Contents

Preface
P. Machanick ................................................................. 1

Research Articles
Active Learning: Issues and Challenges for Information Systems and Technology
RD Quilling, GJ Erwin and O Petkova ................................................................. 5

A Generic Modelling Framework for Interactive Authoring Support Environments
Paula Kotze ................................................................................................. 15

O Petkova and JD Roode ........................................................................ 26

An Information-Theoretic Semantics for Belief Change
T Meyer ................................................................................................... 33

A Complexity Metrics Model for Software Correction
A Törn, T Anderson and K Enholm ................................................................. 40

A Conceptual Design for High-Volume Data Processing of Warehouse Database into Multidimensional Database
Paisarn Trakulsuk and Vichit Avatchanakorn ............................................ 49

A Pragmatic Approach to Bitemporal Databases: Conceptualization, Representation and Visualisation
Chiyaba Njovu and WA Gray ...................................................................... 58

A Building Recognition System
SP Levitt and B Dwolatzky ....................................................................... 68

Computer Programming and Learning to Write
John Barrow ............................................................................................ 77

Co-operating to Learn using JAD Technologies
TA Thomas ................................................................................................ 87

Critical Success Factors for the Implementation of DSS at a Selection of Organisations in Kwazulu/Natal
URF Averweg and GJ Erwin ................................................................. 95

Enhancing the Predictability of Two Popular Software Reliability Growth Models
Peter A Keiller and Thomas A Muzzuchi .............................................. 105
Generalised Unification of Finite Temporal Logic Formulas
Scott Hazelhurst .................................................. 110

Harmonizing Global Internet Tax: A Collaborative Extranet Model
E Lawrence and B Garner ......................................... 119

Improving Object Oriented Analysis by Explicit Change Analysis
Lui Yu, Siew Chee Kong, Yi Xun and Miao Yuan ........ 128

Reconciling the Needs of New Information Systems Graduates and Their Employers in Small, Developed Countries
Rodney Turner and Glenn Lowry ............................. 136

Shortest Delay Scheduling Algorithm for Lossless Quality Transmission of Stored VBR Video under Limited Bandwidth
Fei Li, Yan Liu, Jack Yiu-Bun Lee and Ishfaq Ahmad .......... 146

Software Croma Keying in an Immersive Virtual Environment
Frans van der Berg and Vali Lalioti .......................... 155

Some Automata-Theoretic Properties of \( \cap \)-NFA
Lynette van Zijl and Andries PJ van der Walt ............... 163

Statistical Analysis of an Automated Computer Architecture Learning Environment
JT Waldron, J Horgan and G Keogh ............................ 168

The CILT Multi-Agent Learning System
Herna L Viktor .................................................... 176

The Development of a Generic Framework for the Implementation of Cheap, Component-Based Virtual Video-Conferencing System
Soteri Panagou and Shaun Bangay ............................ 185

The Role of Experience in User Perceptions of Information Technology: An Empirical Examination
Meliha Handzic and Graham Low ............................... 194

What are Web Sites Used for: Cost Savings, Revenue Generating or Value Creating?
Man-Ying Lee .................................................... 201

New Ideas Papers
Approaches to Video Transmission over GSM Networks
Bing Du and Anthony Maeder .................................. 210

From Information Security Baselines to Information Security Profiles
Rossouw von Solms and Helen van der Haar ................. 215

Grounded Theory Methodology in IS Research: Glaser versus Strauss
J Smit ............................................................. 219

Introducing a Continuum of Abstraction-Led Hierarchical Search Techniques
Robert Zimmer and Robert Holte .............................. 223
Multimedia as a Positive Force to Leverage Web Marketing, with Particular Reference to the Commercial Sector

**Stan Shear** .......................................................... 229

Understanding HCI Methodologies

**Peter Warren** .......................................................... 234

Electronically Published Papers


---

**Experience Papers**

A Java Client/Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet

**Dieter Bühler, Gerd Nusser, Gerhard Gruhler and Wolfgang Küchlin** .......................................................... 239

An Object-Oriented Framework for Rapid Client-side Integration of Information Management Systems

**Ralf-Dieter Schimkat, Wolfgang Küchlin and Rainer Krautter** .......................................................... 244

Distributed Operating Systems: A Study in Applicability

**Jürgen Prange and Judith Bishop** .......................................................... 249

Formal Verification with Natural Language Specifications: Guidelines, Experiments and Lessons so far

**Alexander Holt** .......................................................... 253

Introducing Research Methods to Computer Science Honours Students

**Vashti Galpin, Scott Hazelhurst, Conrad Mueller and Ian Sanders** .......................................................... 258

Visualising Eventuality Structure

**ST Rock** .......................................................... 264

Electronically Published Papers

SAICSIT'99
South African Institute of Computer Scientists and Information Technologists
Annual Research Conference 17-19 November 1999
Prepare for the New Millennium
Is there life after y2k?
Mount Amanzi Lodge, Hartebeespoort, South Africa

Sponsors

[Logos of sponsors]

http://www.cs.wits.ac.za/

PHDS
research/PHDS.html

Think different.
Preface

Philip Machanick, Overall Chair: SAICSIT’99

Running SAICSIT’99, the annual research conference of the South African Institute for Computer Scientists and Information Technologists, has been quite an experience.

SAICSIT represents Computer Science and Information Systems academics and professionals, mainly those with an interest in research. When I took over as SAICSIT president at the end of 1998, the conference had not previously been run as an international event. I decided that South African academics had enough international contacts to put together an international programme committee, and a South African conference would be of interest to the rest of the world.

I felt that we could make this transition at relatively low cost, given that we could advertise via mailing lists, and encourage electronic submission of papers (to reduce costs of redistributing papers for review).

The first prediction turned out to be correct, and we were able to put together a strong programme committee.

As a result, we had an unprecedented flood of papers: 100 submitted from 21 countries. As papers started to come in, it became apparent that we needed more reviewers. It was then that the value of the combination of old-fashioned networking (people who know people) and new-fashioned networking (the Internet) became apparent. While the Internet made it possible to convert SAICSIT into an international event at relatively low cost, the unexpected number of papers made it essential to find many additional reviewers on short notice. Without the speed of e-mail to track people down and to distribute papers for review, the review process would have taken weeks longer, and it would have been much more difficult to track down as many new reviewers in so little time.

