An Effort-Length-Complexity Metrics Model for Software

A. Törn
T. Andersson
K. Enholm

Turku Centre for Computer Science
TUCS Technical Report No 99
March 1997
ISBN 951-650-972-X
ISSN 1239-1891

Abstract

An Effort-length-complexity metrics model for software (Elc) based on measurement theory is introduced. The model is defined by $e(p) = l(p)c(p)$, where p is any piece of software, $e(p)$ the effort in developing it, $l(p)$ its length, and $c(p)$ its structural complexity. Rules for computing Elc for the basic structured constructs are given. These make Elc sensitive to sequencing, nesting and modularization. A Scheme function is supplied for computing e, l and c for any program or collection of programs given in form of flowgraphs. The tool may be callibrated to user's model of complexity by specifying some parameters. Arguments for the usability of Elc are supplied by deriving theoretical properties and by case studies.

Keywords: Effort, length, stuctural complexity, hierarchical measure, software metrics, flowgraphs

1 Introduction

One of the first software quality measures proposed 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" [McCabe 1976].

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

$$\nu(f) = e - n + 2,$$

where $\nu(f)$ is the complexity of the flowgraph f, e is the number of edges and n the number of nodes of f. Concatenating two identical flowgraphs by making the end node of the first to be the start node of the second we obtain

$$\nu(f;f) = 2(e-n)+3 = 2\nu(f)-1,$$

which means that McCabe's measure normally is dependent on length. The only exception is when $\nu(f) = 1$ in which case $\nu(f;f) = 1$. Then f is a number of sequential nodes. Only decision nodes count and McCabe's measure is therefore more or less a count of the decision nodes. As normally a longer program will have more decision nodes McCabe's measure thus strongly correlates with length. In Grady [Grady 1994] it is reported that for a system, 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.

Of course it is not in general desirable to define a new measure that strongly correlates with an old one. This problem with McCabe's measure has been observed by several authors. In Banker *et al* [Banker *et al* 1989] the high correlation of cyclomatic complexity with lines of code is given as 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 [Gill and Kemerer 1991] 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 is thus two views of complexity. One view is related to size by recognizing a big system as complex because of the existence of a large number of components. Another view is present when in comparing two pieces of software of roughly equal length one finds a difference in their complexities. The contradiction is very clear when having to decide which of a big complex system or a small complex piece of software is the most complex. In [Fenton and Pfleeger 1996] implications of measurement theory on complexity relations on flowgraphs is discussed based on the requirement that such a relation in order to be at least ordinal (i.e., the flowcharts can be ordered on a complexity scale) 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 xCy (x *is more complex then* y) but that neither xCz nor zCy is valid, contradicting transitivity. It seems that complexity is interpreted as something proportional to length for the conclusion xCy whereas for the negative conclusions that complexity is interpreted differently. The contradiction thus comes from the two different views of the meaning of complexity.

We take this reasoning a bit further. The purpose of our effort is similar to that of McCabe namely to "identify software modules that will be costly to develop, test or maintain". For this we construct a simple complexity model assuming that in addition to length, (length independent) structural complexity is the most important characteristic of software for such identification. This is confirmed by an empirical investigation (6 877 K statements) finding that control structure errors was the most common type of errors amounting to 25 % of all [Beizer 1990].

The model is hierarchically defined on flowgraphs [Fenton and Pfleeger 1996]. In addition to connected structured graphs we also allow collections of such graphs and the possibility to have calls in graphs to subgraphs, which means that the measure is covering also software libraries and modularization.

We motivate the approach by discussing the theoretical properties of the model and by demonstrating the sensitivity of the model on some complexity parameters by applying the model to some simple algorithms. The purpose of our paper is not to present a final model but a first order approximation showing the potential usefulness of the Elc approach.

Our measure is based on measurement theory. We therefore next briefly introduce measurement theory [Fenton and Pfleeger 1996].

2 Measurement of Attributes of Entities

When we are using attributes of some entities it may be the case that for each entity we can say if the entity has the characteristic or not (e.g. is complex). More often it may be the case that the entity has the characteristic to some degree. This means that given two entities we may conclude that one of the entities has more of the attribute than the other entity (e.g. a is more complex than b). Such an empirical relation system is the base for measurement of the attribute and requires that we have a clear conception of the class of entities and the attribute in question.

