing. Vir meer besonderhede sien [3,4,5,7]. In hierdie artikel word die bespreking beperk tot die enkelbedienermodel.

In [2] het Buzen aangetoon hoe die operasionele formules, wat ekwivalent is aan die stogastiese geboorte/sterfte Markovformules, afgelei word en welke aannames in die proses gemaak moet word. In hierdie artikel word aangetoon hoe akkurate formules afgelei kan word vir modelle wat slegs aan een van die twee aannames voldoen.

Die Model

Die model bestaan uit die volgende drie komponente:

1. 'n Bediener wat een taak op 'n slag verwerk en slegs ledig is as daar geen take vir bediening wag nie. Die diensspoed is afhanklik van die aantal take wat vir verwerking wag.
2. Een wagtou met 'n maksimum lengte.
3. Take wat teen 'n sekere aankomstempo by die bediener aankom. Die aankomstempo is ook afhanklik van die toulengte. Nadat die tou sy maksimum lengte bereik, word alle verdere take weggewys.

Die Stelseltoestand

Die toestand N(t) van die stelsel op tydstip t word beskryf deur die aantal take wat op daardie tydstip in die stelsel teenwoordig is. 'n Aankoms of vertrek wat op tydstip t plaasvind, word nie by N(t) in berekening gebring nie. N(t) se waarde bly dus dieselfde as net voor die aankoms of vertrek.

Die Enkelbedienermodel

Die volgende veranderlikes word benodig:

Meetbare Operasionele Veranderlikes

T — lengte van die tydsperiode waarin die stelsel waargeneem word.
M — maksimum aantal take in die stelsel op enige tydstip gedurende die tydsperiode.
m — minimum aantal take in die stelsel op enige tydstip gedurende die tydsperiode.
T(n) — tyd wat die stelsel gedurende die waarnemingsperiode in toestand n deurbring, $n = m, m+1, \ldots, M$.
A(n) — aantal aankomstes terwyl die stelsel in toestand n is, $n = m, m+1, \ldots, M-1$.
C(n) — aantal vertrekke terwyl die stelsel in toestand n is, $n = m+1, m+2, \ldots, M$.

Afgeleide Operasionele Veranderlikes

P(n) — breukdeel van die totale tyd wat die stelsel in toestand n deurbring.

$$P(n) = \frac{T(n)}{T} \quad n = m, m+1, \ldots, M$$

$$\text{Omdat } \sum_{n=m}^{M} T(n) = T \text{ volg dat } \sum_{n=m}^{M} P(n) = 1$$

$\lambda(n)$ — gemiddelde aankomstempo terwyl die stelsel in toestand n is.

$$\lambda(n) = \frac{A(n)}{T(n)} \quad n = m, m+1, \ldots, M-1$$

$$\lambda(M) = 0$$

$\mu(n)$ — gemiddelde vertrektempo terwyl die stelsel in toestand n is

$$\mu(n) = \frac{C(n)}{T(n)} \quad n = m+1, m+2, \ldots, M$$

$$\mu(m) = 0$$

Die Operasionele Geboorte/Sterfte-model [2]

Die volgende twee verifieerbare aannames word gemaak in die operasionele afleiding van die uitdrukkings vir $P(m), P(m+1), \ldots, P(M)$.

Aanname 1 — Basiese Operasionele Ewewig

Die stelseltoestand aan die begin en einde van die tydsinterval moet dieselfde wees,

$$N(0+\epsilon) = N(T) \quad \text{vir 'n baie klein } \epsilon > 0.$$

Aanname 2 — Enkel Aankomstes/Vertrekke

Geen aankoms en/of vertrek vind gelyktydig plaas nie.

Aanname 2 verseker dat die stelsel vanuit toestand n slegs na toestand n+1 of n-1 kan gaan. Verder geld dat vir elke oorgang van toestand n na n+1 'n oorgang van n+1 na n op 'n ander tydstip moet plaasvind (aanname 1 en 2).

Dus: $A(n) = C(n+1) \quad n = m, m+1,\ldots,M-1$

Hieruit volg: $\lambda(n) \cdot T(n) = \mu(n+1) \cdot T(n+1)$

$$\lambda(n) \cdot P(n) \cdot T = \mu(n+1) \cdot P(n+1) \cdot T$$

$$\text{Dus } P(n+1) = P(n)\frac{\lambda(n)}{\mu(n+1)}$$

Rekursiewe toepassing van die formule lewer

$$P(n) = P(m) \quad \frac{\lambda(m)\lambda(m+1)\ldots\lambda(n-1)}{\mu(m+1)\mu(m+2)\ldots\mu(n)} \quad n = m+1, m+2,\ldots,M \quad (1.1)$$

$$\text{Omdat } \sum_{n=m}^{M} P(n) = 1 \text{ volg}$$

$$P(m) = \left(1 + \sum_{n=m+1}^{M} \frac{\lambda(m)\lambda(m+1)\ldots\lambda(n-1)}{\mu(m+1)\mu(m+2)\ldots\mu(n)}\right)^{-1} \quad (1.2)$$

Hierdie formule is ekwivalent aan dié van die stogastiese geboorte/sterfte Markov-model, waarin die waardes wat die stelseltoestande kan aanneem beperk word tot $n = m, m+1,\ldots,M$. Om die stogastiese formules af te lei, moet egter die volgende aannames gemaak word.

1) Die tussenaankoms- en dienstye is toestandafhanklik eksponensieel verdeel.
2) Die stelsel is in statistiese ewewig.

Aanname 1 stem ooreen met die aanname van enkel aankomstes/vertrekke by die operasionele model en aanname 2 met die aanname van basiese operasionele ewewig. Die nadeel is dat hierdie stogastiese aannames nie in eindige tyd geverifieer kan word nie.

'n Verdere probleem by die stogastiese metode is die bepaling van waardes vir die parameters $\lambda(n)$, $n = m, m+1,\ldots,M-1$ en $\mu(n)$, $n = m+1, m+2,\ldots,M$. Beide $\lambda(n)$ en $\mu(n)$ word gewoonlik op dieselfde wyse as by die operasionele model bereken. Volgens die Wet van Groot Getalle sal die waargenome $\lambda(n)$- en $\mu(n)$- waardes in die limiet met waarskynlikheid een respektiewelik na die stogastiese parameters $\lambda(n)$ en $\mu(n)$ nader. Deur die stelsel dus slegs vir 'n eindige periode waar te neem, is die waardes wat vir die parameters bereken word nie noodwendig akkuraat nie. Die afleiding van die formules word in meeste boeke wat elementêre toustaanteorie bespreek, gegee.

Geboorte/Sterfte-formules Sonder Basiese Operasionele Ewewig

Gebruikmakend van die operasionele benadering kan akkurate formules afgelei word vir modelle waarin slegs die enkel aankomstes/vertrekke-aanname geld en nie basiese operasionele ewewig nie. Hulle kan gebruik word indien waarnemings gemaak is oor tydsperiode T waarvoor nie geld dat die begin- en eindtoestande dieselfde is nie en daar ook nie rekord gehou is van die tydstippe waarop die stelsel se toestand verander het nie. Die formules is egter heelwat komplekser as vir modelle waarin die aanname wel geld. Indien daar relatief baie waarnemings gemaak is tydens die waarnemingsperiode, nader die waardes bereken deur hierdie formules na dié bereken uit die model waarin die aanname wel gemaak word.

Gestel die begintoestand is a en die eindtoestand b. Sonder verlies aan algemeenheid kan aanvaar word dat $a < b$.

Die volgende vergelykings kan direk neergeskryf word:

$$A(k) = C(k+1) \qquad k = m, m+1,\ldots,a-1,b,b+1,\ldots,M-1$$

$$A(k) = C(k+1) + 1 \quad k = a, a+1,\ldots,b-1$$

Hieruit volg:

$$\lambda(k) \cdot T(k) = \mu(k+1) \cdot T(k+1) \qquad k = m, m+1,\ldots,a-1,b,b+1,\ldots,M-1$$

$$\text{en } \lambda(k) \cdot T(k) = \mu(k+1) \cdot T(k+1) + 1 \qquad k = a, a+1,\ldots,b-1$$

Gebruikmakend van $P(k) = \frac{T(k)}{T}$ volg:

$$P(k+1) = P(k)\frac{\lambda(k)}{\mu(k+1)} \quad k = m, m+1,\ldots,a-1,b,b+1,\ldots,M-1$$

$$\text{en } P(k+1) = P(k)\frac{\lambda(k)}{\mu(k+1)} - \frac{1}{T\mu(k+1)} \quad k = a, a+1,\ldots,b-1$$

Rekursiewe toepassing van die formule lewer:

$$m \le k \le a: \quad P(k) = P(m) \frac{\lambda(m)\lambda(m+1)\ldots\lambda(k-1)}{\mu(m+1)\mu(m+2)\ldots\mu(k)}$$

$$= P(m) \prod_{n=m}^{k-1} \frac{\lambda(n)}{\mu(n+1)}$$

$$a+1 \le k \le b: \quad P(k) = P(m) \prod_{n=m}^{k-1} \frac{\lambda(n)}{\mu(n+1)}$$

$$\frac{1}{T} \sum_{i=1}^{k-a} \left(\frac{1}{\mu(a+i)} \prod_{n=a+i}^{k-1} \frac{\lambda(n)}{\mu(n+1)}\right)$$

$$b+1 \le k \le M: \quad P(k) = P(m) \prod_{n=m}^{k-1} \frac{\lambda(n)}{\mu(n+1)} -$$

$$\frac{1}{T} \sum_{i=1}^{b-a} \left(\frac{1}{\mu(a+i)} \prod_{n=a+i}^{k-1} \frac{\lambda(n)}{\mu(n+1)}\right)$$

$$\text{Uit } \sum_{n=m}^{M} P(n) = 1 \text{ volg:}$$

$$P(m) = \left\{1 + \frac{1}{T} \sum_{k=a+1}^{b} \sum_{i=1}^{k-a} \left(\frac{1}{\mu(a+i)} \prod_{n=a+i}^{k-1} \frac{\lambda(n)}{\mu(n+1)}\right) + \frac{1}{T} \sum_{k=b+1}^{M} \sum_{i=1}^{b-a} \left(\frac{1}{\mu(a+i)} \prod_{n=a+i}^{k-1} \frac{\lambda(n)}{\mu(n+1)}\right)\right\} * \left[1 + \sum_{k=m+1}^{M} \prod_{n=m}^{k-1} \frac{\lambda(n)}{\mu(n+1)}\right]^{-1}$$

Indien aanvaar word dat $\mu(n)$, $\mu(n)$ en $P(m)$ in die limiet bestaan en groter as 0 is, kan limietverdelings vir hierdie uitdrukkings afgelei word.

Aanvaar

$$\lim_{T\to\infty} A(n) / T(n) = \lambda^*(n) > 0 \qquad n = m, m+1, \ldots, M-1$$

$$\lim_{T\to\infty} C(n) / T(n) = \mu^*(n) > 0 \qquad n = m+1, m+2, \ldots, M$$

$$\lim_{T\to\infty} T(m) / T = P^*(m) > 0$$

Dan volg:

$$\lim_{T\to\infty} P(k) = P^*(m) \prod_{n=m}^{k-1} \frac{\lambda^*(n)}{\mu^*(n+1)} \qquad k = m+1, m+2, \ldots, M \qquad (1.3)$$

en

$$\lim_{T\to\infty} P(m) = \left(1 + \sum_{k=m+1}^{M} \prod_{n=m}^{k-1} \frac{\lambda^*(n)}{\mu^*(n+1)}\right)^{-1} \qquad (1.4)$$

Hierdie formules is ekwivalent aan (1.1) en (1.2).

Limietgedrag van die geboorte/Sterfte-proses waarby beide aannames verswak word [2]

Deur beide die basiese operasionele ewewig en enkel aankomstes/vertrekke-aanname te verswak, kan formules afgelei word wat in die limiet ekwivalent is aan (1.3) en (1.4).

Aanname 1

Die aanname van basiese operasionele ewewig word vervang met die aannames wat ook in die vorige paragraaf gemaak is, naamlik, die bestaan van $\lambda(n)$, $\mu(n)$ en $P(m)$ in die limiet.

Aanname 2

Die aanname van enkel aankomstes/vertrekke word vervang met die aanname dat gelyktydige aankomstes en vertrekke kan plaasvind, onder die voorwaarde dat die verhouding tot T van die totale aantal aankomstes en vertrekke wat nie alleen plaasvind nie, na nul afneem as $T\to\infty$. Laat $q(n,i,j)$ die aantal herhalings van i aankomstes en j vertrekke wees wat gelyktydig plaasvind terwyl die stelsel in toestand n is
($i \geq 0$, $j \geq 0$ en $n = m, m+1, \ldots, M$).

Anders gestel:

$$\lim_{T\to\infty} \frac{1}{T} \sum_{n=m}^{M} \left\{ \prod_{i=1}^{\infty} \prod_{j=1}^{\infty} (i+j).q(n,i,j) + \sum_{i=2}^{M-n} i.q(n,i,0) + \sum_{j=2}^{M-n} j.q(n,0,j) \right\} = 0$$

$$\text{met } q(n,i,j) = 0 \text{ vir } i=j=n \text{ en } n+i-j > M$$

Die afleiding van die formules word gegee in [2].

Prestasiemate

Bedienerbenutting

Benutting word soos volg gedefinieer:

$U = \frac{B}{T}$ met B die tydsperiode waartydens die stelseltoestand groter as 0 is.

Die volgende twee gevalle kan onderskei word, naamlik,
1) $m \neq 0$, dan is $U = 1$
2) $m = 0$, dan geld

$$U = \frac{1}{T} \sum_{n=1}^{M} T(n) = \sum_{n=1}^{M} P(n) = 1 - P(0)$$

Deurvoertempo

Die gemiddelde deurvoertempo van die stelsel in toestand n is $\mu(n)$ vir $n = m+1, m+2, \ldots, M$

Die gemiddelde deurvoertempo van die stelsel oor alle stelseltoestande is

$$X = \sum_{n=m+1}^{M} \mu(n) \cdot P(n) = \sum_{n=m+1}^{M} C(n) \cdot \frac{1}{T}$$

Onder basiese operasionele ewewig geld dat vir alle waardes van n is

$$A(n-1) = C(n)$$

$$\text{Dus } X = \sum_{n=m+1}^{M} A(n-1) \cdot \frac{1}{T}$$

Hierdie waarde is gelyk aan die gemiddelde aankomstempo by die stelsel, dus is in hierdie geval die deurvoertempo gelyk aan die aankomstempo en gee dit gevolglik geen aanduiding van stelselprestasie nie.

Indien $U = 1$ is, is X 'n aanduiding van die maksimum aankomstempo van take wat die stelsel kan hanteer.

Gemiddelde toulengte

Die gemiddelde toulengte $\bar{n}$ van die stelsel is die gemiddelde aantal take in die tou, plus die taak in bediening.

$$\bar{n} = \sum_{n=m}^{M} n \cdot P(n)$$

Responstyd

Responstyd kan uitgedruk word in terme van die gemiddelde toulengte en stelseldeurvoer [1].

$$R = \frac{\bar{n}}{X}$$

Slotsom

Die stogastiese metode is tot dusver met groot sukses in die prestasie-ontleding van reële stelsels gebruik. Alhoewel die aannames nie geverifieer kan word nie, moet tog aanvaar word dat die meeste stelsels wel inherent daaraan voldoen. In teenstelling met die stogastiese benadering, kan die aannames van die operasionele benadering wel bewys word. Verder word daar in die afleiding van die formules ook nie van enige gevorderde wiskundige of statistiese kennis gebruik gemaak nie en kan die gemiddelde prestasie-analis dus met veel meer vertroue van die operasionele benadering gebruik maak.

Verwysings

[1] J.P. Buzen, Fundamental Operational Laws of Computer System Performance, Acta Inf. **7** (1976) 167-182.

[2] J.P. Buzen, Operational Analysis: An Alternative to Stochastic Modeling, Proc. Int. Conf. Performance Computer Installations (North-Holland Publ. Co., Amsterdam, 1978) 175-194.

[3] P.J. Denning en J.P. Buzen, Operational Analysis of Queueing Networks, Proc. of Third International Symposium on Modelling and Performance Evaluation of Computer Systems (North-Holland Publ. Co., Amsterdam, 1977) 151-172.

[4] P.J. Denning en J.P. Buzen, The Operational Analysis of Queueing Network Models, Comp. Surveys **10** (1978) 225-261.

[5] J.D. Roode, Multiclass Operational Analysis of Queueing Networks, Proc. Fourth International Symposium on Modelling and Performance Evaluation of Computer Systems (North-Holland Publ. Co., Amsterdam, 1979).

[6] L. Svobodova, Computer Performance Measurement and Evaluation Methods: Analysis and Applications, (North-Holland, Amsterdam, 1976).