Even so, the number of referees who were willing to help on short notice was a pleasant surprise.

The accepted papers cover an interesting range of subjects, from management-interest Information Systems, to theoretical Computer Science, with subjects including database, Java, temporal logic and implications of e-commerce for tax.

In addition, we were very fortunate in being able to invite the president of the ACM, Barbara Simons as a keynote speaker. Consequently, the programme for SAICSIT’99 should be very interesting to a wide range of participants.

We were only able to find place in the proceedings for 36 papers out of the 100 submitted, of which only 24 are full research papers. While this number of papers is in line with our expectation of how many papers would be accepted in each category, we did not have a hard cut-off on the number of papers, but accepted all papers which were good enough, based on the reviews. Final selection was made by myself as Programme Chair, and Derrick Kourie, as editor of the South African Computer Journal. Additional papers are published via the conference web site.

We believe that we have put together a quality programme, and hope you will agree.

Acknowledgments

I would like to thank the South African Computer Journal production team, Andries Engelbrecht and Herna Viktor, respectively from the Department of Computer Science and Informatics, University of Pretoria, for their work on producing the proceedings.

The reviewers listed overleaf did an excellent job: many wrote very detailed reports, sometimes after being called in on very short notice. Inevitably, there were some glitches resulting from the unexpected workload, but the buck stops with the programme chair: I promise to do better next time.

I would also like to thank my own department for putting up with the extra work and expense that running a conference entails. I tried not to burden them with too much extra work, but our secretaries, Zalm Gowar and Leanne Reddy, inevitably had to take on some extra work. John Ostrowick provided valuable assistance with design of our web pages and call for papers poster. Carol Kernick, who handles our finances and membership records, did a fine job of keeping up with the demands of the conference.

Finally, I would like to thank our sponsors, whose contribution made this conference been possible:

- PricewaterhouseCoopers – sponsored generous prizes and the conference banquet
- National Research Foundation (NRF) – provided financial support
- University of the Witwatersrand – provided financial support
- Programme for Highly Dependable Systems, University of the Witwatersrand – provided financial support
- Standard Bank – provided financial support
Editorial

- Apple Computer - provided equipment for the conference
- Qualica - provided technical support including helping with the conference web site

Web Site

For more information about SAICSIT, including a pointer to the conference site, see <http://www.saicsit.org.za>.

Referees

- Department of Computer Science, University of Pretoria
  - Derrick Kourie
  - Bruce Watson
  - Vali Lalioti
  - Andries Engelbrecht
  - Ivan Mphahlele
- German National Research Center for Information Technology - GMD
  - Gernot Goebbels
- School of Math Stat and IT, Natal University Pietermaritzburg
  - Peter Warren
- Graduate School of Business, University of Cape Town
  - Kurt April
- CSIR
  - James Jardine
- Department of Computer Science, Rhodes University
  - Shaun Bangay
  - Peter Clayton
  - John Ebden
  - Richard Foss
  - George Wells
  - Peter Wentworth
- Information Technology Division, Rhodes University
  - Caro Watkins
- Department of Information Systems, Rhodes University
  - Brenda Mallinson
  - Dave Sewry
- Department of Informatics, University of Pretoria
  - Carina de Villiers
  - Herna Viktor
  - Niek du Plooy
  - Elsje van Rooyen
  - Machdel Matthee
  - Alan Abrahams
  - Jackie Phahlamohlaka
- School of Information Technology & Engineering, University of Ottawa
  - Stan Szpakowicz
  - Dwight Makaroff
- Department of Information Systems, Victoria University
  - Glenn Lowry
  - Peter Shackleton
  - Tas Adam
  - Alastair Wallace
  - Stephen Burgess
  - Dave Burgess
  - Julie Fisher
  - Jerzy Lepa
  - Geoff Sandy
  - Rod Turner
  - Alastair Wallace
  - Andrew Wenn
- Peninsula School of Computing and Information Technology, Monash University
  - Ainslie Ellis
- School of Information Management and Systems, Monash University
  - Angela Carbone
- School of Computer Science & Software Engineering, Monash University
  - Dianne Hagan
  - Judy Sheard
- School of Management Information Systems, Deakin University
  - John Lamp
  - David Mackay
  - Philip Joyce
Marie van der Klooster
Bill Hewett
Jennie Carroll
Rodney Carr

Computer and Automation Research Institute, Hungarian Academy of Sciences
Ferenc Vajda

Department of Computer Science, University of the Witwatersrand
Ian Sanders
David Lubinsky
Conrad Mueller
Yinong Chen
Bob Baber
Vashti Galpin
András Salamon
Scott Hazelhurst
Philip Machanick
Zoltán Fazekas

Department of Electrical Engineering, University of the Witwatersrand
Barry Dwolatzky
Farzin Aghdasi

Department of Mathematics, University of the Witwatersrand
Jonathan Burgess

School of Law, University of the Witwatersrand
Victoria Bronstein

Edward Nathan & Friedland Attorneys, Johannesburg
Justine White

Department of Information Systems & Technology, University of Durban-Westville
Geoff Erwin

Department of Computer Science, Clemson University, Clemson, SC
Karen Hay

Logical SA
Philip Green

Department of Computer Science, University of Bristol

Alan Chalmers

Department of Systems and Computer Science, Howard University, Washington, D.C
Todd Shurn
John Trimble

Department of Computer Science, University of Cape Town
Sonia Berman

King's College, London
Peter Wood

Department of Information Systems, University of Cape Town
Paul Licker
Steve Erlank
Alemayehu Molla

Department of Department of Mathematics and Applied Mathematics, University of Cape Town
Ingrid Rewitzky
Peter Jipsen
Renkuan Guo

Department of Information Systems, University of the Western Cape
Andy Bytheway

Department of Electrical Engineering and Computer Science, The George Washington University, Washington, DC
Rachelle Heller

School of Biophysical Sciences and Electrical Engineering, Swinburne University of Technology
Ian Macdonald

Electrical and Electronic Systems Engineering, Queensland University of Technology
Anthony Maeder