A measurement means mapping the entities on some (numerical) scale with respect to the characteristic of interest. The mapping produces a numerical relation system. The mapping is useful if the relations of the empirical relation system are preserved in the numerical relation system. This is called the *representation condition*. If this is valid we say that the mapping is a measure for the attribute. The measure is normally not the only one possible but defined up to some transformations that will preserve the representation condition (*admissible transformations*). The type of this transformation defines the scale of the measure. The possible scales and their admissible transformations are: the scale may only classify the entities (nominal scale, any one-to-one mapping), permit comparisons (ordinal scale, strictly increasing function), permit assigning a relative measure of the attribute of an entity not including entities with zero measure (interval scale, linear transformation $\alpha x + \beta$, $\alpha > 0$), and including entities with zero measure (ratio scale, αx, $\alpha > 0$). When the measurement is performed by counting giving an integer as measure the measure is the only possible (we talk about an absolute scale, the only transformation possible is the identity transformation).

A statement containing numerical measures is *meaningful* if its truth (or falsity) remains unchanged under scale tranformations. This means that the scale chosen will dictate for instance what statistical central measures like the mean, the median or the mode that can be used for the measured attribute of the entity of interest.

As stated above the class of entities of interest and the attribute to measure must of course be clearly understood. This is very important for the representation condition to be meaningful. For the entity being a process complexity trivially means the complexity of some activity and the resources needed to perform that activity. However, this is true even for the entity being a product. For instance something is complex because it is difficult to *understand* or to *modify*.

In the treatment of complexity in the literature presented in the preceeding section these measurement theoretic implications have not always been adhered to. However, if we want to use measurement as the powerfull tool it is in many other areas we must not ignore theory.

3 The Elc Complexity Model

We will here introduce a simple complexity model containing length and entities in accordance with the two views of complexity presented in Section 1. We want Elc to be sensitive to nesting and modularization.

3.1 Elc for Collections of Software

Because we want the structural complexity $c(p)$ of a piece of software p to be independent of its length $l(p)$, it has to be defined per length unit, i.e., it can be interpreted as average complexity per length unit. Multiplying c and l we obtain an item $e(p)$ representing the other view of complexity dependent on length and also on structural complexity, we call it software size,

$$e(p) = l(p)c(p), \tag{1}$$

giving

$$c(p) = e(p)/l(p). \tag{2}$$

By a collection of software we mean a set of pieces of software. For a collection of software $\{p, q\}$ we then obtain

$$c(\{p, q\}) = e(\{p, q\})/l(\{p, q\})). \tag{3}$$

Assuming that sizes are additive we obtain

$$e(\{p, q\}) = e(p) + e(q) = l(p)c(p) + l(q)c(q). \tag{4}$$

But lengths are additive and therefore we obtain

$$c(\{p, q\}) = \frac{l(p)c(p) + l(q)c(q)}{l(p) + l(q)}, \tag{5}$$

i.e., the complexity of a collection of programs is the length weighted avearage of the complexities of its parts. How could then e be interpreted? It is an entity that better than length expresses the size of the software. The size of software could be measured by the effort or resources (cost) spent in developing the software [Törn 1990]. Costs are additive so this is in accordance with the asumption above about the additivity of software sizes. Comparing with Boehm's COCOMO model [Boehm 1981] where the effort E is estimated as $E = a(KLOC)^b$ and b is near to 1 we could interprete a as an approximation of the average effort per KLOC so that also here the efforts are additive.

Eq. (1) states that the resources needed to develop software is proportional to the product of software length and software complexity which are independent measures. Software complexity thus accounts for everything else except length and could be measured as development cost/length unit.

The complexity for collections (systems, libraries) could be calculated based on the length and complexities of the individual programs contained in the collection by using formula (5).

3.2 Elc for the Structured Program Constructs

In this section we will extend the approach to structured programming constructs so that the complexity of a program can be computed if the length and complexities of its parts are known. We only define Elc for structured programs. This is a limitation but today one would expect programs to be structured and therefore this is not a serious limitation. Furthermore the complexity of unstructured constructs have to be decided by the user and can be given directly as complexity for atoms, se below.

3.2.1 Elc for Sequence

We define the complexity of a sequence to be a constant $c_s \geq 1$ times the complexity for a collection, i.e.,

$$l(p; q) = l(p) + l(q)$$

and

$$c(p; q) = c_s c(\{p, q\}). \tag{6}$$

The constant c_s will make the model sensitive to sequencing.