[7] J.C. van Niekerk, 'n Operasionele Analise van Rekenaarprestasie, M.Sc-verhandeling (RAU, 1980)

Detecting Errors in Computer Programs

Bill Hetzel
Blue Cross and Blue Shield of Florida

Peter Calingaert
University of North Carolina at Chapel Hill

Abstract
A controlled experiment was designed and conducted to compare three methods for detecting errors in computer programs: disciplined, structured reading; specification or black-box testing; and a refined form of typical selective testing. Reading was found to be significantly inferior in effectiveness to the other two methods. Specification and selective testing did not differ significantly from each other. On the average, subjects found little more than half the errors present, even on a severity-weighted basis.

Good performance in detecting errors was found to be closely associated with the experimental subjects' computing education, computing background, and self-confidence in performing the experimental tasks. Little association was found with the amount of time spent in detection. The distribution of time to detect the next error was observed to be approximately uniform.

ACM Reviews Categories 4.6; 2.49

Introduction

A major part of the development of any computer program or system is aimed at assuring that the program actually works correctly. The definition of how a program is supposed to work is provided by its *specification*. If the program does not behave as specified, it is said to contain an *error*. Current techniques for detecting errors are far from perfect. A strong incentive is present to develop better methods and improve the effectiveness of our ability to detect errors when they exist. This paper reports on a study [1] and experiment that were conducted to aid in that process.

The basic aim was to learn more about the error-detection activity. Except for some recent work [2] on the classification and tabulation of error data, surprisingly little empirical research has been done in this area. Debugging has been studied by a number of researchers, but the focus has been on the diagnosis and correction of errors once they are found [3,4]. This study was concerned only with the initial detection of errors and not their removal.

It was desired to compare different error-detection techniques, analyze individual differences in using the techniques, develop hypotheses and in general gain as much insight as possible into the error-detection activity. Data were collected experimentally. The basic design of the experiment centered around error-detection sessions in which subjects were given a program and asked to detect the errors in it by using a specified error-detection method. The next section of the paper describes the experiment in detail. A third section presents the results of the experiment and the final section reviews the major conclusions and significance of the study.

EXPERIMENT

Methods

Three methods were selected for comparison — one based on an examination or reading of the program, a second on specification or black-box testing, and a third on a mixture of both reading and testing. The methods were intended to be broadly representative of the spectrum of possible approaches.

In specification testing, the program source code was not available and subjects were given only the specifications. The subjects selected and executed test cases based on the specifications, and then examined the resulting output for discrepancies. This method is sometimes called black-box testing, for the subject has no way to determine the internal construction of the program being tested.

In the mixed method, subjects constructed test cases based upon examination of both specifications and source code. A software tool was provided that indicated execution counts and permitted the subject to ensure systematically that each statement was executed at least once. Every subject but one did in fact execute each statement. Except for this additional criterion, the mixed method is the testing method typically used by most programmers.

The reading method selected was a disciplined and structured desk check. It relied on the fact that the programs were structured to consist of a simple sequence of paragraphs. Each paragraph had the property that flow of control entered at the top and left from the bottom, with the provision that any paragraph could invoke another. The reading procedure consisted of a bottom-up reading and characterization of each paragraph, followed by a top-down reading of the complete program. As each paragraph was read, its external effects were characterized on a special Effects Summary form. Basically, the subject tried to characterize and record the effects of each paragraph as though it were a high-level statement. After all paragraphs were characterized, the program was read top-down with the aid of the Effects Summary form. The resulting program effects were compared with the specifications and any differences recorded.

Programs

The programs selected were actual applications developed at the University of North Carolina (Chapel Hill) Computation Center. One (ANSI) was a program that reads in an arbitrary Fortran program and converts several statements to conform to the ANSI Fortran standard. A second (LIBRARY) was a program to maintain a file of bibliographic references and print a table of contents and a keyword cross-reference index. The third (TEST) was a program to score and evaluate multiple-choice examinations and provide a cumulative examination-grading report.

Each program was written in a typical production-programming environment as a carefully structured PL/I program. The reason for restricting attention to highly structured programs is the expectation that such code is rapidly becoming the norm. The actual errors that were found during the development and production use of the programs were recorded and retained. Each was then reinserted into its program, provided that proper execution of at least one simple test case was possible. The programs were therefore free of syntactic and semantic errors; only logical errors were present.

The descriptions for each program were carefully revised and their clarity was tested in a pilot experiment. The resulting specifications were considerably clearer and more precise than is usually the case. Although all three programs were relatively simple, the modules do represent a spectrum of both applica-

tion and coding complexity. The smallest program (ANSI) contained 75 statements and 5 paragraphs, and the largest (TEST) had 240 statements and 11 paragraphs.

In this manner three program modules were prepared that reflected a production environment and contained naturally occurring errors. Each module executed at least one simple test case correctly and was a realistic approximation to a module that a programmer might have at the start of testing.

Subjects

The aim of this experiment was to find out not so much what programmers actually do, but rather what they *can* do. The attempt was to design an experimental setting to yield results as good as or better than could be expected in actual practice. This meant obtaining subjects as highly qualified and experienced as possible and ensuring that they were strongly motivated. This was achieved by actively recruiting and selecting subjects to participate in the experiment and offering monetary incentive for high performance. Each subject was paid a minimum of $75. An additional payment of as much as $200 was based on relative performance during the experiment. Thirty-nine subjects were selected, most of whom were highly educated and experienced. Six were female and thirty-three were male. Just under half held a master's or Ph.D. degree. Their average work experience in computing was over three years. All had either work experience or academic course experience with PL/I. Their backgrounds are summarized in Table 1.

TABLE 1
Subject Backgrounds

	Minimum	*Mean*	*Maximum*
Age	20	27	38
Grades (A = 3, B = 2, C = 1)	1	2,3	3
Degree (Ph.D. = 4, M.S. = 3, H.S. = 1)	1	2,4	4
Computing Work Experience (months)	6	38	124
Programming Work Experience (months)	3	36	84
PL/I Work Experience (months)	0	18	61
Computer Science Courses	0	8	17
Programming Courses	0	3	9

Administration

The experiment consisted of each subject trying to detect errors in each of the three programs by using a different method for each program. Each subject was randomly assigned to one of three groups A, B, and C. Table 2 shows for each group the correspondence of testing method to program. For example, each subject in group A used the reading method on ANSI, the specification method on LIBRARY and the mixed method on TEST. The order of the three sessions for each subject was also randomized.

TABLE 2
Assignment of experimental conditions

Method	Program		
	ANSI	LIBRARY	TEST
Reading	A	C	B
Specification	B	A	C
Mixed	C	B	A

The basic design of the experiment permitted for the three error-detection methods a comparison that was controlled for differences in subject, differences in program, and differences in order of experimental tasks. The primary performance measure was the percentage of errors found. An alternative measure was a percentage score weighted according to error severity.

All subjects participated in a five-hour instruction session prior to the start of the experiment. The objectives of the experiment and the error-detection methods were explained. The subjects then participated in three error-detection sessions each lasting between three and five hours. Each subject was given a time limit that depended only on the program being verified. The instruction and error-detection sessions were held during the course of one week in a large classroom reserved for the purpose. An experiment staff of four persons coordinated and monitored each session. Generally, two staff members remained in the classroom to supervise and answer questions while the other two submitted and returned test runs. These test runs were prepared on coding sheets given to the experiment staff to be keypunched and run. After execution the output was returned to the subject. Special computer center procedures established for the experiment made it possible to achieve an average turnaround time under fifteen minutes. Each subject was thus provided with excellent response time and freed to concentrate on the error-detection task.

Subjects did not leave the room without being signed out. They were permitted to sign out of the experiment for a break period to avoid losing experiment time waiting for a run to be returned or to assist the staff in keypunching test data. Any time spent on breaks was not included in the subject's time limit, nor counted in his elapsed time. Subjects were instructed to take a break whenever they became fatigued or had to wait for output.

Data

At the start of the experiment, each subject was given a background survey and an attitude survey. The background survey provided data about each subject's education, experience, self-estimates of ability, and other background variables. The attitude survey requested the subject's opinion towards the various methods and the experiment. It was given again after the final error-detection session to permit an analysis of attitude shifts.

The basic data from each session were recorded on error-detection logs. Each subject logged the submission and return of test cases, breaks taken, and suspected errors. At the end of each session, the subject also completed a survey form containing general information about the session. After the experiment, each log was carefully reviewed and encoded. Descriptions of possible errors entered in the logs were matched against the list of actual errors, and the appropriate error number was coded. The coded data were read into a program that reproduced the logs and provided data for further analysis.

RESULTS

Comparison of Methods

The three error-detection methods were compared with respect to the percentage of the total errors detected with each method. The results are summarized in Table 3. Averaged across the three programs, the mixed and specification testing methods detected just slightly fewer than half of the errors present in the programs. For the reading method, only 37 % of the errors were found. Similar relative results are seen for each of the programs individually. The percentages for mixed and specification testing were very close and the percentages for reading were considerably poorer.

TABLE 3
Mean percentage of errors detected by all programmers in each experimental condition

	ANSI	*LIBRARY*	*TEST*	*Mean of 3 Methods*
Reading	48	33	31	37
Specification	55	52	36	48
Mixed	57	48	35	47

An analysis of variance showed the variances accounted for by the programs and by the methods to be highly significant.

The variances due to each group, each replicate, and the different task orders were very small and not significant. The mixed and specification testing methods were not significantly different, but each was very significantly (at the ,001 level) better than reading.

One question was whether these results might be sensitive to the severity of the errors detected. Some errors were very minor and for a few it was even questionable as to whether they were really errors. A number of weighting schemes were used to assign to each error a score based on its severity. The data were then analyzed using the percentage of the total score as criterion. The conclusions were unchanged. The authors could not even think of any plausible weighting scheme that led to a different conclusion. The method differences are present quite uniformly across the different types of errors and are large enough that the choice of weighting scheme has no effect.

The clear conclusion is that the specification and mixed methods are essentially equivalent and that reading is significantly inferior.

Individual Differences

One object of the study was to try to explain the observed individual differences in performance. The ranges of observed differences are shown in Table 4. In general, the best performer was two to three times as capable as the worst in mixed and specification testing, with the spread somewhat greater for reading. To investigate these differences, an analysis was made of the association between each subject's performance and his background, attitudes and approach. Rank order correlation coefficients and χ^2 contingency tables were used as measures of the association between the various variables and a subject's score.

TABLE 4
Extreme percentages of errors detected by all programmers in each experimental condition

		Lowest	*Highest*
Reading	ANSI	20	80
	LIBRARY	7	60
	TEST	0	67
Specification	ANSI	40	67
	LIBRARY	20	73
	TEST	24	72
Mixed	ANSI	27	73
	LIBRARY	27	67
	TEST	24	48

The analyses showed a number of variables to be significantly related to good performance in detecting errors. Regardless of the program being verified or the method used, close association was present between performance and the subject's computing education, computing experience, and self-confidence in performing the experimental tasks.

Little association was found with basic variables such as age, sex, degree level, other self-estimates and attitudes, and the amount of time used by the subject.

One interesting variable that showed moderate association was the number of test cases run by the subject. Particularly in specification testing, there was evidence that some subjects adopted a "try anything" attitude and just created large numbers of test cases in hope that some error might show up. Discounting subjects who submitted over twenty test cases in a single session, the association between good testing performance and the number of test runs was highly significant.

A factor-analysis model of the data was also developed in an effort to explain the underlying relationships in the data. In each of some tens of runs with different variables, about 70 % of the variance in the data was accounted for by variables that could be grouped into four derived factors. The multiple runs tested the model's sensitivity and showed it to be quite stable. The four factors were interpreted as experience, self-esteem, computing education, and attitude toward the experiment. A regression prediction model was also produced. The best-fitting model contained the variables of academic major, education (measured by number of courses taken), self-esteem, attitude, and work experience; it gave an average prediction error of 18 %. The only highly significant variable was academic major. For the group of subjects in the experiment, this variable was closely correlated with computing education and computing experience. Over-all, the regression and factor models support and strengthen the conclusions obtained from looking at the association measures. Subjects with substantial computing education and experience backgrounds who felt confident about their abilities were the ones most likely to do the best jobs of detecting errors.

Other Analyses

A number of other analyses were made in an attempt to develop hypotheses about the error-detection activity and gain additional insight. Two of the more interesting are reported here.

The first was an analysis of the distribution of the time to detect the next error. A program was written to count the detections that occurred in successive time intervals during error detection. Several theoretical distributions were then fit to the counts. The results were surprising. The best fit was simply a uniform distribution, regardless of the method used or the program being verified. Subjects steadily increased their knowledge about the program and tended to try more complex test cases as the session progressed. A plausible explanation for the uniform distribution may be that the increasing knowledge and test case sophistication just about offset the reduced error population. This is a very different situation from the usual program development case. Both Tucker [5] and Schneidewind [6] have reported an exponential increase in error-detection times in that situation.

A second analysis examined the individual errors to see whether different types tended to be found to different extents by the different methods. If at least one-third more of the subjects detected an error with one method than detected it with another, then the error was considered to be found to a significantly different extent by the two methods. Such errors were then categorized in an effort to establish classes of errors that tended to be more difficult to detect by one method than by another. The results confirmed what might be suspected intuitively. Reading did not work well for errors of omission of code statements. Errors involving interrelationships between code segments in different paragraphs were also difficult. Specification testing was more effective for detecting errors that showed up on test cases suggested by the specifications and less effective for hard-to-generate test cass. Mixed testing, which includes some aspects of both reading and specification testing, tended to fall in the middle. Only one error was detected significantly more times as a result of the requirement to execute each of the statements at least once. In general, the path testing requirement seemed to be of very little value.

CONCLUSIONS

How Significant are the Results?

Comparing the three methods gave a very consistent message. Regardless of the performance measure used, specification testing and mixed testing were essentially equal and reading was a poor third. The programs differed significantly, but the relative performance of the methods was the same for each program. In general, this result was unmistakable and convincing.

What about the Generally Poor Performance?

No subject found all the errors in any of the error-detection sessions and, on the average, only about half the errors were detected. Why was this performance so poor? Every effort was made to obtain maximum performance from the subjects. The programs were clearly structured, computer turnaround was excellent, subjects were highly educated and motivated, and they worked without distractions. It is reasonable to conclude that the results are likely to be better than what can be expected in

actual practice and that the detection of errors is a very difficult process. The experiment shows that an intensive period of independent error detection does not provide any assurance of correctness. One might speculate whether financial incentives to program writers and program testers would lead to the generation of fewer errors and the detection of more.

Why wasn't the Mixed Method Better?

The mixed method was designed to have the advantages of both reading and specification testing without the disadvantages of either. That it did not turn out better shows that the time and mental effort required to comprehend the source can be a detriment. Careful concentration on the specifications seems to lead to better test cases and to increase the likelihood of error detection.

How Well can Individual Differences be Explained?

Three groups of variables were found to be significantly associated with error-detection performance — computing education, computing experience, and self-confidence in error detection. Use of test runs was also associated to a point. These associations were found to be present fairly uniformly for all of the programs and methods. The results suggest that many of the skills needed for good error-detection performance can be taught and acquired.

What Else was Learned?

The distribution of the time to find the next error was shown to be approximately uniform. This was attributed to the increasing subject knowledge and submission of more complex test cases offsetting the reduced error population. The experiment also confirmed the logical deductions that reading was not effective for detecting errors of omission and that specification testing was not effective for detecting errors that were hard to generate. Finally, the experiment showed that very little help in detecting errors was provided by the path testing requirements to execute each statement.

Other Contributions

Other benefits of this investigation include the data resource acquired, the experimental methodology, and some suggestions for future research. Worthy of further study are the effect of different numbers of errors in the program being verified, the relationship of program complexity to error detection, and a closer examination of the usefulness of path testing, especially for larger programs. The raw data have been carefully preserved in the first author's dissertation [1].

Acknowledgements

The authors gratefully acknowledge the influence of studies by Gold [7] and Sackman [8] upon the development of the experimental methodology. This work was partially supported by the national Science Foundation under Grant GJ-30410.

References

[1] W.C. Hetzel, *An Experimental Analysis of Program Verification Methods.* Doctoral Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, 1976. (Available from Xerox University Microfilms, 300 North Zeeb Road, Ann Arbor, MI 48106).

[2] B.W. Boehm, R.K. McLean and D.B. Urfrig. Some Experience with Automated Aids to the Design of Large Scale Reliable Software. In: *Proceedings 1975 International Conference on Reliable Software,* IEEE Catalog no. 75CH0940-7CSR, New York, 1975, pp. 105-113.

[3] J. Gould and P. Drongowski. *A Controlled Psychological Study of Program Debugging.* IBM Research Report RC 4083, Yorktown Heights, NY, 1972.

[4] W.C. Hetzel, A Definitional Framework. In: *Program Test Methods,* Englewood Cliffs, NJ, Prentice-Hall, 1973, pp. 7-11

[5] A.E. Tucker, *The Correlation of Computer Programming Quality with Testing Effort.* System Development Corporation Report TM 2219, Santa Monica, CA, Jan. 1965.