Computer Science Department, Åbo Akademi
Aimo Törn
Kaisa Sere
Mats Aspnas

Division of Informatics, University of Edinburgh
Graeme Ritchie
Arturo Espinosa-Romero
Jon Oberlander
Editorial

- Chris Brew
- Alexander Holt

- Department of Computer Science and Engineering, Helsinki University of Technology
  - Tommi Junttila
  - Nisse Husberg

- Technical University of Budapest, Department of Measurement and Information Systems
  - Andras Pataricza

- NASA-Goddard Space Flight Center
  - Nigel Ziyad

- Department of Computer Science & Information Systems, University of Port Elizabeth
  - Janet Wesson
  - Leon Nicholls

- Computer Technology Department, Indiana University Purdue University Indianapolis
  - Tim Price

- Department of Electrical Engineering and Computer Science, The University of Michigan
  - Trevor Mudge

- Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
  - David Forsyth

- School of Computer Science, University of Birmingham
  - Mark Ryan

- Faculty of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam
  - Carlos Areces

- Department of Computer Science, Universidade do Vale do Rio dos Sinos - UNISINOS Rio Grande do Sul
  - Marcelo Walter

- School of Information Technology & Mathematical Sciences, University of Ballarat
  - Binh Pham
A Complexity Metrics Model for Software

A. Törn  T. Andersson  K. Enholm

Åbo Akademi University, SF-20520 ÅBO, Finland, atorn@abo.fi

Abstract

A complexity metrics model for software is introduced. The model is defined as \( l(p)c(p) = e(p) \), where \( p \) is any piece of software, \( l(p) \) its length, \( c(p) \) an average type complexity measure independent of length, and \( e(p) \) their product. The model is then applied to structural complexity (SC) of collections of software units. Rules for computing SC for the basic structured constructs are given. These make SC sensitive to sequencing, nesting and modularization. The tool may be calibrated to user's perceived model of complexity by specifying some parameters. Arguments for the usability of SC are given by deriving theoretical properties and by a few simple case studies. A Scheme function for computing \( (l,c,e) \) for a software unit or a collection of software units is supplied.

Keywords: Computer program, Hierarchical measure, Software metrics, Structural complexity
Computing Review Categories: D.2.8, D.4.8, K.6.3

1 Introduction

Many concepts have been suggested to define internal attributes of software systems, i.e., attributes, like size, that depend just on the software object. It is believed that there is a relationship between internal attributes and external attributes, i.e., attributes that depend on the environment in which the software exists, like maintainability. Leaning on this relationship the goal is to control the internal attributes so that some external attributes are hold at desirable levels.

The most well-known internal attribute is software length, which is generally accepted even if there are many ways to measure it. Another often proposed attribute is complexity. There exist consensus that no single metric can be used to measure the complexity of large software systems [14],[17], [4], [5]. We will here restrict ourselves to internal complexity attributes.

There are some general principles that should be adhered to when introducing a set of attributes. One such principle is that the attributes should be independent, i.e., they should describe different characteristics of software. When in mathematics we write \( f(x,y) \), \( x \) and \( y \) are independent, so it will never be that \( y = g(x) \). In many cases this has not been respected when suggesting internal attributes for software. In particular this is so for the attributes length and complexity and not only for specific measures but also in papers discussing desirable properties of measures of attributes. (See Section 4.) Sometimes the word complexity is even used to cover both length and something different from length at the same time.

The main purpose of this exposition is to present a simple model that relates three important complexity characteristics. It is shown how the model is inspired by general notations such as rate or density. In applying the model to software, a measure called the software's structural complexity (SC) is derived. Computing the software's SC should give the developer an indication of the rate of maintenance required or of the effort that would be needed to make a change to the software.

1.1 Two Often Used Complexity Measures

One of the most used complexity measures for software is its length defined as the number of lines of code. This measure is in accordance with the view of complexity as sensitive to length. A longer program is more complex than a short one.

Another very popular software complexity measure is McCabe's cyclomatic complexity measure. As described by McCabe, the primary purpose of the measure is to "... identify software modules that will be difficult to test or maintain" [13]. This can be interpreted to mean that McCabe's cyclomatic complexity measure measures something else than just length.

McCabe's cyclomatic complexity measure is defined for a program expressed as a flowgraph as:

\[
v = e - n + 2 = 1 + d,
\]

where \( v \) is the cyclomatic complexity of the flowgraph \( f \), \( e \) is the number of edges, \( n \) the number of nodes, and \( d \) is the number of predicate nodes of \( f \).

\( v \) from \( v = 1 + d \) we conclude that \( v \) more or less is a count of the predicate nodes. As a longer program will normally have more predicate nodes than a shorter one, McCabe's \( v \) strongly correlates with length. In Grady [9] it is reported that for a system studied, the number of updates was proportional to the number of decision statements. This finding is of course not surprising, because it means that the number of updates is proportional to program length which is to be expected. Further objections are raised in [14] namely that \( v \) is insensitive to program restructuring, and that \( v \) takes no account of program nesting levels.

40 SACJ / SART, No 24, 1999
In his paper McCabe also proposed the less known essential complexity measure \( v_c = v - m \), for measuring the overall level of structuredness. Here \( m \) is the number of structured (\( D \) structured) subflowgraphs. This measure is actually the cyclomatic complexity number of the largest prime in the decomposition tree of the flowgraph [5]. This measure is thus independent of length.

1.2 Three Views of Complexity

Of course it is not in general desirable to measure characteristics of entities that strongly correlate. Rather, one want to measure independent characteristics. These could then be used to compute other characteristics which then may correlate. This problem with McCabe's cyclomatic complexity measure in relation to program length has been observed by several authors. In Banker et al [2] the high correlation of cyclomatic complexity with lines of code is given as the reason for proposing a transformed metric, 'density of decision making', obtained by dividing the cyclomatic complexity with the total number of statements. Gill and Kemerer [7] use 'complexity density' defined as the ratio of cyclomatic complexity to thousand lines of code. By this they want to make the complexity measure independent of length.

There are thus three complexity components involved here: length, length independent complexity, and a combination of these two. These measures have very different properties and therefore a failure to differentiate between them will lead to confusion and contradictions.