3.2.2 Elc for Choice

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

$$l(if) = l(b) + l(p) + l(q) + l_{if}$$

and

$$c(if) = c_{if}c(b, p, q), \tag{7}$$

where c_{if} is some constant $> c_s$ which then determines the relative difference in complexity between sequence and an if-statement. For missing **else** part leave out ", q". The constant l_{if} measures the length of the reserved words in an if-statement, i.e., of "if then else", see Elc for Atoms.

3.2.3 Elc for Iteration

For the iteration construct: **while** b **do** p we define

$$l(while) = l(b) + l(p) + l_{do}$$

and

$$c(while) = c_{do}c(b, p), \tag{8}$$

where c_{do} is some constant $> c_{if}$ which then determines the relative difference in complexity between sequence, an if-statement and a while-statement. Of course letting c_{if} and c_{do} be constants is only a first approximation. There could be a range for them and the user could then make an explicite choice based on his evaluation of the complexity of the actual construct. The item l_{do} measures the length of the reserved words of a while-statement ,i.e., "while do", see below.

3.2.4 Elc for Atoms

Using the definitions above the complexity of any structured program can be computed given the complexities and lengths of the smallest parts (atoms) of a program namely assignment statements and expressions. As a first approximation we could let the length and 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 classify the complexity according to *low*, *medium* and *high* with some ranges for each level. Also here it should be remembered that complexity means complexity per length unit. We may assume that $l_{if} = l_{do}$, let us call this l_d. The constant l_d must be chosen in accordance with the unit of length.

3.3 Properties of Complexity Measures

Is then such a model of complexity consistent with our empirical understanding of complexity? Before answering this question we look at some attempts to define desirable properties of complexity measures.

3.3.1 Proposed Axioms for Complexity Measures

Several authors have proposed axioms or properties that a complexity measure should satisfy or have. For a covering treatement, see [Zuse 1991].

First we will present some of Weyukers's axioms [Weyuker 1988]. Property 5 is that concatenation does not decrease complexity. Property 6 is that when concatenating program p_1 and p_2 of equal complexity with the same program q the resulting programs $p_1; q$ and $p_2; q$ may have different complexity. Here property 5 depends on viewing complexity as a function of length and property 6 as something else. By further examining Weyuker's axioms property 7 says that permuting the statements of a program may change the complexity. Property 9 states that the sum of complexities of two programs is less or equal to the complexity of the concatenated program. The other properties of Weyuker $1, 3, 4, 8$ are less interesting or trivial and will not be discussed here.

Included in Bache's axioms of complexity [Bache 1987] is an axiom stating that the complexity of a concatenated program is always larger than the complexity of either part.

Among the required properties proposed by Jayaprakash *et al* [Jayaprakash *et al* 1987] there is the *Additive Property* stated as: "Let $s1$ and $s2$ be two one-entry, one-exit arbitrary pieces of program code with control flow complexities $m1$ and $m2$. Now if a large program is constituted by placing $s1$ and $s2$ in sequence, the resulting program should ideally have a control flow complexity of $m1 + m2$, because sequence structuring does not involve any additional complexity. ...".

In [Tian and Zelokowitch 1995] Tian and Zelkowitz include the following axiom among their set of basic axioms (size is basically measured as number of nodes): *An acceptable measure must assign sufficiently large values to sufficiently large programs.*

All four axiom systems have axioms stating that complexity increases with length of software. This is consistent with the view of complexity interpreted as effort to develop the software. Such an axiom is then in conflict with other axioms exhibiting another view of complexity.

3.3.2 Axioms for Elc

We first address the problem of determinimg the scale. The scale of length and size are ratio so the rescalings for l and e are

$$l' = \alpha l \text{ and } e' = \beta e, \text{ where } \alpha, \beta > 0.$$

This gives

$$c' = \beta e/\alpha l = \gamma c, \text{ where } \gamma = \beta/\alpha > 0.$$

This means that the scale for c is ratio. The ratio scale implies that there exist programs with zero complexity. These could be programs with empty lines as body. The question wether such programs also have zero length depends on how length is measured. When length and complexity are separated as in Elc the length could really be measured just as number of lines. Comment lines could be determined to have zero complexity. By increasing the length and readability by commenting and generous spacing the complexity of the program would decrease so that the size would remain almost the same. Also, c is independent of the choice of unit of the length measure. This seems to be well in line of what we expect from a complexity measure.