[6] N.F. Schneidewind, Analysis of Error Processes in Computer Software. In: *Proceedings 1975 International Conference on Reliable Software,* IEEE Catalog no. 75CH0940-7CSR, New York, 1975, pp. 337-347.

[7] M.M. Gold, *Methodology for Evaluating Time Shared Computer Usage.* Doctoral Dissertation, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1967.

[8] H. Sackman, *Man-Computer Problem Solving: Experimental Evaluation of Time Sharing and Batch Processing,* Princeton, NJ, Auerbach, 1970.

Restructuring of the Conceptual Schema to produce DBMS Schemata

S. Wulf
Comcon (Pty) Limited, Sandton

Abstract
An overall methodology for database design is presented. This includes creation of a data dictionary and relational analysis. An algorithm is presented to create a conceptual schema or logical database design from the set of third normal form relations. The primary emphasis of this paper is to present algorithms for restructuring the conceptual schema to create first pass designs (FPD) for particular DBMS. These FPDs can then be used as the basis for iterative physical database design and performance predictions. Examples are given for creating FPDs for TOTAL and IMS. These can be extended for any other DBMS. Indications are given for future research in extending the methodology.

1. INTRODUCTION

In practice, database design has been found to be complex and difficult. The resulting success of the database is a function of the knowledge of the initial information requirements, the robustness of the design and the subsequent performance of the database. It has been pointed out [1,2,3] that the overall design process should be structured and formalised. The following iterative phases have been proposed [4] to cover the database design activity from gathering of information requirements to physical implementation of the database:

1 Information requirements specification
2 Creation of the data dictionary
3 Relational Analysis
4 Creation of the conceptual schema
5 Structured analysis and local views
6 First pass DBMS design
7 Performance modelling
8 Physical design

In this paper we review briefly the process of relational analysis culminating in the creation of a refined conceptual schema. The conceptual schema is a network diagram with nodes corresponding in general to entities and the links between them to the relationships between these entities. An entity is a person, location or object of interest to the organisation being modelled. It is clearly a logical model — that is, independent of any particular hardware and software implementation constraints.

This paper presents algorithms for specifying operations upon the conceptual schema to modify it to conform to the data structuring constraints of certain target database management systems (DBMS). The objective is to create a DBMS schema [5] as close in flexibility to the conceptual schema as possible. We call this the first pass design (FPD). This design serves as the starting point for iterative physical design.

This entails modelling the proposed database system using queueing network techniques based on algorithms developed in [6] and [7] and modifying the current DBMS schema is arrived at which satisfies minimal performance constraints of the application. Physical design is outside the scope of this paper. In Section 2, a brief review of relational analysis is presented. In Section 3, the production of the conceptual schema is discussed. This serves as the source schema for FPD. Data restructuring algorithms for FPD are presented in Section 4 for two major DBMS — IMS [8] and TOTAL [9]. A sample conceptual schema is then used in Section 5 to prepare a FPD for both DBMS. Future work is discussed in Section 6 and conclusions are presented in Section 7.

2. RELATIONAL ANALYSIS

The process of information requirements specification culminates in the production of a data dictionary. This is essentially a list of data elements sorted alphabetically by name of the data element together with information about the functional dependencies between pairs of data elements [10]. Relational analysis consists of applying successively the procedure of normalisation [11] on this data to produce a set of third normal form relations. By definition each named relation consists of a set of data elements, one or more of which constitute the primary key. We define a foreign key in a relation to be one or more data elements of that relation which are a primary key of another relation. Following [3] we use the following notation to document a relation:

relation-name
** data-element$_1$
** data-element$_2$
,
,
,
** data-element$_n$
data-element$_{n+1}$
,
,
,
data-element$_m$
* data-element$_{m+1}$
,
,
,
* data-element$_p$

where double-starred data elements comprise the primary key of the relation and single-starred data elements comprise foreign keys. This set of relations is used to produce the conceptual schema.

3. THE CONCEPTUAL SCHEMA

The initial conceptual schema network diagram is created by applying the following rules systematically:

1. Each relation becomes a node with the relation name as node name primary key of the relation as key of the node and data elements of the relation of data elements of the node.
2. If a subset of the primary key of any relation is not the key of any existing node then create a new index node with that subset of the primary key as key of the node.
3. If the key of any node is a subset of the key of another node then draw a link between the two nodes with a single headed arrow (→), pointing into the node with the superset key.
4. If the key of any node is a foreign key of another node then draw a link between the two nodes with a double-headed arrow (↠) pointing into the node which has the key data elements as a foreign key.
5. If more than one single-headed arrow is incident on a node, we designate one link as the primary link and make all the others double-headed arrows.

The result will be a network diagram consisting of a set of

nodes with a single and double headed arrows joining them.

The procedure of structured analysis [4] culminates in the production of a data flow diagram based on the functional requirements of the proposed application system. From this overall system data flow diagram, a set of linear data flow diagrams such as those described in [12] can be constructed. There is one linear data flow diagram per application transaction (input-transformation-output). Associated with each linear data flow diagram, that portion of the overall conceptual schema which it needs can be derived.

The overall conceptual schema is based on the complete data dictionary and the functional dependencies between data elements without regard to the functional requirements of the application system. It is likely, therefore, that many of the nodes and relationships of the conceptual schema will not be required on the implemented database. Subsets of the conceptual schema used in each individual linear data flow diagram constitute local views or subschemata of the conceptual schema. By consolidating all these views, we can produce a refined conceptual schema. This is the logical model of the database which will be the basis for physical database design. In practice we have found that the initial conceptual schema for an organisation is large (70-100 nodes with 120-150 links) while the refined conceptual schema is smaller and hence more manageable (10-40 nodes with 15-50 links).

4. RESTRUCTURING OPERATIONS FOR FPD

4.1 Hierarchical and non-hierarchical relations

In [13] analysis of the logical and physical structures of databases is discussed. Hierarchical and non-hierarchical data relationships are defined and application of the schema analysis methodology to restructuring is presented. This work has been used as the basis for the techniques presented in this section for deriving the FPD for various DBMS given a particular refined conceptual schema.

Consider a set relation (A,B) where A and B are nodes of the conceptual schema and there exists a link between A and B with the arrow head pointing into B. If the key of B is a subset of the key of A, we call (A,B) a hierarchical relation. A is called hierarchically superior and B is the hierarchically dependent node. Clearly, every relation in the conceptual schema which has a single-headed arrow is a hierarchical relation. If the set relation (A,B) is not hierarchical, we call the relation non-hierarchical. Every relation in the conceptual schema which has a double-headed arrow is non-hierarchical.

Generally, hierarchical relations denote one-to-many relationships between owner nodes A and member nodes B of a set relationship. Non-hierarchical relatins are generally many-to-many. If a node participates as the member of more than one non-hierarchical relation (ie. two or more double-headed arrows point to the node) it in fact contains the "intersection data" or "link data" for logical occurrences of the implicit many-to-many relationship between the two owner nodes. In [13] and in the various representations of the entity-relationship model [14], this data is not represented explicitly as a node but as data associated with the relation. We have found the representation used here to be closer to DBMS schemata and hence easier to use.

4.2 Restructuring algorithms for FPD

The objective of FPD is to manipulate the source conceptual schema network schematic in a programmed way to ensure that the target DBMS schema is as conformable as possible. The conceptual schema structure is a generalised network with possibly mixed hierarchical and non-hierarchical relations while DBMS schema specifications usually place restrictions on the allowed schema structures. For example, TOTAL does not allow a given node to be both an owner and a member. IMS supports only strictly hierarchical "physical" databases but allows these to be linked together to form "logical" databases. We present here algorithms for both TOTAl and IMS to restructure the source conceptual schema to valid FPDs.

Algorithm for TOTAL

1. For all nodes i,
   If source-node (i) = any owner and source-node (i) not = any member
   create target-node (i, 'master') = source-node (i)
2. For all source nodes i, j with i ≠ j
   If source-node (i) = any member of source-node (j) and target-mode (j, 'master') or target-node (j), 'master', 'index') exists
   create target-node (i, 'variable') = source-node (i)
   link target-node (i) to target-node (j, 'master', 'index')
3. For all source nodes i,
   If source-node (i) = any member and target-node (i, 'variable') exists and source-node (i) = any owner
   create target-note (i, 'master', 'index')
4. Repeat steps 2-3 for all nodes i.

In the above algorithm, 'master' and 'variable' denote the creation of TOTAL master and variable data sets., 'index' denotes creation of a master data set with one key record for each record as the associated variable data set.

Algorithm for IMS

1. For all nodes i,
   If source-node (i) is not = hierarchical member
   create target-node = source-node (i)
2. For all source-nodes i, j and i ≠ j,
   If source-node (i) = hierarchical member of source-node (j) and target-node (j) exists
   create target-node (i) = source-node (i)
   link target-node (1) to target-node (j)
3. For all source-nodes i, j and i ≠ j
   If source-node (i) = non hierarchical member of source-node (j)
   link 'logical' target-note (i) to target-node (j)
4. Repeat 2 and 3 for all nodes i.

In the above algorithm 'logical' denotes the creation of a logical child link between the two nodes.

5. DATA BASE EXAMPLE

Suppose that the following set of third normal form relations is given:

Agent
**agent-no
agent-details

Customer
**customer-no
customer-details

Customer-Account
**customer-no
month-no
account-details

Customer-Address
**customer-no
**address-type
customer-address

Order
**order-no
order-details
*customer-no
*agent-no

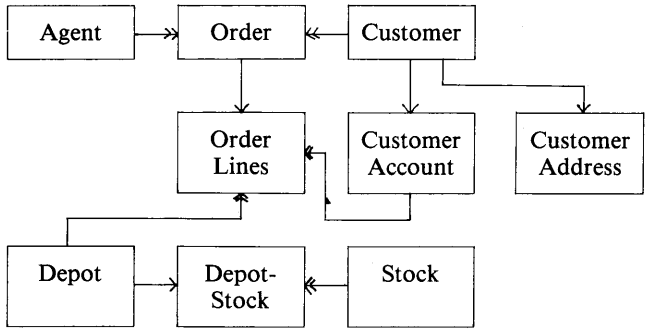
Order-Lines
**order-no
**stock-no
*depot-no
line-details

Stock
**stock-no
stock-details

Depot
**depot-no
depot-details

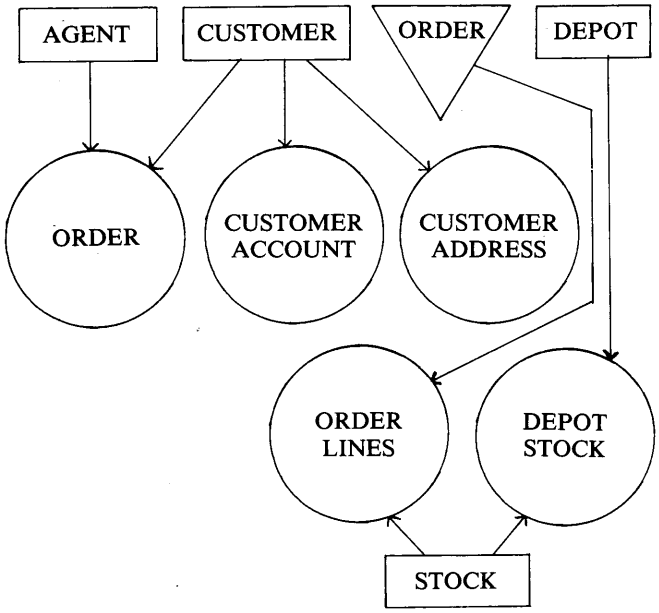
Depot-Stock
**depot-no
**stock-no
quantity

This is a typical set of relations in a commercial accounting environment. Applying the algorithm presented in Section 3 we obtain the following conceptual schema:

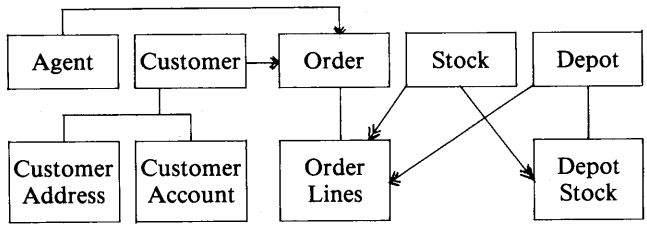


The next step is to complete the data flow diagram for the proposed application. From the data flow diagram, linear data flow diagrams together with their associated local view diagrams can be prepared. The union of all the data flow diagrams corresponds to the refined conceptual schema. We assume that this refined conceptual schema turns out to be equivalent to the above conceptual schema.

If the TOTAL algorithm is applied, the following FPD is obtained (master data sets are designated by □, index master data sets by ∇ and variable data sets by ○):



If the IMS algorithm is applied, the following FDP is obtained:



The double-headed arrow indicates a logical parent-child relationship.

6. Future Developments

In continuing work by the author, application system functional requirements as summarised in the data flow diagram together with the FPD are used to produce a structured design [12] hierarchy diagram. Similar to the algorithms for FDP, algorithms to control subsequent iterative modification of the DBMS schema can be defined. These incorporate the restructuring operations of [15]. Depending on the results of a prediction model at each iteration, appropriate restructuring of the current DBMS schema can be performed to meet either disk space constraints or the response/throughput time constraint. This iteration continues until the DBMS schema most closely resembling the conceptual schema and meeting performance specifications is produced.

7. Conclusions

The algorithms for FPD have been specified here for TOTAL and IMS only. It is relatively simple to produce simple algorithms for other DBMS. For example, Codasyl-type systems such as IDMS need only specifiy procedures for restructuring bijunctive relations to produce an FPD.

They offer a disciplined method for initial physical database design to the database administrator and ensure that the benefits of formal relational analysis are not lost in tailoring the conceptual schema to a DBMS compatible one.

References

[1] E. Aurdal, A multi-level procedure for file design, cascade working paper no. 39, Univ. of Trondheim, 1975.

[2] B. H. Kahn, A method for describing information required by the database design process, Proc ACM Sigmod, 1976.

[3] S. Wulf, A database project methodology, Internal report, Comcon (Pty) Limited, 1977.

[4] S. Wulf, A database project methodology, Internal working paper, Comcon (Pty) Limited, 1980.

[5] Report of the Codasyl DBTG, 1971.

[6] Baskett et al., Open, closed and mixed networks of queues with different classes of customers, JACM 22, no. 2, 1975.

[7] S. B. Yao, An attribute based model for database access cost analysis, ACM TODS, 2, no. 1, 1977.

[8] IMS/VS General Information Manual, Form GH20-1260, IBM, 1982.

[9] TOTAL Application Programmer's Guide, Pub PO2-1236, Cincom Systems, 1981.

[10] E. F. Codd, Further normalisation of the database relation model, Courant Computer Science Symposia, 6, 1971.

[11] C. J. Date, Normalisation — An introduction to database systems, Addison-Wesley, 1975, pp 95-114.

[12] Stevens et al., Structured design, IBM Systems J, 2, no. 2, 1974.

[13] S. B. Navathe, Schema analysis for database restructuring, ACM TODS, 5, no. 2, 1980.

[14] P. P. S. Chen, The entity-relationships model — Towards a unified view of data, ACM TODS, 1, no. 1, 1975.

[15] S. B. Navathe and J. P. Fry, Restructuring for large databases: Three levels of abstraction, ACM TODS, 1, No. 2, 1976.

Managing and Documenting 10-20 Man Year Projects

P. Visser
Software Management Systems Pretoria

Abstract
This paper presents a summary of tutorial material on accepted management techniques and philosophies as applied to the development of large software systems, based upon the documentation for such systems.

INTRODUCTION

Techniques and facilities to increase production can only be effective if they form part of a total management effort. This applies especially in the development of large software systems. Management of software development is not different from normal management, yet it is known to be difficult. The reasons can be found in the characteristics of software development which resembles research work in most management aspects. The method of implementation of management techniques must therefore be adapted to these characteristics:
— Not visible in the normal sense.
— Production facilities not obvious.
— Type of personnel — highly skilled.
— "Peoples" systems.
— End product defined as part of production — direct control is not possible as for manufacturing.
— Measurement dependent on judgement.

PRODUCTION MANAGEMENT FOR SOFTWARE PROJECTS

The purpose of production management is to accomplish the efficient use of company resources which constitute mainly personnel in the case of software development. Planning and control of the following aspects are therefore essential.
— Personnel — appointments, development, training and scheduling.
— Realistic progress estimates.
— Quality of completed items.
— Running cost per project.

Planning should establish realistic objectives to be used as criteria in the control process. The methods for establishing these objectives must however be adapted to the characteristics of software development. The planning and control systems must be based on self-balancing principles. Systems based on direct control principles can only destroy self-motivation and thereby the basis for a democratic process which is essential in the development of systems by a team. (Democratic process is used here in the sense that sound creative ideas from any team member can be incorporated into the design in a controlled way).