One example of this is when having to answer the question: which is most complex, a big system or a small complex piece of software? The answer is that the result depends on which of these measures is used.

A second example is given in [5] where implications of measurement theory on complexity relations on flowgraphs is discussed. Requiring that such a relation should be at least ordinal (i.e., the flowgraphs can be ordered on a complexity scale) means that the relation must be negatively transitive. They contend that no general notation of complexity can give rise to such transitivity. A contradicting example is given. The example involves comparing flowgraphs representing a single choice (\( y \)), two choices in sequence (\( x \)) and an iteration (\( z \)). They state that \( xC y \) (\( x \) is more complex than \( y \)) but that neither \( xC z \) nor \( zC y \) is obviously valid because programmers may disagree. It seems that complexity is interpreted as something proportional to length for the conclusion \( xC y \) whereas for the negative conclusions that complexity is interpreted differently. The contradiction thus comes from not respecting the different meanings of complexity.

Our third example comes from research in establishing acceptable axioms for a complexity measure. (See Section 4.1.) The failure to realize the existence of different views leads in this case to conflicting axioms.

All three views of complexity are important and contained in the complexity model to be presented. We illustrate our model by applying it to structural complexity of software. In addition to software units we also allow collections of software units and the possibility to have calls in units to other units, which means that the measure is covering also software libraries and modularization, i.e., software systems. The measures in the presented model could be used to rank order software units with respect to expected rate of maintenance or to estimate the cost of a given maintenance task.

2 A Complexity Model for Software

We will here introduce a simple complexity model containing length and attributes in accordance with the two other views of complexity presented in Section 1.

2.1 A Simple Generic Model

In many contexts, a concept can be described by the simple generic linear model

\[
T = Sd, \tag{1}
\]

where \( S \) is some type of size measure, \( d \) is some type of density measure defined per size unit, and their product \( T \), a totality measure connected with \( d \).

For instance if \( d \) is the price of one unit of some goods and \( S \) denotes the quantity bought then \( T \) is the total price of the purchase. If \( d \) is the stretch traveled per time unit and \( S \) is the time spent in traveling then \( T \) is the total stretch traveled. If \( S \) is the volume and \( d \) the density of the matter filling \( S \) then \( T \) is the total mass of the matter filling \( S \).

Both measures \( T \) and \( S \) are additive, i.e., given \( T_i = S_i d_i \), \( i = 1,\ldots,n \) it makes sense to consider

\[
T = T_1 + \ldots + T_n, \quad S = S_1 + \ldots + S_n.
\]

For the density \( d \) we obtain

\[
d = T/S = (S_1d_1 + \ldots + S_n d_n)/(S_1 + \ldots + S_n) \tag{2}
\]

i.e., it can be interpreted as the size weighted average of the densities of its parts. For the examples above for two purchases the total price is obtained by adding the two total prices, the quantities may be added to obtain total quantity and the unit price may be computed from Eq.(2); for the stretch-time-velocity example the same is true and Eq.(2) then gives the average velocity.

Using measurement theory it is easy to conclude that the measures \( S,d \) and \( T \) are of ratio type, i.e., the scale transformations are \( g(x) = ax \), where \( a > 0 \) (see for instance Example 2.29 in [17]).

For a measure of density type it is not enough to just give \( d \), also \( S \) must be given, so what we have is actually the pair \((S,d)\).
2.2 Application to Software

When applying the model to software the entity $S$ corresponds to length. Let us denote the length of a piece of software $p$ as $l(p)$. We denote the component corresponding to $d$ by $c(p)$ and call it average complexity, and the third component corresponding to $T$ by $e(p)$ and call it total complexity. We thus have the SC model

$$e(p) = l(p)c(p).$$

For a collection of software units (systems, libraries) $P = \{p_1, ..., p_n\}$ the measures $l(P)$ and $c(P)$ could be calculated based on the length and complexities of the individual units contained in the collection by using Eq.(2) giving the pair $(l(P),c(P))$, where

$$l(P) = l(p_1,...,p_n) = l(p_1) + ... + l(p_n)$$

and

$$c(P) = c(p_1,...,p_n) = \frac{l(p_1)c(p_1) + ... + l(p_n)c(p_n)}{l(p_1) + ... + l(p_n)},$$

i.e., the average complexity of a collection of software units is the length weighted average of the average complexities of its parts. For $e(P)$ we obtain

$$e(P) = e(p_1,...,p_n) = e(p_1) + ... + e(p_n).$$

3 Structural Complexity (SC) of Software Units

In this section we refine the general model for software defined in the previous section to describe structural complexity of software units. We will define the measures $l$ and $c$ for programming constructs so that the structural complexity of a program can be computed if the length and average complexities of its parts are known. The computation of the pair $(l,c)$ will for each construct be reduced to simple arithmetic operations involving computation of the pair $(l,c)$ for the collection of its parts.

3.1 SC for Sequence

We define the average complexity of a sequence of statements $p_1;...;p_n$ to be a constant $c_r \geq 1$ times the complexity for the collection of the same items, i.e.,

$$l(p_1;...;p_n) = l(p_1,...,p_n)$$

$$c(p_1;...;p_n) = c_r c(p_1,...,p_n)$$

$$e(p_1;...;p_n) = c_r [e(p_1) + ... + e(p_n)],$$

Choosing $c_r > 0$ will make sequences more complex than collections.

3.2 SC for Choice

For the choice construct: if $b$ then $p$ else $q$, we define

$$l(if) = l(if) + l(b,p,q)$$

$$c(if) = c_{if}c(b,p,q)$$

$$e(if) = c_{if} [e(b) + e(p) + e(q)] \quad (l_if = 0),$$

where $c_{if}$ is some constant $> c_r$ which then determines the relative difference in complexity between sequence and an if-statement. For missing else part just leave out terms involving $q$. The constant $l_if$ measures the length of the reserved words in an if-statement, i.e., of "if then else". (See Section LC for Atoms.) The notation $(l_if = 0)$ after the third equation means that the third equation is true for $l_if = 0$.

