CYCLOMATIC COMPLEXITY: THEME AND VARIATIONS

Brian Henderson-Sellers,
Yagna Raj Pant,
June M. Verner
School of Information Systems,
University of New South Wales
Kensington, Australia

ABSTRACT

Focussing on the ``McCabe family" of measures for the decision/logic structure of a program, leads to an evaluation of extensions to modularization, nesting and, potentially, to object-oriented program structures. A comparison of rated, operating and essential complexities of programs suggests two new metrics: `inessential complexity" as a measure of unstructuredness and ``product complexity" as a potential objective measure of structural complexity. Finally, nesting and abstraction levels are considered, especially as to how metrics from the ``McCabe family" might be applied in an object-oriented systems development environment.

INTRODUCTION

Though software complexity research began around the 1970s, a significant number of problems related to mastering complexity and significantly boosting programming productivity remain to be solved. Indeed, failure to master complexity is the cause of many of present problems in software development. These problems are manifested by a high demand for human resources (a precious commodity), product deficiencies, late delivery, etc.

Traditional complexity measures like those of McCabe (1976), whilst not incorporating the human side of software development and maintenance, do provide some measure of the logical structure of a program. Such logic structure measures, together with measures of data structures (e.g. number of variables), size measures (e.g. lines of code, total number of tokens), style measures and internal cohesion measures offer the capability of describing the intra-module or procedural complexity as one type of product metric. (For present purposes, the words measure and metric are used interchangeably. For more precise definitions, see Melton et al. 1990). A second type of module metric is semantic cohesion which is strongly influenced by human cognition and knowledge descriptors. System-level measures are a third type of structural complexity measure which describe inter-module coupling (Figure 1).

In this paper, we focus on the ``McCabe family" of measures for the decision/logic structure of a program, which are those encapsulating the cyclomatic complexity (e.g. Myers, 1977; Piwowarski, 1982; Sagri, 1989) including extensions to describe modularization (Henderson-Sellers and Tegarden, 1993), nesting (Piwowarski, 1982) and object-oriented program structures - although we focus here on the simplest case of single-entry, single-exit modules. (Extensions to multi-entry/multi-exit modules are described in Henderson-Sellers and Tegarden, 1993). In the following section we review two definitions of software complexity. Rated, operating and essential complexities of programs as defined by McCabe (1976) and Sagri (1989) are discussed together with the introduction of two new metrics: inessential complexity and product complexity. In the final section we consider the application of these measures to modularization and nesting and some of these ideas are then discussed from an object-oriented viewpoint.

COMPLEXITY

There have been many attempts to define software complexity (e.g. Curtis, 1979; Basili, 1980). Such definitions generally identify three important elements: descriptions of tasks/problem space, resources and interacting systems. The description of complexity depends on the nature of the interacting systems which in turn is related to the resources that have to be expended in this process. The interacting systems, for example, can either be computers or programmers, or other software. When the interacting system is a computer, parameters like CPU time, the number of decisions that are to be evaluated, the number of disk accesses for performing a given task and the memory required to perform the computation all become important. These are the accidental difficulties of software that today attend its production but are not inherent in the software itself (Brooks, 1987). They may be

termed ``computational complexity" (e.g. Zuse, 1991 and Figure 1). On the other hand, when a programmer is interacting with software, the ease of performing tasks such as coding, debugging, testing, or modifying the software are of prime consideration. These tasks aim to address some aspects of the complexities of software - a construct of interlocking concepts and many competing and often contradictory requirements (Brooks, 1987) including both people and artefacts (programs). This is the area of ``psychological complexity" (e.g. Zuse, 1991) (Figure 1). Except for problems related to temporal logic, management of psychological complexity (understandability, readability, maintainability, etc.) can be extremely difficult, encompassing, as it does, problem characteristics, programmer characteristics and logical and syntactic structure within and between modules (Figure 1).