MANAGEMENT INPUTS

Facilities
Word processing facilities are so essential to the design process as hardware, compilers and other utilities to the development of executable programs. Other essential facilities (services) are typing/editing and a program library.

Software Standards
Standards should prescribe work procedures suitable for the proper planning and control of progress, cost and quality. Standards should also set criteria for the evaluation of the quality of development work.

Development of Personnel
Personnel should be encouraged to use the various techniques and methods for software development skilfully. This can only be achieved through proper guidance in a competitive environment. Training in techniques and specific items (e.g. software packages) is also essential in the specialization of personnel in order to provide them with a proper background for executing their allocated tasks effectively.

Organization
The organization of the team is essential for the division of work. It must provide for the democratic process needed during design but without endangering the integrity of the design. An efficient project management system is also essential but must be an integral part of the organization and work procedures.

PLANNING AND CONTROL

Principles
Three activities must be pursued on an ongoing basis:
— Setting of criteria (planning)
— Measurement of performance
— Evaluation of performance and corrective actions
These activities must be directed towards:
— Quality of work
— Progress of work
— Cost of work

Criteria for evaluation (Objectives)
In order to use a criterion for objective evaluation it must be measurable and have short term attainability (2-4 weeks). To set criteria for progress therefore requires that a large project be broken into small, executable tasks, each with a defined physical output. Coding has a natural output in the form of program listings, but constitute the smallest portion of work. The output for all other phases should be prescribed in the form of documents to be generated. The completion of such a document then marks the completion of a task and should be achieved in a short period of time.

If the completion of each small task is measurable, the progress for the total project can be calculated. It is assumed in this type of calculation that all tasks needed for project completion are included. Software standards should therefore include a standard framework for the planning of a project. This framework must be consistent with the team's work procedures which must therefore also be prescribed. The planned activities must then be scheduled against the available personnel taking sequence, skills, deliveries and other factors into account. Using the relative weight of each task, an expected progress curve for the project can be established.

High quality of work can be attained only through the combined effect of the following:

— Setting of standards for design, coding, testing and documentation.
— Correct use, development and training of personnel.
— Prescription of work procedures which will result in work of high quality.

Setting cost criteria is merely a calculation process, using the progress criteria as a basis.

Performance measurement
Progress estimates for small tasks individually contribute very little to the total progress. It is however assumed that no rework will be necessary. Quality control on the product of each small task is therefore essential as well as the total coordination of tasks. Especially during the design phase, coordinated performance to attain design integrity is important, and methods to attain this should be build into the work procedures and standards for documentation. Cost measurement can again be calculated from reliable progress estimates.

Evaluation and corrective action
If techniques are established for setting criteria as well as performance measurement, evaluation can be done regularly and corrective actions can be taken timeously. Measurement of the effect of delays is also possible both in terms of development time and development cost.

SOFTWARE STANDARDS

Implementation of Software Standards
Standards should be the documentation of sound practice developed from theory and positive experience. It should assist the organisation by providing a structure for the efficient operation of a department or team. Software standards should also describe a desired result rather than techniques open for misinterpretation. As shown in previous paragraphs the adoption of proper standards can create an environment for the development of software which is suitable for the execution of normal management practices. As such, software standards and their proper implementation are a prerequisite for planning and control of software projects. Software standards should be adhered to by competent software engineers, else it would only be a rulebook on the shelf of the manager.

Responsibilities and work procedures
A standard team structure is used in all organizations which should be adapted to the type of systems developed by the organization. Responsibilities and authority of the various positions should be set out clearly. Work procedures for the major activities of the team should be planned such that the desired end results will be attained. These activities include the different types of design, reviews and other.

Configuration Management and QA
Quality of design and coding can only be built into a system during development and must therefore be controlled within the project team. The only management input is through sound work procedures as discussed above and the training of competent personnel. Poorly designed systems normally result in delays caused by rework with an adverse effect on cost. The quality and training level of personnel must therefore always match the requirements of the system.

Product quality can be maintained by software quality assurance procedures. These procedures must be controlled from outside the development team. Configuration Management practices are geared towards safeguarding completed copies of software and documentation, and maintaining the integrity of the completed portion. The program library is the basic vehicle needed for this task. Configuration Management during development is just as essential as for a fully operational system. It must however be adapted for this work phase and its corresponding work procedures.

Configuration Management and Quality Assurance procedures must form an integrated system and must be adapted to the size of the organization and other characteristics of the system being developed. Change control procedures form a natural part of Configuration Control and should be incorporated.

Documentation Standards
Documents form the physical output of any type of design process. It is therefore necessary to define a set of documents suitable for the following processes and to cater for the specific characteristics of the type of systems developed by the organization.
— Customer and user information
— Further design and development processes
— Quality and progress control
— Coordination in the development process
— Maintenance

For each of these documents a standard layout and contents definition must be given. The documents should also be based on the same principles underlying the work procedures in order to be the natural outcome of such procedures.

Design and coding standards
As for documentation standards, design standards must also be correlated with the prescribed workprocedures. These standards should be viewed as the main criteria for making design decisions. Standards should therefore be specified for each design level independently and should be applicable to methods, control structures and data structures.

Where a standard framework for program or system structures are used in an organization, these should also be included as part of these standards. Design standards should not be open for misinterpretation. Techniques should therefore not be prescribed as a standard but rather the desired characteristics of the end result.

UNDERLYING PRINCIPLES FOR THE DEVELOPMENT OF LARGE DYNAMIC SYSTEMS

Characteristics of development
In a dynamic computer system coordinated actions are needed. The interactions and interfaces between programs result in a complex structure which influences the correctness and efficiency of the system. Mastering this global complexity in a system is the key to successful systems.

A totally integrated approach to system design and development can lead to the mastering of complexity and prevent mistakes at a lower level rather than correcting them. The design process should therefore be an ordered sequence of problem definition and decision making. The criteria for choosing between alternatives depend on the design stage. Such an ordered approach with specified decision criteria can maintain the integrity of the design throughout the design and implementation stages. Program development has a physical output in the form of compiler listings and demonstrable software. Too much attention is therefore focussed on this stage and software personnel are motivated to ignore the preceding design decisions, thereby ignoring the global complexity of a system.

The only solution to this dilemma is to enforce prescribed documentation in such a way as to provide the design phases with physical output. This facilitates progress, cost and quality control of the design phase and therefore the application of normal management techniques. More important however, it also enforces the ordered sequence of decisions by the designers, which is an essential prerequisite for the development of efficient software systems.

Global approach to design
The techniques of top-down design, development and testing focus on the program as total entity. The same approach may however be used to create an ordered approach to the design of dynamic software systems. This global design approach consists of various design levels, each of which focus on a particular entity, viewed as a black box at that stage, and its interactions with other similar entities and its environment. These levels of design determine the decision making criteria at that level, and are discussed below. In the same way the techniques of top-

down development and testing may be implimented.

This approach then allows one to focus attention on the global structure of a dynamic system during design, development and testing. The biggest problem encountered with the global design approach is that a great deal of insight with regard to the later implementation of each concept is needed. The only way to overcome this problem is knowledge, experience and preliminary investigation of implementation techniques.

External design

All systems are built to satisfy a specific need. The first task is to formalize this need in order to define a goal for system design and development. With this in mind, and taking resources and circumstances into account, a formal set of objectives for the system can be defined. The environment includes operators, recipients of output and equipment.

To describe all actions, reactions and other functions and characteristics in a consistent way, a standardized approach must be used. External design has as output documents describing the objectives and functional specification of a system. The criteria to guide design decisions during this phase must be stated explicitly in both company standards and in the objectives defined for each system.

System design

The system design level focus on the task as entity. Tasks must therefore be viewed as black boxes performing specified functions, interacting with one another and using specified interfaces to achieve this. The major activities during system design is therefore the following:

— Define independent subsystems i.e. subsystems which can perform their normal functions independently within their functional environment.
— Define the tasks in each subsystem.
— Define the interactions between tasks explicitly.
— Design interfaces between tasks as well as with the external environment.

The criteria for making design decisions during this stage, must be included in the company standards and must always be within the restrictions of the functional specification and system objectives.

A standardized form for documenting a system design must be included in the company standards. In these documents both the logic and the various data must be described. The logic of the system is described in terms of the functions and of the characteristics of tasks and their interactions. All types of data are described in the interface design. The various documents describing the system design therefore contain a complete specification for each task in the form of a description of all its functions, interactions and interfaces.

Program design

A task as defined during system design is a dynamic entity in the system, capable of communicating with other entities and reacting in a prescribed way on impulses depending on various operating conditions. Equivalent programs must therefore be designed to fulfill these specifications. The logic of each program and the corresponding datastructures are designed and documented during the program design phase.

This phase focusses on the module as entity. A program's logic are described through design methods and module definitions, as well as their dynamic and static structures. Data structures form the interfaces between modules and are as such also included in this design phase. A module is viewed as a separately compilable collection of statements and may contain several procedures. Various techniques for program design are documented in textbooks. The same applies for design criteria on which to base decisions during this phase. These criteria must however be incorporated into the company standards, as well as the approach to the documentation of this work phase.

Module design

Modules are the smallest units specified in design documentation. Module design forms part of program development. It is primarily concerned with the explicit planning of language code. Its outcome forms a natural part of module listings in the form of structured comments and the resultant code statements.

TEAM OPERATION

Team organization

Larger software projects can only be developed by teams with the accompanying problems of coordination between personnel and the maintenance of integrity of design. Specialization on the other hand can only be economical in large teams (more than 25) or in large software organizations. In the normal team for SA (5-20) a combination of ideas for team organization can be used.

The Project Manager is the coordinator with the customer and responsible for system analysis and specification. His counterpart is a senior designer responsible for the complete design and its implementation. Technically he is assisted by one or more designers and programmers.

Design coordination

The identification of subsystems and their interrelation is done by the senior designer following the system specification. During the design phase however each designer is allocated the sole responsibility for a subsystem with the most difficult or most important one to the senior designer. The design is then developed by the responsible person, assisted during meetings by all other designers. In this way other designers and especially the senior designer can present ideas, coordinate interfaces and define potential problem areas. The original ideas of the senior designer can in this way be maintained or replaced by better ideas without the danger of loosing cohesiveness between parts of the system. The designer responsible for each subsystem should then be able to impliment this design with the aid of one or more programmers which function in the same way as described above.

Coordination of development and testing

The workplan for development (coding and debugging) should follow the principles of top-down development which is a coordination tool in itself. On the other hand the standard for documents should be such that every aspect of interactions and interfaces is described fully.

Testing is validating that the development process generated the product expected by the user. Design documents should include a testplan and test procedures which describes the actions and reactions of the system. This will identify differences between user expectations and system specifications at a very early stage.

Customer/user coordination

Reviews during design and implementation of each sub-system are essential. After the design of each subsystem this design should be presented, highlighting the consequences of specifications and the restrictions of design ideas. Before acceptance testing a review should again be held to highlight implementation problems encountered and the solution implemented.

Data Structure Traces

S. R. Schach
University of Cape Town

Abstract
Three levels of traces for data structures (as opposed to simple variables) are defined. A machine-code core dump is essentially a low level trace. A high level trace reflects the high level language in which the data structure being traced has been implemented. A very high level trace displays the data structure in the format in which the programmer conceptualizes it. Three traces written by the author (a graphical FORTRAN array trace, a portable trace for the Pascal heap, and a graphical Pascal data structure trace) are described, and the level of each trace is then analyzed.

INTRODUCTION

Just as languages may be described as high level or low level, so we may classify the level of a trace. For example, suppose we wish to trace the Pascal statement

total := total + 50

If the value of total was 1 000 before the statement was executed and the statement itself is at line 123, a high level trace might print something like

line 123: total = 1000/1050

Such a trace may be described as high level because it reflects the high level language in which the statement being traced was written.

But if the trace prints the machine-code equivalent of the original Pascal statement, or the trace takes the form of a core dump, then such a trace is low level.

When tracing data structures, rather than simple variables, a third category of trace may be defined. If the output of the trace displays a data structure in the format in which the programmer conceptualizes it then such a trace will be termed very high level. For example, if tracing an array causes the elements of that array to be displayed arranged in rows and columns, or if a tree or doubly-linked list appears as such, then since the "shape" of the trace output corresponds to the "shape" of that data structure within the programmer's mind this is a very high level trace.

In this paper the above three categories of data structure trace are analyzed and evaluated. Then three data structure traces written by the author are described, and the category into which each falls considered. The traces are:
(1) A FORTRAN array trace [6].
(2) A trace for the Pascal heap [7].
(3) An interactive graphical trace of the Pascal heap [2].

COMPARISON OF DATA STRUCTURE TRACES

Low level data structure traces

Debugging a program written in a high level language by analyzing a machine-code core dump of a data structure is most undesirable. In the first place, the programmer is required to have detailed knowledge of the internal representation of his program and data, defeating the whole purpose of high level language programming. Furthermore, the ability to understand core dumps is becoming increasingly rare as fewer and fewer programmers are being trained in low level languages. But even if a programmer does possess this skill, it is strictly non-portable from machine to machine, and generally from compiler to compiler, as there is certainly no guarantee that (say) two Pascal compilers will store packed records in the same way. Thus low level data structure traces should be employed only if there is no alternative form of tracing available.

High level data structure traces

When programming in a high level language such as FORTRAN or Pascal, it should be possible for a user at all times to "think high level". That is to say, he should almost never have to consider how the machine is handling either his program or the data on which it operates. The only occasion when the high level language programmer should have to descend to code level is in order to optimize critical loops to speed up program execution; this in itself should be very infrequent for the vast majority of programmers.

The ideal situation is that if a data structure needs to be traced then the output from the trace should resemble the original high level source code as closely as possible, For example, the actual user-defined names of variables should appear (rather than their addresses); furthermore, the names of fields within records should be specified. In this way the programmer is freed from the burden not ony of having to understand the internal machine representation of his data structure, but also of having to think in terms of that internal representation, thereby defeating the whole purpose of programming in a high level language [8].

Very high level data structure traces

It can be very helpful for a user to "see" a data structure which he has conceived in the format in which he has conceived it. For example, if a user has conceptualized a two-dimensional array in terms of rows and columns, but has for some reason transposed the elements of such an array, then a graphical display will quickly show up his mistake. On the other hand, merely listing the array elements, even in a high level format, may not solve the problem; the user may simply not appreciate that (say) the first index, rather than the second, corresponds to "row number".

At a more advanced level, the fact that a language like Pascal allows literally an infinite number of different possible data structures means that very complex structures can arise; presenting a user with graphical output depicting his data in a form as close as possible to the way he "sees" it is again a quick and helpful debugging aid. (The reader will no doubt at this point recall the oft-quoted Confucian proverb relating the comparative worth of a picture and 1K words of natural language).

EXAMPLES OF DATA STRUCTURE TRACES

A Fortran array trace

The author has constructed an interactive graphical trace for FORTRAN arrays [6]. The system permits the contents of up to four arrays (of one or two dimensions) to be displayed simultaneously on a TEKTRONIX [5] graphics screen. At any one time no more than a 10 x 10 portion of each array can appear, but the entire array may be displayed piecewise by choosing the appropriate options. The user may decide how the arrays selected for tracing are to be arranged on the screen by means of the graphics cursor.

Typical output from the array trace is shown in Figure 1. Four arrays are being traced, a double-precision array DUBBLE(50,30), an integer array IARRAY(50,30), a complex array COMP(50,30) and a real vector A(100). The current value of any element appears in the top half of the corresponding

"box", which is labelled by its row and column index; a new value appears in the lower half. If the value again were to change, this would result in the entry in the lower half being overwritten; option 'R'(Refresh) causes the screen to be redrawn with the latest values once more in the top half of each rectangle, thus obviating the possibility of overwriting values.

The system is implemented in the form of a pre-processor which transforms the user's FORTRAN program into a FORTRAN program containing the relevant calls on subroutines which perform the plotting. It is written in FORTRAN IV, and hence is fully portable. For further details the reader is referred to Schach [6]; here we are more concerned with the level of this trace.

The fact that each array is specifically labelled on the screen with its name as given by the user in his or her FORTRAN source code, and that each element bears the appropriate row and column number means that we are dealing with a high level trace; the output is in a format closely resembling the original high level language source program. But further, since the data structures (arrays) are displayed in terms of their rows and columns as the user visualizes them means that this is in fact a very high level trace.

A trace for the Pascal heap

The package HEAPTRACE [7] is a precompiler for Pascal programs which enables the user to trace the heap, selectively dumping dynamically created records in a high level format. Each field of the record is named, and its value given in a form as close as possible to the original source code. Each record to be traced is assigned a sequence number as it is created on the heap, and these numbers not only provide unique identification during program execution, but are used when tracing pointer (e.g. POINTER P POINTS TO NODE — 456).

Figure 2 shows the output produced when HEAPTRACE was applied to the example on pages 44-46 of the Pascal User Manual [4]. When HEAPTRACE intervenes, the user is informed of the line number in his original Pascal program. When a node is dumped, its type identifier (in this case *person*) as named by the user is given. Then each component field is named, and its value given.