3.3 SC for Iteration

For the iteration construct: while $b$ do $p$, we define

$$l(while) = l(while) + l(b,p)$$

$$c(while) = c(while) = c_{while}c(b,p)$$

$$e(while) = c_{while} [e(b) + e(p)] \quad (l_while = 0),$$

where $c_{while}$ is some constant $> c_{if}$ which then determines the relative difference in complexity between sequence, an if-statement and a while-statement. The item $l_while$ measures the length of the reserved words of a while-statement i.e., "while do". (See Section SC for Atoms.)

The technique to multiply by a constant $> 1$ to penalize nesting is used in [3]. Their complexity measure family is not of density type but the members are rather length measures, e.g. counts of different types of statements.

3.4 SC for Other Program Constructs

Above we have defined SC for structured programs. Lately produced programs tend to be structured but surprisingly many gotos still occur [11]. In order to allow for unstructured programs we treat the goto construct: goto $L$, in the following way

$$l(goto) = l_{go} + l(L)$$

$$c(goto) = c_{go}(1 + lev),$$

where $c_{go}$ is some constant characterizing the breach of well structuredness in using a goto and $lev$ is the number of levels of nesting jumped. Even if intuitively appealing, this is of course a simplification but easy to implement. In the flowgraph approach it is required, that measures for all primes in the decomposition of the flowgraph must be given, which is not a simple task. Prather [14] suggests a rather complicated way to handle the problem. In order to be able to make explicit calculations he must then arbitrarily decide on an unknown constant.
3.5 SC for Atoms

Our approach uses the structure of programs as expressed in the programming language used. Another approach proceeds by transforming the program into a flowgraph [5], by decomposing this into a unique hierarchy of prime flowgraphs [6], and by defining hierarchical measures on these. This approach does not distinguish between programs using a structured programming construct and one where the equivalent construct is realized with the aid of goto statements.

Using the definitions above the complexity of any program can be computed given the lengths and average complexities of the smallest parts (atoms) of a program namely assignment statements, expressions, procedure calls and goto's. As a first approximation we could let the length and average complexity of simple assignment statements and boolean expressions be 1. Alternatively we could determine the length as the number of tokens or characters and for instance choose the complexity of atoms as a number in some given interval. It should be remembered that also in this case average complexity means complexity per length unit. Starting from the atoms we can then successively compute \((l,c)\) for larger and larger parts of the program until the pair \((l,c)\) for the whole program is reached.

We may assume that \(l_d = L_d = L_p\), let us call this \(l_d\). The constant \(l_d\) must be chosen in accordance with the unit of length.

3.6 Some Comments on the Model

The model contains a number of parameters \(c_s, c_f, c_d, c_w, l_d\) that have to be given explicit values in order for the model to be used. Five unknowns may seem to be rather many but if the model should be able to differentiate between the complexity of different programming constructs this is very much a minimal set. We return to this question in Section 5.

4 Properties of Complexity Measures

Now when we have defined our measures we can derive their theoretical properties and evaluate how they satisfy proposed axioms for complexity measures.

4.1 Proposed Axioms for Complexity Measures

Several authors have proposed axioms or properties that a complexity measure should satisfy or have. For a covering treatment, see [17],[18]. The axioms proposed are intended to describe what properties any "good" complexity metric should have. In most cases the authors have total complexity in mind but in some axiom sets both views are represented resulting in contradictory requirements.

Prather [14] defines a measure called a proper measure of program complexity if it satisfies the three axiom schemes:

Prather (a): \(m(\text{begin } p_1; \ldots; p_n \text{ end}) \geq m(p_1) + \ldots + m(p_n)\)

Prather (b): \(2SS \geq m(\text{if } b \text{ then } p_1 \text{ else } p_2) > SS = m(p_1) + m(p_2)\)

Prather (c): \(2m(p) \geq m(\text{while } b \text{ do } p) > m(p)\)

where the two left-hand inequalities for (b) and (c) are required only for sufficiently large \(m(p)\). The lower bound in (b) could be sharper, something like \(m(p_1) + m(p_2) + m(b)\), assuming that \(m(b)\) is defined, would be more realistic. However, then (c) must be redefined, otherwise iteration could be less complex than selection. Prather's \(m\) is a totality measure because of (a).

Other axiom sets for totality measures are for instance those of [1], [12], and [15]. For these totality measures there is a theorem giving the conditions for the ratio scale [18]. The measure has to be an extensive structure where the totality feature is guaranteed by the axiom of monotonicity.

Most of Bache's axioms are natural for totality measures, interesting are those on nesting requiring \(\mu(F(G)) > \mu(F;G) \text{ and } \mu(H(G)) > \mu(G(H))\), where \(\mu(H) > \mu(G)\), i.e., nesting \(F(G)\) is more complex then sequence, and the complexity of the outer level construct counts most.

Weyukers axioms [16] is an example of an axiom set with axioms in conflict:

Weyuker 2: There are only finitely many programs \(p\) for which \(c(p) = a\), where \(a\) is a constant \(> 0\).

Weyuker 5: Sequencing does not decrease complexity.

Weyuker 6: The programs \(p_1; q\) and \(p_2; q\) may have different complexity even if the programs \(p_1\) and \(p_2\) are equally complex.

Weyuker 7: Permuting the statements of a program may change the complexity.

Weyuker 9: The sum of complexities of two programs is less or equal to the complexity of the concatenated program.

Here property 2 and 5 depends on viewing complexity as a function of length and property 6 as something else. The other properties of Weyuker 1,3,4,8 are less interesting or trivial and will not be discussed here.

In [4] the attribute class size, length, cohesion and coupling are treated. Their size is like our length with the obvious properties. Length is an attribute class that can be used for characterizing for instance depth of a hierarchy. The operation used for reducing length over components is \(\max\), like for McCabe's \(V_e\). This makes this attribute independent of their size. However, for complexity they require the property: "... the complexity of a system \(S\) should be at least as much as the sum of the complexities of any collection of its modules, such that no two modules share relationships...".

All axiom systems have axioms stating that complexity...
ity increases with length of software. This is consistent with the view of complexity interpreted as total complexity. Such an axiom is then in conflict with other axioms exhibiting the other view of complexity and makes complexity to correlate with length.