For Elc the 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 wighted average. We thus have

$$\min[c(p), c(q)] \leq c(p, q) \leq \max[c(p), c(q)]. \tag{9}$$

A similar inequality is true for sequence, we only have to multiply the min and max by c_s. This means that Elc does not have Weyuker's property 5, "concatenation does not decrease complexity" and this is only natural for Elc. How is it then with property 6? We have for $c(p_1) = c(p_2) = c_0$ that

$$c(p_i, q) = [l(p_i)c_0 + a]/[l(p_i) + b], \quad i = 1, 2$$

where a and b are some constants. This gives

$$c(p_1) = c(p_2) \wedge l(p_1) \neq l(p_2) \Rightarrow c(p_1, q) \neq c(p_2, q), \tag{10}$$

which means that Elc has Weyuker's property 6. Elc obeys commutativity with respect to collection and sequence i.e.

$$c(p; q) = c(q; p). \tag{11}$$

This follows naturally from the interpretation of (5) as a weighted average and is in contradiction with the axiom 7 of Weyuker. This is consistent if the complexity is viewed as a syntactical measure. Weyukers axiom 9 cannot be directly applied because the sum of complexities is not tied to a program but we could restate axiom 9 as: "The complexity of two concatenated components is always greater than the complexity of the collection of the same components.". This is true if $c_s > 1$.

The effect of using constants $c_s, c_{if}, c_{do} > 1$ is that concatenation and nesting will increase complexity. By letting $c_s < c_{if} < c_{do}$ we can make Elc correspond to the intuition that iteration is a more complex construct than choice which is more complex than sequence which is more complex than collection.

One important technique for mastering complexity is modularization. We may therefore want to compare the complexities of the programs $p_1; p_2$ and $\{p_1, (p_{call}; p_2)\}$, i.e., a concatenated program and a collection of p_1 as a module and the concatenation of a call to p_1 and p_2. By assuming the length and complexity of a call to be small enough compared to the length and complexities of the program components p_1, p_2 the difference

$$c(p_1; p_2) - c(\{p_1, (p_{call}; p_2)\}) \approx l_1 c_1 (c_s - 1)/(l_1 + l_2) > 0,$$

so that modularization brings down the complexity and the effect is proportinal to the complexity of the modularized part. This result regarding modularization is according to our intuition.

In the introduction we gave a counter example for a complexity relation on flowgraphs based on the requirement that such a relation in order to be at least ordinal must be negative transitive. Using the approach above we conclude that the compexities always can be computed and that therefore no contradiction exists.

4 Applying Elc

In this section 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 a program given in the form of a flowgraph [Fenton and Pfleeger 1996]. We use the algorithm to compute complexities for the example programs and compare the results with our empirical complexity ranking of the programs.

4.1 Experimental Setting

In order to be able make explicite calculations 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 *apriori* 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 a set of programs. These consists of the basic structured constructs and of four other real programs two of which concern searching and two of which are different versions of a sorting algorithm [Gonnet and Baeza-Yates 1991]. Our basic choice of values for the atoms is 1 for both length and complexity. In some cases atom complexity is set to 2 for those atoms regarded more complex than the others. We next describe the programs used to illustrate the application of Elc. With this choice it is appropriate to let the value of l_d be zero.

s-s: Simple sequense: $a; b$

s-if: Simple choice: **if** b **then** a

s-do: Simple iteration: **while** b **do** a

In all these we assume that a, b have length and complexity 1.

seq-sh: Sequential search:

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

Here the lengths and complexities for atoms are set to 1 except the complexities of the boolean expressions which are set to 2 and the length for the first boolean which is set to 2.

bin-sh: Binary search

$low := 0;$
$high := n;$
while $high - low > 1$ **do begin**
    $j := (high + low)$ **div** $2;$
    if $key \ \leq r[j].k$ **then** $high := j$
        else $low := j$
    end;
if $key = r[high].k$ **then** $found(r[high])$
        else $notfound(key);$

Here all lengths and complexities of atoms are set to 1 except the complexities for the assignment statement after **while** and the two following booleans which are set to 2.