Integers, reals and characters are output in the conventional way, while the values of types defined by enumeration (including Boolean) are explicitly printed out. The user is informed if a field is of type set, and the contents of the set (if any) are printed out as above. For arrays, the indices and values of the first and last elements are printed. For example, if the program includes the declaration

specimen : array [-4..9,false..true,34..45] of real,

then the output from HEAPTRACE would include

specimen : array
specimen [-4,false,34] : 72.96
specimen [9,true,45] : -7.52

For packed arrays of char, the first and last character strings are given.

A record field within a record is identified as such, and its fields are in turn indented a further four spaces (see *birth* and *ddate* in Figure 2). Indentation is also used for tagfields (*ms* in the figure), and for the fields of variant parts.

But despite the fact that the underlying structure of each record is reflected through indentation, HEAPTRACE is not a very high level trace. The reason is that while the contents of any one individual record of the data structure are provided, the user is not given the overall picture of his data structure in the format in which he conceptualizes it. In terms of the terminology of this paper, HEAPTRACE is thus a strictly high level trace.

An interactive trace for the Pascal Heap

The Pascal pre-processor HEAPTRACE described in the section above provides information as to the contents of selected Pascal dynamic records. GRAPHTRACE, on the other hand, allows the user to "see" the overall shape of his data structures at a graphics screen. Each record is represented as a node, and the nodes are interconnected by directed links representing the pointers of the data structure. However, the contents of the individual fields do not appear on the screen.

Typical GRAPHTRACE output is shown in Figure 3a. Each record to be traced is assigned a number, as before. There are nodes of two distinct types, represented here by circles and diamonds respectively, corresponding to the two record types in the user's program. The three types of pointer defined in the program being traced are also distinguishable. The user is provided with a key to enable him to match the node or pointer to the name he gave it in his Pascal source.

The user specifies which subset of the records he wishes to see displayed in the current plot (or all of them), and may also select the root of the graph to be drawn. He may specify that certain links are to be drawn horizontally or drawn vertically, or are to be ignored. On the other hand, if he is unsure of the exact shape of his data structure he may simply allow the package to draw it as it sees fit, and the user will then refine his instructions stepwise, indicating that a particular record is the root of a tree, and so on. Figure 3b shows the same data structure as before, but with node 10 specified to be the root of a tree.

Figures 3c and 3d again show the identical data structure, but the user has specified various choices of directions for drawing his three types of pointer (or has chosen to suppress one type).

The method used for displaying the graph is used on the UDRL algorithm of Becker and Schach[1], but modified to allow for links which are neither horizontal nor vertical to be superimposed on the basic structure. For further details see Getz et al [2].

With regard to the level of GRAPHTRACE, the fact that the data structures may be displayed precisely as the user conceives them means that this is a very high level trace. But at the same time, GRAPHTRACE is not a high level trace.

The reason is that the only Pascal variable names with which GRAPHTRACE is concerned are the names of the types of the records, and of the pointers. As mentioned above, the contents of the nodes themselves are not displayed on the screen. Thus, strictly speaking, GRAPHTRACE is not a high level trace. However, GRAPHTRACE does allow the user to interact with the HEAPTRACE routines at any time. The user is permitted to dump selectively the contents of the heap at a printer or VDU screen.

By combining GRAPHTRACE with HEAPTRACE we thus have both a high level and a very high level trace, which together provide the user with maximal information for tracing the Pascal heap.

CONCLUSION

High level languages like Pascal or Ada[7] support powerful and flexible variable types, thus permitting highly sophisticated and complex data structures to be constructed. The price that must be paid for this is that if there is an error within a complicated data structure, then it is often not easy to detect and correct it. Use of a low level trace is entirely unsatisfactory, and at the very least a high level trace should be employed, and preferably a very high level one. Such traces do not exist for the new language Ada, but as soon as Ada compilers become available it would be advantageous for high level and very high level traces to be written.

```
 OPTIONS? <D,I,N OR R>     LINE NUMBER          38        38        38        52        52        52
>N                                              52
```

DUBBLE

	25	26	27
41	.ØØØØØ	.ØØØØØ	.ØØØØØ
42	.ØØØØØ	.ØØØØØ	.ØØØØØ
43	.ØØØØØ	.ØØØØØ	.ØØØØØ

IARRAY

	1	2	3
2	21	22	23
3	31	32	33
4	41 41	Ø 42	Ø 43

COMP

	3	4	5	6	
1	13.ØØØ	113.ØØ	14.ØØØ	114.ØØ	15.ØØ	115.ØØ	16.ØØØ	116.ØØ
2	23.ØØØ	123.ØØ	24.ØØØ	124.ØØ	25.ØØ	125.ØØ	26.ØØØ	126.ØØ
3	33.ØØØ	133.ØØ	34.ØØØ	134.ØØ	35.ØØ	135.ØØ	36.ØØØ	136.ØØ
4	.ØØØØØ 43.ØØØ	.ØØØØØ 143.ØØ	.ØØØØØ 44.ØØØ	.ØØØØØ 144.ØØ	.ØØØØØ 45.ØØØ	.ØØØØØ 145.ØØ	.ØØØØØ 46.ØØØ	.ØØØØØ 146.ØØ

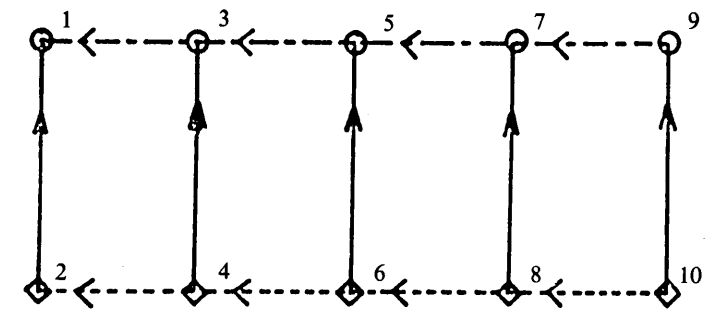
A

	1
3	3.ØØØØ
4	4.ØØØØ
5	5.ØØØØ
6	6.ØØØØ

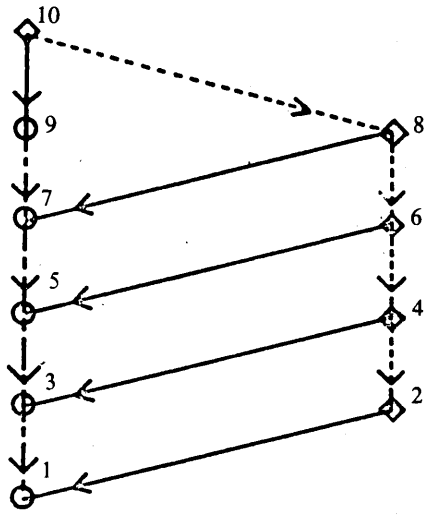
Figure 1: Sample FORTRAN Array Trace Output

```
***** HEAPTRACE CALLED AT LINE   44

NODE #  1     TYPE -  PERSON

 NAME           :     RECORD
     FIRST          :    ARRAY
                   STRING : EDWARD
     LAST           :    ARRAY
                   STRING : WOODYARD
 SS             :  845680539
 SEX            :  MALE
 BIRTH          :     RECORD
     MO             :  AUG
     DAY            :  30
     YEAR           :  1941
 DEPTDS         :  1
   MS             :  SINGLE
     INDEPDT        :  TRUE


NODE #  2     TYPE -  PERSON

 NAME           :     RECORD
     FIRST          :    ARRAY
                   STRING : NICOLAS
     LAST           :    ARRAY
                   STRING : ROBERTSMAN
 SS             :  627259003
 SEX            :  MALE
 BIRTH          :     RECORD
     MO             :  MAR
     DAY            :  15
     YEAR           :  1932
 DEPTDS         :  4
   MS             :  DIVORCED
     DDATE          :     RECORD
          MO             :  FEB
          DAY            :  23
          YEAR           :  1972
     FIRSTD         :  FALSE
```

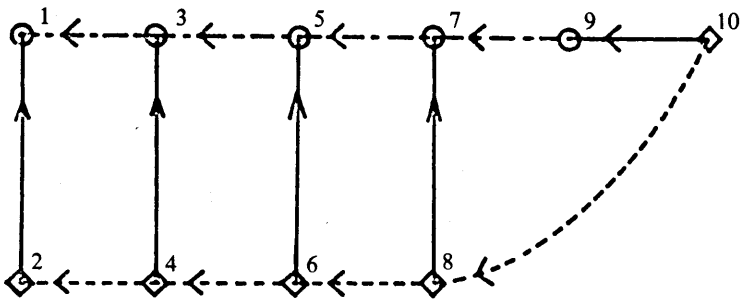
Figure 2: Sample HEAPTRACE Output.



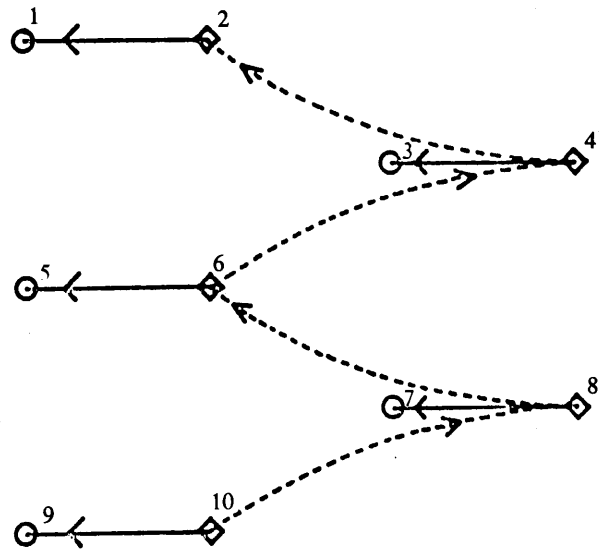
(a)



(b)



(c)



(d)

Figure 3: The same data structure drawn by GRAPHTRACE with various choices of the direction for the 3 types of pointers.

REFERENCES

[1] R. I. Becker & S. R. Schach, Drawing labelled binary graphs on a grid. Submitted to Networks.

[2] S. L. Getz, G. Kalligiannis & S. R. Schach, A very high-level interactive graphical trace for the Pascal heap. To appear in IEE Trans. on Software Engineering.

[3] J. D. Ichbiah et al, Preliminary Ada reference manua, ACM SIGPLAN Notices, Vol. 14, Number 6, 1979.

[4] K. Jensen & N. Wirth *Pascal User Manual and Report,* 2nd Edition, Lecutre notes in Computer Science 18, Springer-Verlag, Berlin, 1974.

[5] PLOT-10 Terminal Control System, User's Manual No. 062-1474-00, Tektronix, Inc., Beaverton, Oregon, 1974.

[6] S. R. Schach, An interactive graphical array trace. Quaest. Inform. Vol. 2, No. 1, pp 23-26.

[7] S. R. Schach, A portable trace of the Pascal heap, Software — Practice and Experience, Vol. 10, 1980, pp. 421-426.

[8] D. A. Watt & W. Findlay, A Pascal diagnostics system. In: *Pascal: The Language and its Implementation,* D. W. Barron, Editor, Wiley, Chichester, 1981.

Case-Grammar Representation of Programming Languages

Judy Mallino Popelas and Peter Calingaert
University of North Carolina at Chapel Hill, USA

Abstract
The correction of errors in programs can be based on an analysis, that subordinates syntactic relationships to functional relationships among elements of a program. For this purpose, *case grammars,* originally developed to model natural languages, have been adapted to model programming languages. The component parts of such a modified case grammar are described, and a case grammar for a subset of Pascal presented.

INTRODUCTION

The laudable goal of ensuring that a program is correct before it is presented to a computer is an elusive one even for experienced professional programmers. It is virtually unattainable for the rapidly increasing masses of persons for whom some computer programming is a necessary, but nevertheless part-time, activity. Errors occur both in *programming,* the design of an algorithm and the selection of data structures, and in *coding,* the representation of the algorithm and data structures in a programming language (PL). Although programming errors are of great importance, our research has focussed on the more mundane, but very aggravating, errors in coding.

Humans surely spend much less time encoding computer programs than they do communicating with one another by means of natural language (NL). We hypothesize that many of the errors humans make in encoding programs are similar to those they make in encoding thoughts into NL utterances. This suggests that techniques for correcting encoding errors in NL should help to correct coding errors in PL, and that useful models of NL representation should lead to useful models of PL representation.

Communication between humans can proceed effectively even when the utterances violate rules of syntax. Thus, "you was coming" and "they gave it to John and I" are clearly understandable, although incorrect. Even "today me shirt buy" is far from incomprehensible. The human who hears such an utterance does not immediately reject it because of its faulty syntax. He tries instead to understand it, using whatever nonsyntactic clues he can find. In translating computer programs with syntax errors, the compiler, too, can be made to use nonsyntactic clues to determine the underlying meaning when normal syntax correction would fail.

A particularly attractive model of NL, well capable of representing the meaning of the syntactically incorrect utterances presented in the previous paragraph, is the *case grammar* of Fillmore[1]. Case grammar (CG) concentrates on the underlying deep structure by associating with each verb a *case frame.* The case frame is occupied by one or more phrases, each of which plays a specific role demanded by the associated verb.

Thus the verb "buy" requires an agent who buys (a phrase in the *agentive* case) and an object that is bought (a phrase in the *objective* case). The phrases must often possess specified attributes; the agent of "buy" must be animate. In the example "today me shirt buy" the case frame requirement for an agent is filled by "me" and the case frame requirement for an object by "shirt".

Meaning can be extracted from the NL utterance without performing conventional syntactic analysis, by identifying the phrases that occupy the case frames. By adapting CG to PLs, which are much less complex than NLs, we expect to improve our ability to correct coding errors, and to perform correction without conventional syntactic analysis.

We have begun by developing a CG for Pascal and an algorithm for translating a syntactically incorrect program into its CG representation. We chose Pascal because it is formally defined [2], designed for efficiency of conventional translation [3], and widely used. Our adaptation of CG to PL is presented in Section 2, and illustrated in Section 3 by a subset of the Pascal grammar.

CASE GRAMMARS FOR PROGRAMMING LANGUAGES

Overview

Like their NL counterparts, CGs for programming languages emphasize functional rather than positional relationships between object phrases and verbs. Again like Fillmore's CGs, they emphasize an object's attributes rather than its form.

CGs for programming languages, hereafter referred to simply as CGs, have three basic components. The first component, an object space denotation, defines the objects found in a language in terms of attributes and attribute combinations. The second component, the verb dictionary, contains a *case frame entry* for each verb in the language. Each case frame entry is composed of a *header* and a *case frame.* The header gives the attributes of the object resulting from the filled case frame. The case frame itself describes the objects required by the verb, together with the functional relationship, or *case relation,* between each object and the verb. The constituent object descriptions in a case frame are called *case frame slots,* or more generally, *frame slots.* The final CG component, an access form dictionary, is similar to the verb dictionary. It is composed of *form frame entries,* which consist of a header and a *form frame.* Each form frame defines the access to an object other than a simple literal, by describing the objects that compose it or can be converted into it. Again, the object descriptions in a form frame are called frame slots, or more specifically, *form frame slots. Form relations* between the component objects and the resultant object may be specified. However, unlike case relations, which are often indicated explicitly via keywords and other PL markers, form relations generally are not indicated explicitly in PLs. Any keyword or PL marker that indicates a case or form relationship is called a *case* or *form relation predictor.* The header and frame of a case or form frame entry are analogous to the left- and right-hand sides of a context-free production.

Object Space Denotation

We use the word "object" to denote any entity that can be referred to or manipulated within the context of a given PL. Integer and real numbers, for example, are objects in most PLs. An object space is defined in large part by the attributes pro-

vided in a language, and by the combinations of attribute values that the objects in the language can possess. Attributes can be classified as being *universal* if they apply to every PL object, or *dependent* if they apply to only a subset of the PL objects.

Usage, class, and structure are three universal attributes. Access is a fourth universal attribute, but it is an attribute more of the frame slots than of the objects that satisfy frame slots. Usage refers to the way in which an object can be used. 'Value' usage indicates that an object can be used only as a value, whereas 'variable' usage indicates that an object can be used both as a value and as a store.

To be used, objects must be accessible. Commonly provided access methods include naming, referencing, direct representation, generation, and modification. Naming is one of the most common. Names, which have no inherent meaning, must be bound to a particular object before they can be used to refer to it. Languages commonly provide declaration parts or declaration statements for this purpose. References can be regarded as machine-generated names. They are usually used to refer to dynamically created variables. Direct representation differs from naming in that it is a permanent association between representation and object. In most languages, for example, '5' always represents the integer 5. Generation involves the execution of a sequence of one or more operators. The expression '2+4' generates the integer 6. Access by modification refers to the methods commonly used to refer to objects such as array or record components. Modification is similar to generation, except that no explicit operator is present.