4.2 Properties of SC

Properties of I are trivial and will not treated here. Also the properties of the total measure e are rather obvious. From the formulas (5), (6) and (7) we can conclude that e basically is a proper complexity measure in the sense of Prather (see the comments in the presentation of proper measure above).

For c we have already concluded that the scale is ratio. The ratio scale implies that there exist programs with zero complexity. These could be programs with empty lines as body. The question whether such programs also have zero length depends on how length is measured.

When length and complexity are separated as in SC the length could really be measured just as number of lines. Giving comment lines zero average structural complexity results in the following property: "When increasing the length by commenting and generous spacing the average structural complexity c of the program will decrease so that the total complexity e remains approximately the same." In this case for a program to have low average structural complexity it also must look well structured.

The measure c is an average complexity measure and therefore is not in accordance with axioms requiring that sequencing does not decrease complexity. The average complexity of a collection of items is always inside the interval of complexities determined by the complexities of the components. This follows easily from the interpretation of c as a weighted average. We thus have

$$\min[c(p), c(q)] \leq c(p, q) \leq \max[c(p), c(q)]. \quad (9)$$

A similar inequality is true for sequence, we only have to multiply the min and max by c_r.

This means that c is not in accordance with the axiom sets mentioned in the previous section, e.g. it does not obey Prathers (a) or have Weyuker's property 5.

Further checking the axiom sets in the previous section, how is it then with Weyuker's property 6? Let \( p_1, p_2, q \) be characterized by \((l_1, c_1), (l_2, c_2), \) and \((l_q, c_q)\) respectively. Then

$$c(p_1; q) - c(p_2; q) = c_r(l_q(l_2 - l_1) - (c_q - c_r))/(l_1 + l_q(l_2 + l_q)), \quad (10)$$

which is different from zero if \( l_1 \neq l_2 \) and \( c_r \neq c_q \), which means that c has Weyuker's property 6. In accordance with an axiom of Bache the measure c obeys commutativity with respect to sequence and collection i.e.

$$c(p; q) = c(q; p). \quad (11)$$

This is consistent if the complexity is viewed as a measure of structure and follows naturally from the interpretation of (4) as a weighted average and is in contradiction with the axiom 7 of Weyuker. Weyuker's axiom 9 cannot be directly applied to the measure c because the sum of complexities is not tied to a program. We could restate axiom 9 as: "The complexity of two components in sequence is always greater than the complexity of the collection of the components.". This is true if \( c_r > 1 \).

Prather's axioms also involve addition and cannot therefore be directly applied but if we make the same interpretation as for Weyuker's axiom 9 it follows that c is a proper measure of program complexity in the sense of Prather.

The effect of using constants \( c_s, c_{f1}, c_{do} > 1 \) is that sequencing and nesting will increase complexity. By letting \( c_s < c_{f1} < c_{do} \) we can make c have the property that iteration is a more complex construct than choice which is more complex than sequence which is more complex than collection. This also implies for nesting that if on while is more complex than while on if, i.e., the complexity of the outer level construct counts most. This is in accordance with Bache's axioms.

One important technique for mastering complexity is modularization. This is not covered by the axiom sets of the previous section but is so by SC. We thus compare the average complexities of the programs \( p_1; p_2 \) and \( \{p_1, (p_{call}; p_2)\} \), i.e., a sequence of two programs and a collection of \( p_1 \) as a module and the sequence of a call to \( p_1 \) (p_call), and \( p_2 \). Let \( l_{call} \) and \( c_{call} \) be the length and average complexity of \( p_{call} \). Then the difference in average complexity, \( c(p_1; p_2) - c(\{p_1, (p_{call}; p_2)\}) \), can after some manipulations be written as

$$\{l_1c_1(c_r - 1) + l_{call}c_r(p_1, p_2) - c_{call}\}/(l_1 + l_2 + l_{call}), \quad (12)$$

showing that modularization brings down average complexity always if the average complexity of the collection \( \{p_1, p_2\} \) is larger than the complexity of the call. Making modules of program pieces deeper down the in the nesting pays even more. Using the Scheme program in the Appendix it is easy find out the exact result. Our complexity model thus supports the intuitive understanding of modularization.

Hatton in a polemic paper [10] concludes based on several empirical studies "that small components tend to have a disproportionately larger number of bugs than bigger components", and that this is in conflict with general beliefs about the benefits of modularization. The finding that small modules may have many bugs is not surprising. The reason for creating a small module is either that its average complexity one, and reuse increases complexity because of multiple requirements. In the first case the number of bugs can be expected to be equal independent of modularization, but making changes can be expected to be easier in a well defined module because of reduced complexity. In the second case making a reusable module will cost more, but reuse will then make modularization profitable.
5 Evaluation of SC

We have defined a complexity model consisting of the pair $(l, c)$ containing both length and length independent complexity in an integrated model. No other complexity model found in the literature has such a two-dimensional structure. By explicitly containing the two different views of complexity some confusion which would otherwise exist is avoided.

The theoretical properties of the model, covering such important aspects of programming as unstructuredness, sequencing, nesting, modularization, and layout are intuitively qualitatively correct.

Because the model variables are of ratio type it is of interest to test if the parameters can be chosen in such a way that quantitative differences between programs in accordance with intuition can be achieved.

In the following we will present our experimental setting including example programs, choice of parameter values $(c_s, c_{if}, c_{do}, l_d)$, and an algorithm for computing length and complexity for any collection of programs. The programs are given in a form preserving the structure of the programs, where boolean expressions, assignment statements and goto statements are replaced by tuples containing length and structural complexity data for these. We use the algorithm to compute length and complexity of the example programs for a set of parameter values and compare the results with our empirical complexity ranking of the programs in order to illustrate the capability of the model.