bubble: Bubble sort

while $up > lo$ **do begin**
    $j := lo$
    $i := lo$
    while $i \leq up - 1$ **do begin**
        if $r[i].k > r[i+1].k$ **then begin**
            $tempr := r[i];$
            $r[i] := r[i+1];$
            $r[i+1] := tempr;$
            $j = i;$
            end
        $i := i + 1;$
        end;
    $up := j$
    end

Here all lengths and complexities of atoms are set to 1 except the complexity of the boolean of the **if** statement which is set to 2.

bub-mod: Bubble sort where the inner **while** has been replaced by a call to a program equivalent to the inner **while**. The program *bub-mod* thus consists of a collection of these two programs. The length of the *call* is set to 1 and its complexity to 2.

4.2 Computing Elc Measures for Programs

In order to be able to compute the Elc measures for collections of structured programs we introduce the following notations:

1) collection: $p_1, p_2, ...$ $(c\ p_1\ p_2\ ...)$
2) sequence : $p_1; p_2, ...$ $(s\ p_1\ p_2\ ...)$
3) choice: **if** B **then** p_1 [**else** p_2] $(\mathit{if}\ B\ p_1\ [p_2])$
4) iteration: **while** B **do** p_1 $(\mathit{do}\ B\ p_1)$

Above $B, p_1, p_2, ...$ could be any of the structures 2)-4). This means that the computation of Elc 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 and assignment statements in the non-decision nodes are given as

decision node $(b\ l_a\ c_a)$
program node $(n\ l_a\ c_a)$

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 *seq-sh* above has the following node representation:

```
(s (n 1 1)
   (do (b 2 2)
      (n 1 1)
   (if (b 1 2)
      (n 1 1)
      (n 1 1))))
```

In Appendix A a program written in Scheme for computing the Elc measures is listed. The expression to be submitted to Scheme is of the form $(\mathit{Elc}\ '(\mathit{node\ representation}))$.

p			*s-s*	*s-if*	*s-do*	*seq-sh*	*bin-sh*	*bub-mod*	*bubble*
$l(p)$			2	2	2	7	10	12	11
c_s	c_{if}	c_{do}	$c(p)$						
1.0	1.2	1.4	1.0	1.2	1.4	1.8	1.8	1.8	2.1
1.1	1.2	1.4	1.1	1.2	1.4	2.0	2.1	2.0	2.6
1.1	1.3	1.5	1.1	1.3	1.5	2.0	2.2	2.2	3.1
1.1	1.5	2.0	1.1	1.5	2.0	2.7	3.0	3.2	5.8
McCabe's ν			1	2	2	3	4	–	4

Table 1: A set of parameter values and the corresponding complexity values for the programs *s-s*,..., *bubble*

Results in computing Elc for the example programs for some different choices of parameter values are given in Table 1. 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 callibrated to conform with the user's empirical experience. The last row presents the values of McCabe's measure. The equalities of the complexities for *bin-sh* and *bubble* stems from the fact that McCabe's measure does not distinguish between choice and iteration and that it is not sensitive to nesting. We conclude that contrary to Elc, McCabe's measure is too simple to be used to model empirical understanding of complexity.

5 Summary and Conclusions

The proposed effort-length-complexity model (Elc) connects length and the two complexity components effort and (lenght-independent) structural complexity. This connection leads to cleaner concepts and resolves some contradictions which must otherwise exist. We also argue that the new concept of complexity c, better conforms with our empirical understandig of the word complexity. We have shown that the properties of the c are in accordance with empirical understanding of software complexity. We have also calculated the complexities of some well known algorithms for searching and sorting for different choices of values for some complexity parameters. It was found that it is possible to adjust the parameters so that the complexities of the programs conform with our empirical expectations. The result of this evaluation is directly extendable to larger programs because the measure is length independent and computable given the lengths and complexities of the atoms, i.e., assignment statements and expressions.

The introduction of complexities for the different structured constructs have been made in the simplest way allowing the complexity concept to have some properties important for the acceptance of the measure. Here more sophisticated approaches are possible. We have only touched upon the choice of length and complexity of the atomic components. Of course the result of applying Elc is highly dependent of which principles are governing this choice.

The model which is directly applicable to software in the form of final code also could be applied to designs as long as these can be written as flowgraphs. One problem here is to estimate the lengths and complexities of the atoms because these are less atomic then when code is considered.