Some access-usage combinations can be referred to by a single word. 'Literal', for example, refers to objects accessed by direct representation and used as values. Conversely, other access-usage combinations may encompass several distinguishable kinds of objects. Both array components and record components, for example, are accessed by modification and used as variables.

A *class* is defined as a set of scalar values and a set of case and form frame slots that accept those values. The restriction to scalar values effectively separates the concepts of class and structure, which together provide a complete and minimal set of concepts for describing any type of object. Some common classes of objects are integer, real character, boolean, verb, [verb], procedure, function, name, pointer, label, and class attribute. The notation '[verb]' stands for a verb together with the objects it requires. This constitutes a completed case frame (*i.e.*. a programming language statement or expression).

Most languages allow for structured as well as scalar objects. A structured object does not have a single associated class attribute. Rather, each of its component objects, if scalar, has an associated class attribute.

Some objects may have dependent attributes in addition to the four universal attributes. Exactly which dependent attributes an object possesses is determined by the value of some other attribute. Only objects whose class attribute is 'real', for example, have a precision attribute.

Specification of Object and Object-Phrase Requirements

An object phrase consists of one or more objects, plus preceding keywords and surrounding punctuation. Both case and form frame slots specify object-phrase requirements. Objects are the most important components of object phrases. Their requirements may be specified by stating permissible structure, class, usage, and access attribute values, as well as dependent attribute values. For example, the specificatin 'scalar, real,value,direct_representation' will be satisfied by objects like 1.0, 5.37, *etc.* The combination access-usage specification 'literal' can replace the separated specification 'value,direct_representation'. Some attributes may remain unrestricted. The specification 'scalar,real,value' places no restriction on the access method. Note that, since a variable can be used as a value, it satisfies a usage specification of 'value'. To force a restriction to non-variable objects, one could either specify 'value_only', or use an access-usage combination such as 'literal'. Alternative attribute values may be specified, as in 'scalar,integer|real,value'. Restrictions on the values of dependent attributes are specified in parentheses after the attribute value they modify, as in 'scalar,real(single_precision),value'.

Punctuation symbols are classified as predecessors if they precede objects, brackets if they bracket objects, separators if they separate like objects, and successors if they succeed objects. A *simple object phrase* is defined as a keyword, which acts as a case or form relation predictor, followed by a predecessor, a left bracket, an object or a sequence of like objects separated by separators, a right bracket, and a successor, in that order. Of these components, only a single object is mandatory. The text *'with* a, b, c' represents a simple object phrase. The keyword *'with'* is a case relation predictor; 'a', 'b', and 'c' represent like objects (record variables); and ',' acts as a separator.

To specify a simple object phrase, first specify the object, as already described. Attach a superscript to the object specification to indicate the number of like objects permitted in the object phrase. The superscript '+' indicates one or more like objects; '*' indicates zero or more like objects; 'op' indicates an optional object; and a positive integer indicates that number of like objects. The default value is unity. An encoding of the surrounding punctuation, enclosed by parentheses, is also attached as a superscript. A *case label,* denoting both the functional relationship (case relationship) of the object to the verb, and the particular keyword or other PL symbol, if any, that acts as the case relation predictor, is attached as a subscript. The case relation predictor appears parenthesized, after the case relationship. Some common case relationships are *indicant,* which specifies a name object used by a verb that binds names to other objects (variable declaration statements have indicant objects); *selector,* which indicates an object used by a verb to select among many possible objects (GOTO, IF, and CASE statements have selector objects); *donor,* which indicates an object whose value is given to another object (assignment statements have donor objects); and *objective,* a general case relationship that indicates an object that receives the action of a verb (operators such as +, −, and * have objective case objects). Thus, the specification

$$\text{record,variable}^{+}\ \underset{\text{selector}(\textit{with})}{(00,0)}$$

is satisfied by the simple object phrase 'with a, b, c', assuming 'a', 'b', and 'c' are the names of record variables. Note that '0' is used in the punctuation encoding to indicate the absence of a predecessor, brackets, and a successor. The generic form

$$\text{OBJECT}\underset{\text{CL}}{\text{MULT PUNCT}}$$

where OBJECT represents an object specification, MULT a specification of the number of like objects in the object phrase, PUNCT the punctuation encoding, and CL the case label, describes a simple object phrase specification.

A *complex object phrase* is defined as a case or form relation predictor, followed by a predecessor, a left bracket, one or more object phrases (simple or complex), or a repeated sequence of one or more object phrases separated by separators, a right bracket, and a successor, in that order. Of these components, only a single object phrase is mandatory. The text '[1:10]' represents a complex object phrase, where '1:' and '10' represent simple object phrases, and '[' and ']' are used as brackets. To specify a complex object phrase, first parenthesize the interior object phrase specification(s). Then attach the subscripts and superscripts to the parenthesized specification(s), in the same way as for a simple object phrase. The complex object phrase specification

(scalar,integer,literal$^{op(000:)}$
scalar,integer,literal)$^{+\ (0[],0)}$

is satisfied by either of the complex object phrases '[1:10]' and '[5,2:10]', and by many others as well. The generic form

$$(OBJ1 \ldots OBJN)^{MULTI\ PUNCT}_{CL}$$

describes the specification of a complex object phrase. OBJ1 ... OBJN represent specifications of simple or complex object phrases, and MULT, PUNCT, and CL represent exactly what they do in the specification of a simple object phrase.

Verb Dictionary

The verb dictionary contains one case frame entry for each verb in the language. Case frames are enclosed by square brackets. They contain specifications for each object phrase required to the verb. They also indicate the case relation of each object phrase phrase to the verb, even if that relation is not made explicit by a keyword or other PL marker. Following is a case frame entry for the GOTO verb.

scalar,[verb](GOTO,active,imperative,regular),literal
[scalar,label,value$_{selector\ (goto)}$]

GOTO requires one object, a label, which acts as a selector. The keyword *'goto'* precedes or predicts the selector object. CASE and IF statements also require selector objects, although these are predicted by different keywords.

In most languages, verbs and their corresponding completed case frames ([verb]s) will have significant dependent attributes. Four such attributes are discussed here: name, voice, mode, and influence. Name simply identifies the verb, as shown for GOTO. Voice may be 'active' or 'passive'. Passive voice indicates a verb, like the variable-creation verb, that can be executed at most once. Active verbs may be executed repeatedly. Verb mode may be 'imperative' or 'operator'. Operators include verbs like addition, multiplication, and binary selection (*i.e.,* the IF statement verb) that result in a single object. Imperative verbs, like assignment, result in changes to the environment. Verb influence may be 'regular' or 'meta'. 'Meta' indicates that the verb requires objects that are themselves statements. The case frame entry for the metaverb BINARY_SELECTION, without interior labels, is the following.

scalar,[verb](BINARY_SELECTION,active,
operator(scalar,[verb](,active,imperative,)),meta), literal
[scalar,boolean,value$_{selector(if)}$
[verb](,active,imperative,)$_{objective(then)}$
[verb](,active,imperative,)$^{op}_{objective(else)}$]

Besides the selector object, BINARY_SELECTION requires either one or two statement objects that are in the objective case, which receives the action of the verb. The objects must be active, imperative statements. The blank in the name and influence attribute positions indicates that any value for those attributes is acceptable. IF and other meta operator statements also satisfy the requirements because they ultimately generate active, imperative statements. Note that BINARY_SELECTION, because it is an operator, has dependent attributes that specify the structure and class of its resultant, generated object.

Verbs that require multiple objects often require agreement among two or more of them. We introduce *attribute variables* to express interobject dependencies. The attribute variables used in a given frame are implicitly created at the beginning of the frame, and remain accessible throughout the frame. Attribute variables are implicitly assigned values when they prefix an attribute restriction in an object specification. The specification 'scalar,CLSl:integer|real,value' causes the value of the class attribute of the object satisfying the specification to be assigned to the attribute variable CLSl. Attribute variables can be used without a specific attribute restriction, as in 'scalar, CLSI: , value'. The blank following 'CLSl:' indicates that there is no restriction placed on the class attribute. The colon indicates that the variable CLSl is to be assigned a value. When attribute variables appear without a succeeding colon, they specify a restriction to whatever attribute value they currently possess. Consider the case frame entry for assignment.

scalar,[verb](ASSIGN,active,imperative,regular),literal
[STRl: ,CLSl: ,variable$_{recipient}$
STRl,CLSl,value$_{donor(:=)}$]

When the recipient object phrase is encountered, the attribute variables STRl and CLSl are assigned values. By using STRl and CLSl to specify attribute values for the donor object phrase, agreement between the two objects is forced.

Conditional clauses may be used to modify a succeeding attribute value specification, simple object-phrase specification, or complex object-phrase specification. They consist of a predicate enclosed by '(= =)' brackets. The case frame for an ASSIGN verb that allows integers to be assigned to reals can be specified by using a conditional clause.

scalar,[verb](ASSIGN,active,imperative,regular),literal
[STRl: ,CLSl: ,variable$_{recipient}$
STR,CLSI| (=CLSI=real=) integar, value$_{donor(:=)}$]

Access Form Dictionary

The access form dictionary defines the access methods. For example, a complex literal like a procedure would have an access frame describing each of its component object phrases. For array components, which are accessed by modification, the access frame describes the array object and the objects that could be used as indices. Access frames specify transformations of objects to other objects.

Access frames are enclosed in angular brackets. Following is a simplified form frame entry for name access.

STRUCT, CLASS, USAGE, named
<name(bound(STRUCT: ,CLASS: ,USAGE:))>

It states that a bound name object may be transformed into an object whose structure, class, and usage attribute values are determined by the dependent attributes of 'bound'.

Semantics

Because a case grammar deals with language at the object level rather than at the symbol level, at least a partial definition of semantics is needed to make it complete. The semantics must specify the creation of objects, the association of attributes with objects, and the deletion of objects. A complete notation for defining the semantics in a case grammar for Pascal is given in the first author's dissertation[4]. The details of the notation are unimportant, since many other notations would have served as well. However, a simplified subset is presented here to enable the reader to understand the case grammar example presented in the next Section.

Semantic action statements are used to assign values to attribute variables explicitly. They are of the form 'attribute variable <— value', and are enclosed by '{}' brackets. They may appear anywhere in a case or form frame.

SYMTAB is a global attribute variable, accessible from any frame. It contains the kind of information commonly found in symbol tables, the association of names with the objects they represent. In particular, the value of SYMTAB will be a sequence of name literals, each with its associated dependent binding attribute. The value 'bound' has, in turn, three associated dependent attributes: structure, class, and usage. These give the structure, class, and usage of the object to which the name has been bound. Both 'unused —> (unbound)' and 'used —> (bound(scalar,integer,variable))' represent legitimate entries in SYMTAB. The operator '+||' will be used to add entries to SYMTAB. Similarly, '−||' is used to delete entries from SYMTAB. The verbs CREATE_PROG and CREATE_VAR demonstrate the +|| operation.

EXERPTS FROM A CASE GRAMMAR

We present here excerpts from a case grammar for a very small subset of Pascal. The subset includes an abvreviated program statement, the *var, begin,* assignment, *if,* and *while*

statements, and several operators. Labels are omitted. The boolean entities 'true' and 'false' are treated as literal values the lexical structure of simple literals such as integers and reals.

Object Space Denotation

The object space is defined by three tables. Table 1 lists the attributes used in the grammar, together with the values they may assume, Table 2 lists attribute dependencies, and Table 3 shows the co-occurrence of attribute values in objects. Because the language has only scalar objects, neither Table 1 nor the rest of the grammar includes a structure attribute.

Attributes	Values
class	integer, real, boolean, name, class__attribute, verb, [verb], program
usage	value, variable
access	directly__represented, generated, named
access-usage	literal, generated__value, named__constant, named__variable
binding	bound, unbound
voice	active, passive
mode	imperative, operator
influence	regular, meta

TABLE 1: Attributes and Values

Dependency	Attributes
name	binding
bound	class, usage
verb	name, voice, mode, influence
operator	class

TABLE 2: Attribute Dependencies

	CLASS							
ACCESS-USAGE	1	2	3	4	5	6	7	8
literal	X	X	X	X	X	X	X	X
generated__value	X	X	X				X	
named__constant								X
named__variable	X	X	X					

where 1: integer, 2: real
3: boolean, 4; name
5: class__attribute, 6: verb
7: |verb|, 8: program

TABLE 3: Object Availability

Verb Dictionary

[verb](CREATE__PROG,passive,imperative,meta),literal
[{ SYMTAB <—nul }
name(unbound)$^{(000;)}_{indicant(program)}$
{ SYMTAB <— + || VALUE(indicant)—>
(bound(program,value)) }
program,literal$^{(000.)}_{base}$]

VALUE is an operator that can be applied to objects to yield their value. In CREATE__PROG, VALUE returns the actual name literal used to satisfy the indicant case object requirement. In the CREATE__VAR frame, VALUE is used to yield a class value.

[verb](CREATE__VAR,passive,imperative,regular),literal
[(name(unbound)$_{indicant}$
{ SYMTAB <—SYMTAB + || VALUE(indicant—>
(bound(VALUE(specifier),variable))|)$^{+(00,:)}$
class__attribute$_{specifier}$)$^{+(00;0)}_{objective(var/}$]

[verb](COMPOUND,active,imperative,metal),literal
[[verb] (,active,imperative,)$^{+(00;end)}_{objective(begin)}$]

[verb](ASSIGN,active,imperative,regular),literal
[CLASS: ,variable$_{recipient}$
CLASS| {CLASS = real } integer, value$_{donor(:=)}$]

[verb] (BINARY__SELECTION,active,
operator([verb](,active,imperative,)),meta),literal
[boolean,value$_{selector(if)}$
[verb](, active, imperative,)$_{objective(then)}$
[verb](,active,imperative,)$_{objective(else)}$]

[verb] (REPETITION,active,imperative,meta),literal
[boolean,value$_{governor(while)}$
[verb](,active,imperative,)$_{objective(do)}$]

[verb] (ADD,active,operator(CLASS),regular),literal
[CLASS1:integer|real,value$_{objective}$
CLASS2:integer|real,value$_{objective(+)}$
{ CLASS <— integer
(= CLASS1 = real | CLASS2 = real =) CLASS <—real}]

The case frames for the other regular operators in the language are not shown.

Access Form Dictionary

program,literal
< [verb](CREATE__VAR, , ,)$^{op(000;)}$
[verb](COMPOUND, , ,),literal>
CLASS,generated__value
< [verb](,active,operator(CLASS:),) >
CLASS,USAGE,named
< name(bound(CLASS: ,USAGE:)) >

CONCLUSION

Case grammars define PLs in terms of objects and verbs, and their relationships to each other. Although the notation is capable of defining the syntax completely, the emphasis remains at the object rather than at the symbol level. By following CG as a model, we may be able to design PLs that incorporate some features of NL and are therefore more comfortably used. Multiple surface structures can be allowed, perhaps permitting multilingual translators. CGs can also serve as a vehicle for comparing PLs concentrating on their deep representational abilities rather than on their surface structures. Nevertheless, the most important application of CGs offer the following advantages over context-free grammars. First, syntactic details are clustered into punctuation encodings, and can easily be ignored. If errors occur at this level, they are likely to have a minimal effect on the parser's functioning. Second, attribute-value information is stressed, whereas in most context-free grammars and parsers it is ignored. Finally, functional case relationships are emphasized over positional relationships. This suggests that errors of position can be well tolerated.

REFERENCES

[1] C. Fillmore, The case for case. In: *Universals in Linguistic Theory,* E. Bach and R. Harms, Editors, new York, Holt, Rinehart, and Winston, 1968, pp. 1-88.

[2] C.A.R. Hoare and Wirth, N. An axiomatic definition of the programming language Pascal. In: *Acta Informatica,* vol. 2, fasc. 4, 1973, pp. 335-355.

[3] K. Jensen and N. Wirth,*PASCAL User Manual and Report,* New York, Springer-Verlag, 1975.

[4] J. M. Popelas, Ph.D. dissertation, University of North Carolina at Chapel Hill, 1981.

Die Definisie en Implementasie van die taal Scrap

Martha H. van Rooyen
NNWW, WNNR.

Samevatting
SCRAP is 'n hoëvlaktaal geskik vir stelsel- sowel as toepassingsprogrammering. Die uitstaande kenmerke van die taal is 'n aantal goedgedefinieerde sintaktiese konstruksies, 'n modulêre struktuur, masjienafhanklike datatipes en 'n meganisme vir kommunikasie met die onderliggende stelsel. SCRAP word selfvertalend geïmplementeer. Die vertaler is in drie gange verdeel, te wete sintaksontleding en semantiese ontleding, wat saam die masjienonafhanklike deel daarvan uitmaak, en kodegenerasie, die masjienafhanklike deel.