5.1 Computing SC Measures for Programs

In order to be able to easily experiment in computing the SC measures for collections of structured programs we write these in a standard form using the following notations:

1) collection: $p_1, p_2, \ldots$ $(c \ p_1 \ p_2 \ \ldots)$
2) sequence: $p_1; p_2; \ldots$ $(s \ p_1 \ p_2 \ \ldots)$
3) choice: if $B$ then $p_1$ [else $p_2$] $(i f \ B \ i f \ p_1 \ [e l s e \ p_2])$
4) iteration: while $B$ do $p_1$ $(w h i l e \ B \ d o \ p_1)$

Above $B, p_1, p_2, \ldots$ could be any of the structures 2)-4). This means that the computation of SC for a given program is made recursively until only the atomic components remain. The atomic components of the flowgraph, boolean expressions in the decision nodes, assignment statements in the non-decision nodes, and goto statements are given as

- decision node: $(b \ l_a \ c_a)$
- program node: $(n \ l_a \ c_a)$
- goto node: $(g o \ l_a \ c_a)$

where $l_a$ and $c_a$ are the length and complexity of the atoms respectively. Let us call the representation just described "node representation". As an example the program $s-s$ for sequential search

\[
i := 1; \\
\text{while } (i < n) \text{ and } key \neq r[i].k \text{ do } i := i + 1; \\
\text{if } key = r[i].k \text{ then } \text{found}(r[i]) \text{ else } \text{notfound}(key);
\]

consists of three sequential statements; a simple assignment statement, a while statement, and an if statement giving the following node representation:

\[
(s \ (n \ 1 \ 1) \ (d o \ (b \ 2 \ 1) \ (n \ 1 \ 1)) \\
(if \ (b \ 1 \ 1) \ (n \ 1 \ 1) \ (n \ 1 \ 1))
\]

The lengths and the complexities of the nodes have been assumed to be 1, except for the length of the predicate in the while statement, which we assume to be 2. In Appendix A a program written in Scheme for computing the SC measures is listed. The expression to be submitted to Scheme is of the form: \((SC ': \ldots\ldots\ldots)\). For the choice of parameters $(c_s, c_{if}, c_{do}, l_d) = (1.1, 1.3, 1.5, 0)$, the calculation successively gives: \((s \ (n \ 1 \ 1) \ (n \ 3 \ 1.5) \ (n \ 3 \ 1.3)) \Rightarrow (n \ 7 \ 1.1 \times 9.47) \Rightarrow (n \ 7 \ 1.48), i.e., l = 7 \text{ and } c = 1.48\) for $s-s$.

5.2 Example Programs

We next describe the programs used to illustrate the application of SC. These consists of the basic nestings and of four other simple programs, two of which concern searching (sequential search, binary search) and two of which are different versions of the bubble sort algorithm [8]. Our basic choice of values for the atoms is 1 for both length and complexity. With this choice it is appropriate to let the value of $l_d$ be in [0, 1]. Our programs are:

- if $do$: if $B_1$ then while $B_2$ do $a$ else $b$
- do $if$: while $B_1$ if $B_2$ then $a$ else $b$
- $s-s$: Sequential search: The program is given in Section 5.1.
- $b-s$: Binary search
  \[
  \text{low} := 0; \\
  \text{high} := n; \\
  \text{while } \text{high} - \text{low} > 1 \text{ do begin} \\
  \text{j} := (\text{high} + \text{low}) \div \text{2}; \\
  \text{if } \text{key} \leq \text{r}[\text{j}].k \text{ then high} := \text{j} \text{ else low} := \text{j}; \\
  \text{end}; \\
  \text{if } \text{key} = \text{r}[\text{high}].k \text{ then } \text{found}(\text{r}[\text{high}]) \text{ else } \text{notfound}(\text{key});
  \]
- $bu$: Bubble sort
  \[
  \text{while } \text{up} > \text{lo} \text{ do begin} \text{j} := \text{lo}; \text{i} := \text{lo}; \\
  \text{while } \text{i} \leq \text{up} - 1 \text{ do begin} \\
  \text{if } \text{r}[\text{i}].k > \text{r}[\text{i} + 1].k \text{ then begin} \\
  \text{temp} := \text{r}[\text{i}].k; \text{r}[\text{i}] := \text{r}[\text{i} + 1]; \\
  \text{r}[\text{i} + 1] := \text{temp}; \text{j} := \text{i}; \text{end}; \\
  \text{i} := \text{i} + 1; \\
  \text{end}; \\
  \text{up} := \text{j}; \\
  \text{end};
  \]
- $bum$: Bubble sort where the inner while has been replaced by a call to a program equivalent to it. The program $bum$ thus consists of a collection of these two programs. The length and complexity of the call is assumed to be 1.
We also used the examples of Prather [14]:

P1: A sequence of three while statements. (His Fig.1(a)).
P2: Three nested while statements, where the first and second have an extra statement node in order to make it have the same length as P1. (Fig.1(b)).
P3: A structure with 8 decision nodes and nestings up to the third level but with two goto statements jumping 3 respectively 4 levels (Fig.6).
P4: A structure also with 8 decision nodes and nestings up to the fourth level. (Fig.2).
P4': A structured program similar to P4 but with the gotos replaced with nodes (n 1.5) equivalent to a while with an empty body. (Our revision of p4).

5.3 Dependence of c on Parameter Values

In order to be able make explicit calculations of c for our example programs we must decide on the values of the constants \(c_s, c_{if}, c_{do}, l_d\), and how to evaluate length and complexity for the atoms. It is not a priori clear what values to use because the sensitivity of the model to changes in these values is not obvious. We therefore perform experiments with different sets of parameter values used to calculate the complexities for our set of programs.

Results in computing c for the example programs for some different choices of parameter values are given in Table 1. For \(l_d = 1\) the values are 5-10% larger. Of course it is a matter of taste which choice of parameters values best conforms with some empirical understanding. However, what is important is that by adjusting the parameters the measure can be adjusted to conform with the user’s empirical experience. Based on our experiments we believe that this is the case.

Table 1 shows that c for b-s is lower than that for bu, which may seem surprising. However it should be realized that c just measures the structural complexity of the code not the complexity of the method behind the code. As stated in Section 1 there is no single metric that can be used to measure the entire complexity of software.

The last row presents the values of McCabe’s measure. The equalities of the complexities for b-s and bu stems from the fact that McCabe’s measure does not distinguish between choice and iteration and that it is not sensitive to nesting.

<table>
<thead>
<tr>
<th>(p)</th>
<th>(l(p))</th>
<th>(c_s)</th>
<th>(c_{if})</th>
<th>(c_{do})</th>
<th>(c(p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_d=0)</td>
<td>4</td>
<td>1.0</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>(i_d=1)</td>
<td>4</td>
<td>1.1</td>
<td>1.3</td>
<td>1.5</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 1: A set of parameter values and the corresponding SC values c for the programs ifdo..., bu \((l_d = 0)\).