Elc could also be used to calculate the complexity measure c as an indirect measure by using measures for l and e (Eq. (2)) giving the unit for complexity as *effort/line*. The measure could be used to identify software with high complexity. High complexity could be caused either by high complexity of the problem solved or that a sophisticated solution was implemented. If neither of these explanations apply then a high complexity measure could indicate low quality of software.

In general we conclude that the proposed model of complexity (Elc), including software length, effort, and length-idependent structural software complexity, in the simple form implemented here is promising enough for practical evaluation and for further theoretical development.

References

[Banker *et al* 1989] D.D. Banker, S.M. Datar and D. Zweig, Software complexity and maintainability, In: *Proceedings of the Tenth International Conference on Information Systems*, Dec. 4-6, Boston, 1989, 247-255.

[Bache 1987] R. Bache, Metricating A-KINDRA BT Project 610287, Document W021, South Bank.

[Beizer 1990] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold.

[Boehm 1981] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, New Jersey.

[Fenton and Pfleeger 1996] N.E. Fenton and S.L. Pfleeger, *Software Metrics – A Rigorous & Practical Approach*, International Thomson Computer Press, London.

[Gill and Kemerer 1991] G.K. Gill and C.F. Kemerer, Cyclomatic Complexity Density and Software Maintenance, *IEEE Trans. Software Engineering*, Vol. SE-17, 1284-1288.

[Gonnet and Baeza-Yates 1991] G.H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley, New York.

[Grady 1994] R.B. Grady, Successfully applying software metrics, *Computer*, Vol. 27, No. 9.

[Jayaprakash *et al* 1987] , S. Jayaprakash, K.B. Lakshamanan, P.K. Sinha, MEBOW: A Comprehensive Measure of Control Flow Complexity, COMPSAC 87, 238-244.

[McCabe 1976] T. McCabe, A complexity measure, *IEEE Transactions of Software Engineering*, Vol. SE-1, 312-327.

[Tian and Zelokowitch 1995] J. Tian and M.V. Zelkowitz, Complexity Measure Evaluation and Selection, *IEEE Transactions on Software Engineering* Vol. 21, No. 8, 641-650.

[Törn 1990] A.A. Törn, Models of software accumulation, *J. of Systems & Software* Vol. 12, 39-42.

[Zuse 1991] H. Zuse, *Software Complexity*, de Gruiter, Berlin.

APPENDIX A Scheme program for computing Elc from flowgraphs

```
(define Elc (lambda (list)   ; Function Elc for computing ;
  (define length-help        ; the length and complexity  ;
   (lambda (lis)             ; of programs represented as ;
    (cond ((null? lis) 0)    ; flowgraphs                 ;
      ((atom? lis) (length-help (eval lis)))
      ((eq? (car lis) 'n) (eval (cadr lis)))
      ((eq? (car lis) 'b) (eval (cadr lis)))
      ((eq? (car lis) 'c)
         (cond ((null? (cdr lis)) 0)
         (else (+ (length-help (cadr lis))
                  (length-help (cons 'c (cddr lis)))))))
     (else (+ l-d (length-help (cons 'c (cdr lis))))))))
  (define complexity-help
   (lambda (lis)
    (cond ((null? lis) 0)
      ((atom? lis) (complexity-help (eval lis)))
      ((eq? (car lis) 'n) (eval (caddr lis)))
      ((eq? (car lis) 'b) (eval (caddr lis)))
      ((and (eq? (car lis) 's) (null? (cddr lis)))
               (complexity-help (cons 'c (cdr 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 (cdr lis)))))
       ((eq? (car lis) 'if)  (* c-if
                        (complexity-help (cons 'c (cdr lis)))))
       ((eq? (car lis) 'do)  (* c-do
                        (complexity-help (cons 'c (cdr lis))))))))
   (let ((length (length-help list))
```

```
        (complexity (complexity-help list)))
    (display "Length     = ") (display length) (newline)
    (display "Complexity = ")
         (display (/ (round (* 10 complexity)) 10)) (newline)
    (display "Effort     = ")
         (display (/ (round (* 10 length complexity)) 10)))))

; Parameter values
(define  c-s  1.1)
(define  c-if 1.3)
(define  c-do 1.5)
(define  l-d  0.0)

; Application of Elc to sequential search
(Elc seq-sh)
Length     = 7
Complexity = 2.0
Effort     = 14.0
```

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**