Abstract
SCRAP is a high-level language that provides facilities for systems programming, but can equally well be used for applications programming. The prominent features of the language are a well-defined syntax, a modular structure, machine-dependent data types, and the possibility of communication with the underlying system. SCRAP is to be implemented by a self-compiling compiler. Compilation consists of three passes: syntax analysis and semantic analysis, forming the machine-independent part; and code generation, the machine-dependent part.
ACM Reviews — kategorie: 4.22

1. Inleiding

Programmatuur kan as stelselprogrammatuur of toepassingsprogrammatuur geklassifiseer word. In hierdie referaat word met 'stelselprogramme' programme bedoel wat deel van 'n rekenaarstelsel uitmaak en wat ondersteunende fasiliteite aan al die gebruikers van die stelsel bied, terwyl toepassingsprogramme nie deel van die stelsel as sodanig is nie.

SCRAP ('Systems Construction and Applications Programming Language') is 'n hoëvlaktaal wat by uitstek vir stelselprogrammering op minirekenaars geskik is, maar ook algemeen genoeg vir toepassingsprogrammering is [1]. SCRAP is gedefinieer en ontwikkel om so goed as moontlik aan die kriteria van doeltreffendheid, algemeenheid, betroubaarheid, onderhoubaarheid en oordraagbaarheid te voldoen. Aangesien doeltreffendheid en oordraagbaarheid in 'n mate teenstrydige kriteria is, is die uitweg van pseudo-oordraagbaarheid gevolg. As gevolg van pogings om optimale doeltreffendheid in elke aparte omgewing te verkry, mag die voorkoms van SCRAP van een rekenaaromgewing na 'n ander verskil. 'n SCRAP-program is dan oordraagbaar tussen die omgewings in soverre die omgewings versoenbaar is.

Die kenmerkende eienskappe van SCRAP word kortliks bespreek, met die klem op dié eienskappe wat vir stelselprogrammering van belang is. 'n Kritiese beskouing van die mate waartoe SCRAP aan sy ontwerpskriteria voldoen, word gegee. Daarna volg 'n oorsigtelike bespreking van die tegnieke wat vir die eerste implementasie van SCRAP gebruik is. Vir meer besonderhede word die leser na [1] verwys.

2. Die Taal SCRAP

Slegs sommige van die uitstaande kenmerke van SCRAP word kortliks hier bespreek, te wete unieke afsluitsleutelwoorde, modules, masjienafhanklike datatipes en kommunikasie met die onderliggende stelsel.

2.1 Unieke afsluitsleutelwoorde

Alle komplekse sintaktiese konstruksies word deur unieke begin- en afsluitsleutelwoorde afgebaken. Die afsluitsleutelwoord vir 'n gegewe konstruksie word gevorm deur 'end' aaneen te skakel met die eerste letter van die beginsleutelwoord, bv. 'loop...endl', 'if...endi'. 'n Natuurlike en leesbare programuitleg word sodoende verkry. As gevolg van die gebruik van beheerstrukture wat op dié manier afgebaken is, kon in SCRAP die konsep van 'n saamgestelde stelling afgebaken deur 'begin...end'-hakiewoorde, asook die gepaardgaande probleme, vermy word. Maklike sintaksontleiding en fouthantering is 'n verdere voordeel van die unieke afsluitsleutelwoorde.

2.2 Modules

Die konsep van modulariteit dra aansienlik tot die oordraagbaarheid, betroubaarheid en onderhoubaarheid van programmatuur by, met die voorbehoud dat die modulesterkte hoog moet wees en die koppeling tussen modules laag [2]. Dié konsep is op 'n unieke manier in SCRAP verwenslik.

'n Program bestaan uit een of meer modules, wat saam of afsonderlike vertaal kan word. 'n Module word uit 'n aantal globale blokke opgebou, nl. letterlike blokke, datablokke en prosedures. Die blokke kan sintakties in enige volgorde voorkom. Een en slegs een van die modules moet 'n inisialiseerblok aan die einde daarvan insluit, waar programuitvoering begin. Die raamwerk van 'n module word in Figuur 1 uiteengesit.

Die programmeerder kan absolute adresse aan datablokke en prosedures toeken, wat spesifiseer dat die betrokke datablok of prosedure vanaf die betrokke adres gelaai moet word. Hoe die absolute adresse met laaityd op geheue-adresse afgebeeld word, is masjienafhanklik. Die adres as sodanig het geen betekenis vir die vertaler nie, en dit berus by die programmeerder om dit op die regte manier te gebruik. Absolute adresse word benodig vir onderbrekingstabelle en -prosedures, statuswoorde, ens. Hierdie fasiliteit is dus 'n vereiste in 'n omgewing waar produksie-stelselprogramme wat direk op die apparatuur van die teikenrekenaar moet uitvoer, ontwikkel word.

Prosedures is by versuim dinamies; 'n prosedure mag egter eksplisiet as staties verklaar word. In die geval van dinamiese prosedures word stoorplek vir lokale gebruik op 'n uitvoertydstapel gereserveer elke keer as die prosedure geroep word; in die geval van statiese prosedures word lokale stoorplek in statiese geheue gereserveer wanneer die program gelaai word. 'n Statiese prosedure se lokale veranderlikes bly dus van een roep na 'n volgende behoue. Dinamiese prosedures is multitoeganklik en kan rekursief geroep word; daarenteen kan statiese prosedures nie rekursief geroep word nie en is slegs multitoeganklik as die teikenrekenaar die fasiliteit bied om die data en stellings van 'n program in afsonderlike geheue-areas te stoor. Statiese prosedures verkry beheer wanneer onderbreking-inkom en kan dus nie dinamies wees nie. Dit mag ook wenslik wees om statusinligting van een onderbreking na 'n volgende te behou.

'n Module kan as 'n heining beskou word rondom die modules waaruit dit bestaan; die module versteek sodoende inligting aangaande sy blokke van ander modules. Modules kommunikeer met mekaar op die globale blokvlak. 'n Module maak datablokke en prosedures vir ander modules toeganklik deur dit eksplisiet uit te voer; 'n module verkry toegang tot

datablokke en prosedures wat in ander modules verklaar is deur dit eksplisiet in te voer. 'n Blok wat deur een module uitgevoer word, is bekend slegs in dié modules wat dit invoer; desgelyks kan 'n blok slegs ingevoer word as dit deur 'n ander module uitgevoer word.

Modules soos in SCRAP gedefinieer, verskaf 'n werktuig om programme met 'n hoë modulesterkte en lae modulekoppeling te implementeer. Prosedures wat 'n funksionele verbintenis het, en die data waarop hulle bewerkings uitvoer, kan saam in 'n module gegroepeer word; die mate waartoe modules gekoppel is, blyk duidelik uit die globale blokverklarings.

'n Nadeel van die ekslisiete invoer en uitvoer van blokke is dat dit in sommige gevalle tot heelwat ekstra skryfwerk kan lei. As 'n datablok ingevoer word, moet al die verklarings daarin herhaal word om die eienskappe van die data binne die module bekend te stel; as 'n prosedure ingevoer word, moet die hele opskrif daarvan herhaal word. SCRAP bied geen standaard-prosedure vir toevoer/afvoer en rekenkundige funksies nie; indien 'n biblioteek van sulke funksies vir gebruik in 'n bepaalde omgewing bestaan, moet die toepaslike prosedures in elke module wat dit gebruik, ingevoer word. Bogenoemde probleme kan oorbrug word deur die gebruik van 'n programmatuurontwikkelingstelsel op die teikenrekenaar om die wisselwerking tussen programmodules te administreer; slegs die name van datablokke en prosedures hoef dan ingevoer te word. So 'n ontwikkelingstelsel word in [3] beskryf.

2.3 Masjienafhanklike Datatipes

'n Verdere uitstaande kenmerk van SCRAP is die masjienafhanklike definisies van die basiese tipes vir heelgetal-, wisselpunt- en adresdata. Hierdie eienskap het 'n aansienlike invloed op die doeltreffendheid en pseudo-oordraagbaarheid van SCRAP.

Ten einde aan die vereiste van doeltreffendheid ten opsigte van geheuekompleksiteit te voldoen, moet elke SCRAP-tipe in 'n gegewe rekenaaromgewing eenduidig en sonder verlies aan doeltreffendheid of noukeurigheid op 'n apparatiese tipe afgebeeld kan word. Die noukeurigheid asook die beskikbaarheid al dan nie van die basiese datatipes hang van die apparatuur van die teikenrekenaar af; as 'n tipe nie apparaties ondersteun word nie, is dit uitgesluit uit die SCRAP vir die bepaalde omgewing.

Die volledige stel basiese tipes sluit die volgende in: heelgetal van normale lengte; heelgetal met lengte kleiner as normaal; heelgetal met lengte groter as normaal; greep; karakter; wisselpunt en dubbelnoukeurige wisselpunt. 'n Voorbeeld van hoe die basiese tipes op verskillende rekenaars daar kan uitsien, word in Figuur 2 gevind. Die adrestipe word as een van die heelgetaltipes gerealiseer, afhangende van hoe groot die teikenrekenaar se adresseerbare geheue is.

2.4 Kommunikasie met die Onderliggende Stelsel

'n Program voer op 'n onderliggende stelsel uit wat oor die nodige fasiliteite vir kommunikasie met sy omgewing beskik. Die enigste manier waarop 'n program met sy omgewing kan kommunikeer, is via die fasiliteite van die stelsel, hetsy direk of indirek. Wat die onderliggende stelsel alles behels, asook die fasiliteite wat dit aan die gebruikers daarvan verskaf, is geheel en al van die omgewing onder beskouing afhanklik — dit kan wissel van slegs apparatuur tot 'n gesofistikeerde bedryfstelsel. Ten einde die stelsel se fasiliteite te kan benut, is 'n koppelvlak nodig, wat noodwendig masjienafhanklik is.

By die ontwerp van die meeste hoëvlaktale is gepoog om 'n aantal masjienonafhanklike fasiliteite vir toevoer/afvoer, gelyktydige programmeering, ens. te definieer, wat dan masjienafhanklik geïmplementeer word; die gevolg is 'n onvolledige en dikwels ondoeltreffende koppelvlak na die stelsel, en 'n gebrek aan oordraagbaarheid van die taal self. Om aan SCRAP se vereiste vir doeltreffendheid en uitdrukkingsvermoë te voldoen, moet al die stelselfasiliteite in 'n bepaalde omgewing tot beskikking van die gebruiker wees, met die minimum bokoste.

Die filosofie wat by die ontwerp van SCRAP gevolg is, is om slegs 'n meganisme vir die koppelvlak na die onderliggende stelsel daar te stel. Dié meganisme is die 'svc' -stelling. Dit neem die vorm van 'n spesiale prosedureroep aan — die 'prosedure' wat geroep word, is die stelsel self. Die aantal en tipes van die parameters wat die roep vergesel, asook die manier waarop die roep geïmplementeer word, is masjienafhanklik. Die parameters spesifiseer die versoek aan die stelsel, die manier waarop dit uitgevoer moet word, die data-areas wat betrokke is, ens.

Indien die stelsel slegs uit apparatuur bestaan, kan 'n aantal 'pseudo-svc'-stellings vir die omgewing gedefinieer word, wat dan met vertaaltyd na die ooreenstemmende objekkode omgeskakel word. Normaalweg sal die stelsel egter 'n bedryfstelsel insluit, in welke geval die 'svc'-stelling na ooreenstemmende roepe na die bedryfstel omgeskakel kan word. Die implementasie van SCRAP op die Perkin-Elmer 3220 kan as 'n verteenwoordigende voorbeeld beskou word. Roepe na die OS/32MT-bedryfstelsel moet op lae vlak as 'n opdragkode met twee operande gespesifiseer word — die eerste operand is 'n heelgetal wat die versoek aandui, en die tweede is die adres van 'n parameterblok. Die 'svc'-stelling se parameters stem met dié twee operande ooreen, en die stelling kan sonder enige bokoste na die ooreenstemmende objekkode omgeskakel word.

Die 'svc'-stelling, ten koste van oordraagbaarheid van die programmatuur wat dit gebruik, voldoen dus in 'n hoë mate aan die vereiste vir doeltreffendheid en algemeenheid. 'n Bykomende voordeel is dat dit maklik is om te implementeer en dat oordragkoste wat met die 'svc'-stelling as sodanig verbonde is, laag is.

3. Die Implementasie van SCRAP

SCRAP word self-vertalend geïmplementeer. Met die doel om die oordraagbaarheid van die SCRAP-vertaler te bevorder, is die vertaler in twee basiese dele verdeel, wat onderskeidelik masjienonafhanklik en masjienafhanklik is. Die masjienonafhanklike deel aanvaar 'n program in die brontaal, voer die masjienonafhanklike funksies van die vertalingsproses uit en lewer 'n intermediêre voorstelling van die bronprogram. Die masjienafhanklike deel aanvaar dan dié intermediêre voorstelling, hanteer die vertalingsaspekte afhanklik van die betrokke omgewing en produseer die ooreenstemmende teikenkode. Die masjienonafhanklike deel bly basies dieselfde vir alle omgewings waarin SCRAP geïmplementeer word; dit sal slegs ten opsigte van die masienafhanklike konsepte van SCRAP verskil, byvoorbeeld lengtes van datatipes en toevoer-/afvoer meganismes. Daarenteen moet die masjienafhanklike deel vir elke omgewing oorgeskryf word. Beide dele word in SCRAP geskryf.

Die skoenriemproses vir die eerste implementasie van SCRAP word vervolgens uiteengesit. Die metodes wat in die ontwikkeling van die vertalers gebruik word, word oorsigtelik bespreek aan die hand van die eerste vertaler in die skoenriem; slegs die uitstaande kenmerke daarvan word uitgelig. Die metodes bly basies dieselfde vir al die vertalers in die skoenriem.

3.1 Die Skoenriem vir die Eerste SCRAP-vertaler

Die eerste SCRAP-vertaler word op 'n Perkin—Elmer 3220-rekenaar geïmplementeer, met behulp van 'n drievoudige skoenriem. Pascal is as implementasietaal vir die eerte vertaler in die skoenriem gekies, wat 'n deelversameling van SCRAP ter ondersteuning van die uiteindelike selfvertalende vertaler implementeer. Die skoenriem word met behulp van die notasie in Figuur 3 diagrammaties in Figuur 4 voorgestel [4].

3.2 Die Struktuur van die Vertaler

Die vertaler is in drie gange verdeel, naamlik leksikale en sintaksontleding, semantiese ontleding en kodegenerasie. Die eerste twee gange vorm die masjienonafhanklike, en die derde die masjienafhanklike deel. Die natuurlike fases van die vertalingsproses is so ver moontlik in die verdeling behou; dit het die voordeel dat die funksie van elke gang duidelik afgebaken is en dat die intermediêre tale maklik gedefinieer kan word. In die finale vertaler sal 'n optimerende gang na gang twee ingevoeg word, en moontlik ook na gang drie.

Die brontaal vir die eerste skoenriemvertaler is 'n deelver-

sameling van SCRAP voldoende om die volgende vertaler te implementeer; konstruksies wat slegs betreklik moeilik geïmplementeer kan word, is uit die deelversameling uitgesluit. Die teikentaal is die Perkin-Elmer samestellertaal CAL. Die twee intermediêre tale is slegs voorstellings van 'n gedeeltelik vertaalde bronprogram; hierdie tale is masjienonafhanklik en voldoende vir alle implementasies van SCRAP.

Die sintaksontleder en semantiese ontleder lees toevoersimbole een-vir-een in soos dit benodig word, en genereer afvoer tydens die ontledingsproses. Die kodegenerasiegang lees 'n basiese blok in, en genereer afvoer stelling-vir-stelling. Die simbooltabel bestaan slegs in die tweede gang; semantiese inligting word saam met die items waarop dit van toepassing is, in die intermediêre kode ingevoeg.

Die SCRAP-produksiereëls voldoen aan die LL(I)-eienskap [5], op een uitsondering na — daar kan nie uit sintaktiese oorwegings tussen 'n enkelvoudige veranderlikenaam en 'n parameterlose funksieroep onderskei word nie. Hierdie beslissing word uitgestel tot die semantiese fase.

'n Prosedure mag geroep word voordat dit verklaar word. Hierdie vorentoe-verwysings word soos volg opgelos. Die sintaksontleder skryf alle inligting wat op prosedure-opskrifte betrekking het, na 'n afsonderlike lêer. Die semantiese ontleder lees dan hierdie inligting vooraf in, en doen inskrywings daarvoor in die simbooltabel. Sodoende kan tydens semantiese ontleding alle prosedureroepe op dieselfde manier hanteer word.

Die manier waarop die funksies van die gange en die koppelvlakke tussen die gange gedefinieer is, het aansienlik tot 'n spoedige implementasie van die vertaler bygedra. Die intermediêre voorstellings is wel groot, maar dié nadeel word oortref deur die vereenvoudiging van die gange self.