<table>
<thead>
<tr>
<th>(p)</th>
<th>(P1)</th>
<th>(P2)</th>
<th>(P3)</th>
<th>(P4)</th>
<th>(P4')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l(p))</td>
<td>6</td>
<td>6</td>
<td>39</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>(c(p))</td>
<td>1.7</td>
<td>2.8</td>
<td>2.8</td>
<td>4.5</td>
<td>2.5</td>
</tr>
<tr>
<td>(e(p))</td>
<td>10</td>
<td>17</td>
<td>108</td>
<td>112</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 2: Results (rounded) for P1, P2, P3, P4, P4' [14].

Prather uses \(\mu\) as a totality measure to estimate the total testing effort for a program. The \(\mu\) measure is probably too small for the if statement, see the comment in Section 4.1. Actually, \(e\) could be seen as a refinement of \(\mu\), meaning that instead of multiplying the maximum complexity of terms with their number, the sum of the individual products is used and the result is further multiplied by constants relating the complexities of the different primitive programming constructs. It can therefore be expected that \(\mu\) and \(e\) will give similar results, which can be seen from Table 2. However \(\mu\) and \(e\), because they are total complexity measures, does not reveal the inherent complexity of the programs which is important for estimating the effort needed to make changes. For this purpose \(c\) can be used.

As an example take the two programs P2 and P3. They are very different when using \(\mu\) or \(e\), but \(c\) shows that they are of equal average complexity, i.e., making changes to these programs can be expected to need equal efforts. Comparing P3 and P4 we see that their total complexities are more or less the same but \(c\) reveals that making a change to the unstructured P4 can be expected to be considerably more costly.

6 Summary and Conclusions

The proposed complexity model connects length (l), average complexity (c), and total complexity (e) in a natu-
ral way requiring \((l, c, e)\) to be noted for a piece of software. By considering the triple some contradictions coming from involuntarily switching between the two complexity concepts, total complexity and average complexity, can be avoided. This means that all three measures, related through the equation \(e = le\), are important characteristics of software.

We have applied the model to the concept structural complexity of software (SC). The SC model contains five complexity parameters. We have calculated the complexities of some well known algorithms for searching and sorting for different choices of values of the complexity parameters. We found that it is possible to adjust the parameters so that the average complexities of the programs could be made to quantitatively conform with our expectations. We believe that this result is extendable to larger units of software (eg. collections of programs) because of the theoretical properties of the SC model.

In the calculations we assumed that the lengths and average complexities of the atomic nodes all but one were 1, with one exception. Here it is possible to be more sophisticated. For instance, one could take into account the use of data structures in the statements and expressions by adjusting the complexities accordingly. Also the length of the atomic nodes could be calculated using a finer grain.

We also calculated SC for some example programs presented by Prather [14], finding agreement between our total complexity measure \(e\) and his \(\mu\) measure, and motivation for using our average complexity measure \(c\).

We have shown that the properties of the SC measures are in accordance with our qualitative understanding of structural complexity for sequencing, nesting, modularization and unstructuredness. Few of the measures now used have these properties. Despite the limited empirical validation we think that the approach deserves further consideration.

References

A Scheme Program for Computing SC from Flowgraphs

(define (SC list)) ;*** Function SC for computing
(define (length-help lis)) ;*** program length 1
  (cond ((null? lis) 0)
    ((atom? lis) (length-help (eval lis)))
    ((eq? (car lis) 'n) (eval (cadr lis)))
    ((eq? (car lis) 'b) (eval (cadr lis)))
    ((eq? (car lis) 'go) (+ 1-d (eval (cadr lis))))
    ((eq? (car lis) 'c)
      (cond ((null? (cdr lis)) 0)
        (else (+ (length-help (cadr lis))
                  (length-help (cons 'c (cddr lis))))))
    (else (+ 1-d (length-help (cons 'c (cddr lis))))))
(define (complexity-help lis)) ;*** and average structural
  (cond ((null? lis) 0) ;*** complexity c
    ((atom? lis) (complexity-help (eval lis)))
    ((eq? (car lis) 'n) (eval (caddr lis)))
    ((eq? (car lis) 'b) (eval (caddr lis)))
    ((eq? (car lis) 'go) (* c-go (eval (caddr lis))))
    ((and (eq? (car lis) 's) (null? (cddr lis)))
      (complexity-help (cons 'c (cddr lis))))
    ((eq? (car lis) 'c)
      (cond ((null? (cdr lis)) 0)
        (else (/ (+ (* (length-help (cadr lis))
                    (complexity-help (cadr lis)))
                  (* (length-help (cons 'c (cddr lis)))
                    (complexity-help (cons 'c (cddr lis))))))
        (+ (length-help (cadr lis))
            (length-help (cons 'c (cddr lis)))))))
    ((eq? (car lis) 's) (* c-s
      (complexity-help (cons 'c (cddr lis))))
    ((eq? (car lis) 'if) (* c-if
      (complexity-help (cons 'c (cddr lis))))
    ((eq? (car lis) 'do) (* c-do
      (complexity-help (cons 'c (cddr lis)))))
    (let ((length (length-help list)));*** by main starting here
      (complexity (complexity-help list))) (newline)
    (display 'l = "") (display length) (newline)
    (display 'c = "")
    (display (round (* 10 complexity)) 10)) (newline)
    (display *e = "")
    (display (round (* 10 (* length complexity))) 10)) (newline))

48 SACJ / SART, No 24, 1999
Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1½ or double spacing.
- The first page should include:
  - the title (as brief as possible)
  - the author’s initials and surname
  - the author’s affiliation and address
  - an abstract of less than 200 words
  - an appropriate keyword list
  - a list of relevant Computing Review Categories
  - Tables and figures should be numbered and titled.
  - References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj, which gives a number of examples.

SACJ is produced using the \LaTeX\ document preparation system, in particular \LaTeX\ 2e. Previous versions were produced using a style file for a much older version of \LaTeX, which is no longer supported.

Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.