3.3 Sintaksdiagramme

Die sintaks van sowel die brontaal as die intermediêre tale word met behulp van sintaksdiagramme beskryf]6[. 'n Sintaksdiagram is 'n grafiese voorstelling van BNF-notasie en bestaan uit 'n gerigte grafiek met nodusse wat die terminale en nieterminale simbole van die ooreenstemmende BNF-produksiereël voorstel. Die SCRAP-produksiereël vir 'n module, omskryf in BNF-notasie en 'n sintaksdiagram, word in Figuur 6 gegee.

Sintaksdiagramme is 'n stelselmatige en doeltreffende werktuig vir die ontwerp van 'n vertaler. Dit is gebruik om die sintaks van die brontaal te spesifiseer; om die transformasie van elke gang op sy toevoerdata voor te stel; om die sintaksontleder te konstrueer; en om fouthantering in die sintaksontleder te implementeer.

'n Beter insig in die gebruik van sintaksdiagramme kan verkry word deur een produksie te bestudeer. Die 'loop'-stelling is vir die doel geskies. Die drie sintaksdiagramme vir dié konstruksie word in Figuur 6 gegee. Hoe die diagramme in die sintaksontleder gebruik is, word in die volgende paragraaf bespreek.

3.4 Gang 1 — Leksikale en Sintaksontleding

Die eerste gang konstrueer die bo-af, mees linkse afleiding van die bronprogram en beeld terselfdertyd die bronprogram af op die eerste intermediêre voorstelling daarvan.

Die leksikale ontleder (taster) is as 'n prosedure van die sintaksontleder geïmplementeer en word geroep wanneer 'n simbool benodig word. Aangesien die terminale simbole deur 'n reëlmatige grammatika beskryf word, is die taster as 'n deterministies eindige-toestandoutomaat geïmplementeer [7,8].

Sintaksontleding word met behulp van rekursiewe afdaling gedoen [7,8]. Die sintaksontleder bestaan hoofsaaklik uit 'n aantal sintaksprosedures, wat ooreenstem met die sintaksdiagramme van die brontaal. Die prosedures is eenvoudig en direk vanaf die betrokke sintaksdiagramme gekodeer. Vir elke nie-terminale nodus in die toevoerdiagram word die ooreenstemmende prosedure geroep; vir elke terminale nodus word afvoer gegenereer soos beskryf deur die ooreenstemmende nodus in die afvoerdiagram; en die vloei van beheer word deur die gerigte sye van die diagram, tesame met die volgende toevoersimbool, bepaal.

Die algoritme vir fouthantering wat in die sintaksontleder ingebou is, is eenvoudig en stelselmatig en lewer uitstekende resultate. Dit berus daarop dat toevoersimbole na 'n sintaksfout oorgeslaan word totdat 'n sleutelsimbool teëgekom word, vanwaar sintaksontleding weer normaalweg kan voortgaan. Die sleutelsimbole vir elke sintaksdiagram by elke punt daarvan kan direk neergeskryf word; dit sluit in die terminale en die handvatsels van die nie-terminale in die diagram. Indien 'n sintaksfout herken word, word die sleutelsimbole by die betrokke punt saam met 'n aanduiding van die tipe fout na 'n fouthanteringsprosedure oorgedra. Die roetine skryf 'n foutboodskap en slaan toevoersimbole oor tot by die eerste toevoersimbool wat met een van die sleutelsimbole ooreenstem. Die skema impliseer dat elke sintaksprosedure die sleutelsimbole van sy roepende prosedure as 'n parameter moet ontvang; lokale sleutelsimbole word dan bygevoeg as die fouthanteringsprosedure of 'n ander sintaksprosedure geroep word. Die unieke afsluitsleutelwoorde van SCRAP het grootliks tot die suksesvolle implementasie van die algoritme bygedra.

3.5 Gang 2 — Semantiese Ontleding

Die tweede gang aanvaar die eerste intermediêre voorstelling van 'n program, voer alle masjienonafhanlike semantiese toetse daarop uit, los alle vorentoe-verwysings op en genereer die tweede intermediêre voorstelling.

Alle tabelle met semantiese inligting word in dié gang opgestel en gebruik. Die simbooltabel bevat inskrywings vir die identifiseerders van die bronprogram, terwyl die tipe-inligting wat met dié identifiseerders verband hou, in 'n tipe-tabel gestoor word. Inligting wat op struktuurvelde en vorentoe-verwysings betrekking het, word in afsonderlike tabelle gehou.

Vir die verklarende gedeeltes in die toevoer word die nodige semantiese toetse uitgevoer en die toepaslike inskrywings in die tabelle gedoen.

Die semantiese ontleding van uitvoerbare kodes word met behulp van 'n vertaaltydstapel gedoen; die beskrywing van bewerkings word terselfdertyd van tru-Poolse notasie na drietalle omgeskakel (Figuur 6).

3.6 Gang 3 - Kodegenerasie

Die derde gang beeld die tweede intermediêre voorstelling van 'n program op die toepaslike teikenkode af, en voer ook alle masjienafhanklike semantiese toetse uit.

Kodegenerasie vir verklarings is eenvoudig, aangesien alle semantiese inligting aangaande die data in die intermediêre kode verskyn. Wat die uitvoerbare stellings betref, word kode vir 'n basiese blok op 'n slag gegenereer. Wanneer die begin van 'n prosedure se uitvoerbare gedeelte of 'n inisialiseerblok herken word, word basiese blokke ingelees en verwerk totdat die einde van die prosedure of inisialiseerblok herken word. Vir elke basiese blok word volgende-gebruikinligting versamel, en vervolgens word kode vir elke drietal in die blok gegenereer. Die kodegenerasie vir 'n drietal behels hoofsaaklik die toewysing van 'n register waarin die bewerking uitgevoer kan word, sowel as die bepaling van die opdragkode(s) en die adressering van die operand(e) vir die bewerking.

4. Gevolgtrekkings

Die eerste vertaler is reeds geruime tyd in gebruik, en te oordeel na die sukses daarvan, lyk dit asof SCRAP wel aan sy ontwerpsvereistes voldoen. Wat bedoel is om slegs 'n skoenriemvertaler te wees, is intussen met welslae in verskeie ander toepassings gebruik. Die meeste van die leemtes wat tans bestaan, sal deur die volledige SCRAP aangevul word.

SCRAP is maklik om aan te leer en te verstaan. Persone wat alreeds in ander hoëvlaktale geprogrammeer het, het SCRAP binne 'n dag of twee bemeester.

Die modulêre struktuur dra aansienlik tot stelselmatige en oordraagbare programimplementasie by, veral waar meer as een persoon besig is om aan 'n projek te werk. Verder verskaf SCRAP voldoende uitdrukkingsvermoë vir elegante programimplementasie.

Die definisie en implementasie van SCRAP het getoon hoe moeilik dit is om 'n taal daar te stel wat beide doeltreffend en

masjienonafhanklik is. By die implementasie op die Perkin-Elmer kan die invoer en uitvoer van blokke uit modules byvoorbeeld nie vir kommunikasie tussen multitake wat dinamies gelaai word, gebruik word nie; die teikenkode moet eers geredigeer word om daarvoor voorsiening te maak. Wat die vertaler self betref, moes sekere masjienafhanklike inligting, byvoorbeeld lengtes en gerigtheid van datatipes, noodwendig in die masjienonafhanklike deel gebruik word.

```
MODULE modulenaam;
  LIT
    (*letterlike definisies*)
  ENDL:
  DATA databloknaam;
    (*dataverklarings*)
  ENDD databloknaam;
  PROC prosedurenaam prosedure-opskrif;
    (*prosedure*)
  ENDP endprosedurenaam;
    (*verdere letterlike blokke, datablokke en prosedures*)
BEGIN
  (*inisialiseerstellings*)
ENDM modulenaam.
```

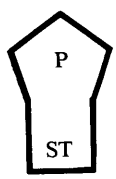
FIGUUR 1 — Die Raamwerk van 'n Module

	Perkin-Elmer 3220	PDP 11/45	Univac 1110
INT	32	16	36
SINT	16	-	-
LINT	-	32	72
BYTE	8	8	-
CHAR	-	-	6
REAL	32	32	36
DREAL	64	64	72

FIGUUR 2 - Noukeurigheid van Basiese Tipes (Lengtes in bisse)

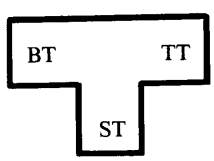
P = program
ST = skryftaal
BT = brontaal
TT = teikentaal
IT = interpreteerbare taal
MT = masjientaal
——► = kopieer
◄——► = hierdie programme is identies
----► = produseer met die hand (die veranderde taal word aangedui)
⌒► = produseer deur vertaling.

'n Rekenaar, sy masjientaal en sy samestellertaal word almal deur dieselfde notasie aangedui.

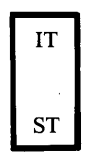


'n program

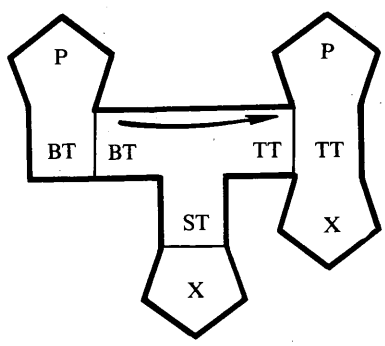
'n rekenaar



'n vertaler



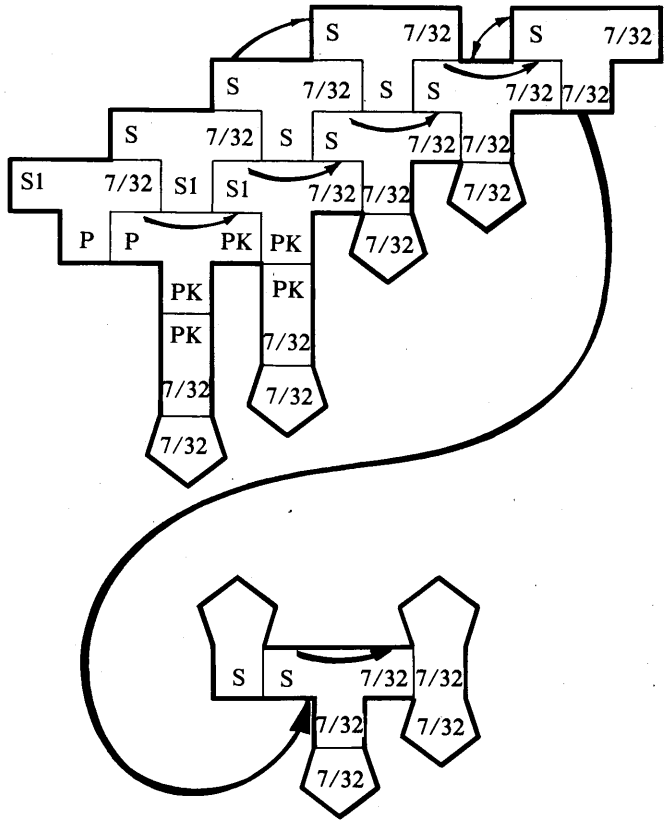
'n interpreteerder



Vertaling en uitvoering van program P, in BT geskryf, op masjien X.

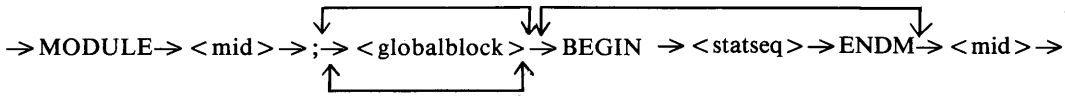
FIGUUR 3 - Notasie vir 'n Skoenriem.

P = Pascal-4A
PK = P-kode
S1 = deelversameling van SCRAP
S = volledige SCRAP
7/32 = Interdata 7/32

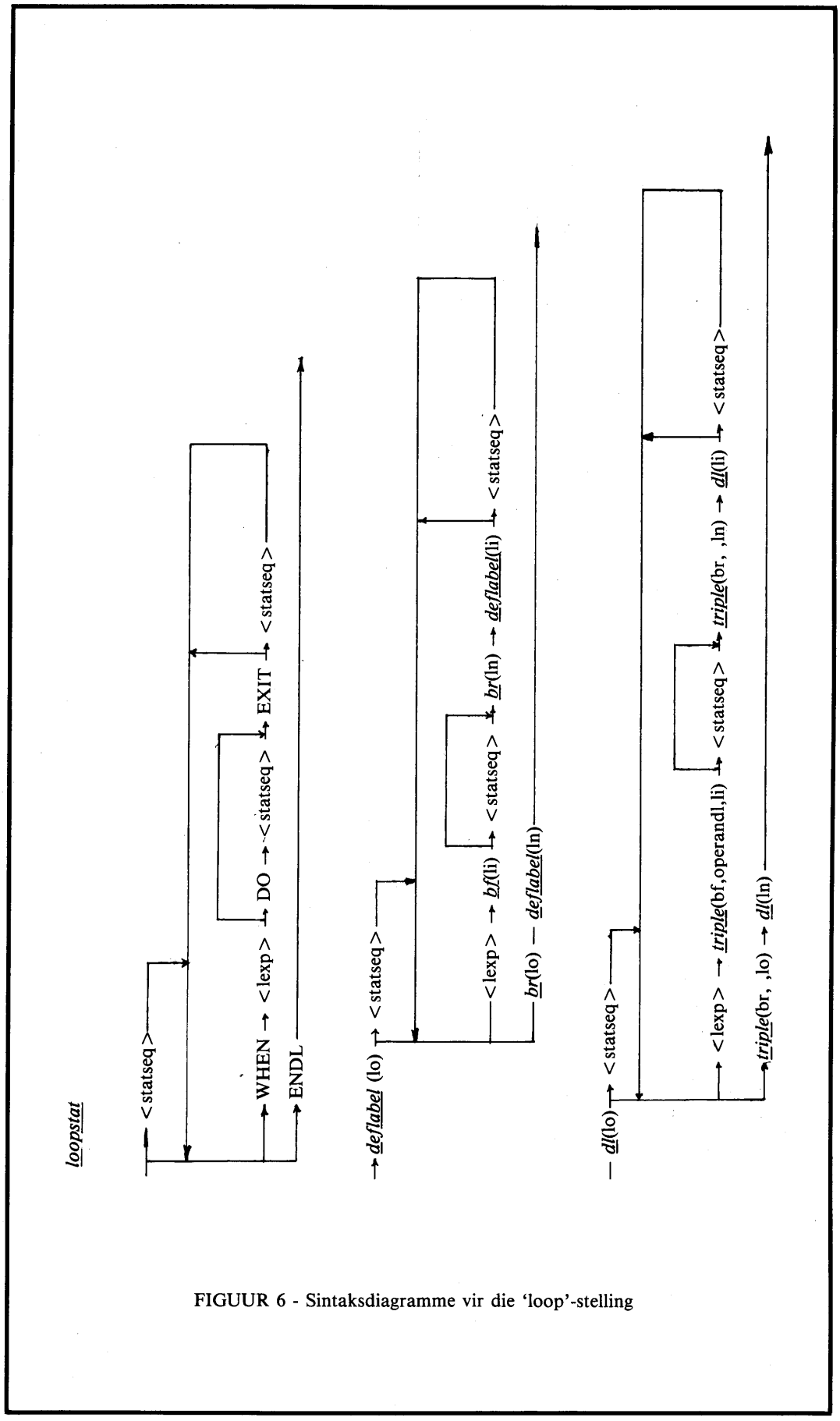


FIGUUR 4 - Die skoenriem vir die eerste SCRAP-vertaler

<module> ::= MODULE <module ident>; {<global block>} [BEGIN <statement sequence>] ENDM <module ident>



FIGUUR 5 - Die produksiereël vir 'n module



FIGUUR 6 - Sintaksdiagramme vir die 'loop'-stelling

Verwysings

[1] VAN ROOYEN, M.H., 'Die Ontwerp en Implementasie van 'n Taal vir Stelselprogrammering', TWISK 190, WNNR, Pretoria, 1981.

[2] MYERS, G.L., 'Reliable Software Through Composite Design', Mason/Charter Publishers, Inc., London, 1975.

[3] ANDERSSEN, E.C. 'Automatisasie van die Administrasie van Programmodules met Spesiale Verwysing na 'n Saamstelvlak-taal', M.Sc.-tesis, RAU, 1979.

[4] LECARME, O. and PAYROLLE-THOMAS, M.C., 'Self-compiling compilers: An Appraisal of their Implementation and Portability', Software — Practice and Experience, Vol. 7, 1978 (149-170).

[5] AHO, A.V. and ULLMAN, J.D., 'The Theory of Parsing, Translation and Compiling, Volume 1', Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1972.

[6] HARTMAN, A.C., 'A Concurrent Pascal Compiler for Minicomputers', Springer-Verlag, 1977.

[7] AHO, A.V. and ULLMAN, J.D., 'Principles of Compiler Design', Addison-Wesley, 1977.

[8] GRIES, D., 'Compiler Construction for Digital Computers', John Wiley & Sons, Inc., 1971.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

*Presented at the second South African Computer Symposium held on 28th and 29th October, 1981.

Printed by Studio Press, Johannesburg