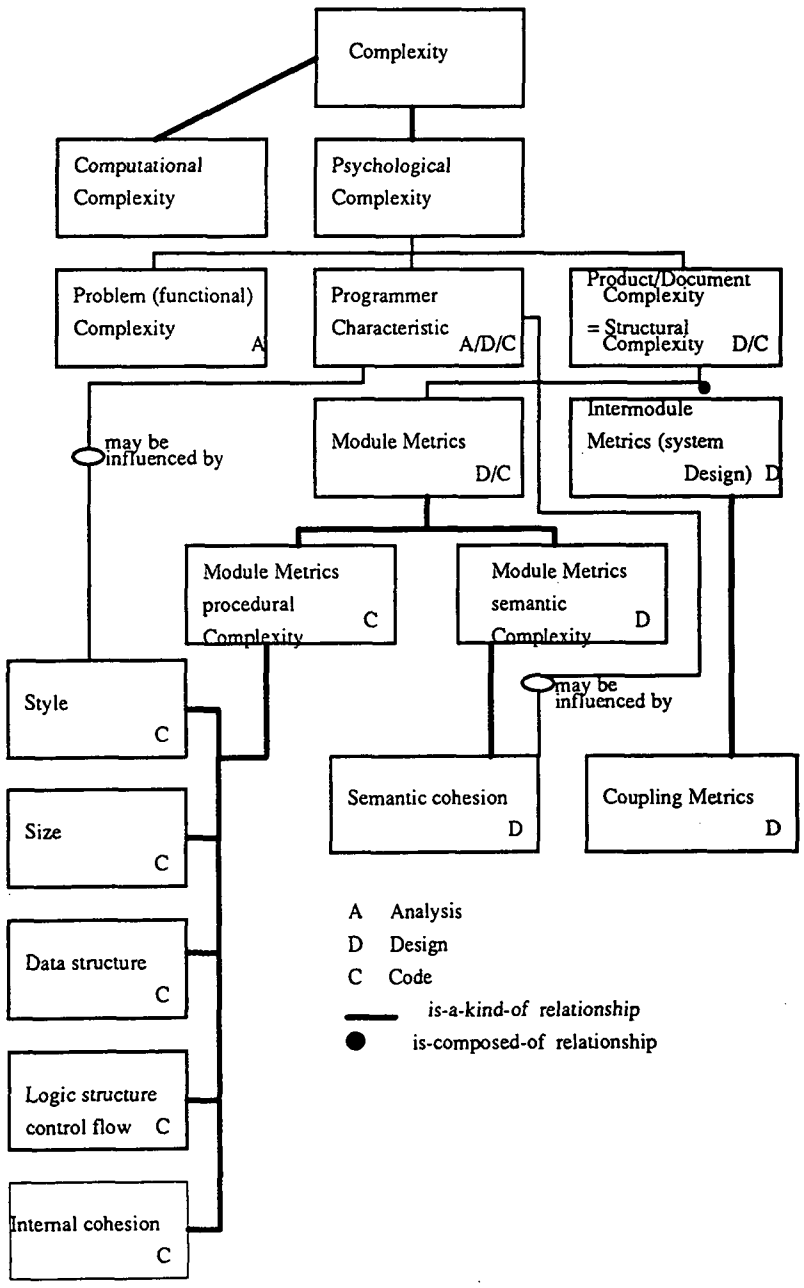

Figure 1

With the enormous gains in performance : price ratio in hardware technology in the last thirty years, execution efficiency is not normally a major issue over the entire life cycle when compared with the difficulties posed by specification, design and testing while developing and maintaining software (Brooks, 1987). Melton et al. (1990) caution against equating structural complexity with the difficulty

of understanding a piece of software. One may informally relate structural complexity with `synthesis': the creation of design and code; whilst psychological complexity relates to `analysis': dissection of existing code in order to comprehend as a prelude to maintenance. Whilst psychological complexity describes the difficult area of human--software interaction, one of its components is more easily discussed objectively: the inherent complexity of the software *product* (or ``structural complexity" --- Figure 1). Here we concentrate solely on some of the available product metrics and suggest some extensions; focussing primarily on procedural complexity mainly at the intramodule level.

PROCEDURAL COMPLEXITY MEASURES

McCabe (1976) deduced from graph theory that, for a directed acyclic graph (DAG) extracted from the code, the cyclomatic complexity, V(G), could provide a measure of the decision structure and hence assist in evaluating the testing/validation effort required. Despite the unjustified extrapolation by some authors of V(G) to a prognostic tool for reliability and effort (cf. Shepperd, 1988), the McCabe cyclomatic complexity measure remains useful as a quantification of the complexity of testing paths and may provide some insights into cognitive complexity --- although such a link has not yet been satisfactorily established, most propositions being purely speculative, not scientific (Baker et al., 1990).

V(G) is defined for a single DAG by

$$V(G) = e - n + 2 \qquad (1)$$

where e is the number of edges in the graph and n is the number of nodes. For p disjoint graphs, McCabe suggests

$$V(G) = e - n + 2p \qquad (2)$$

whilst Henderson-Sellers (1992c) and Henderson-Sellers and Tegarden (1993) recommend

$$V_{LI}(G) = e - n + p + 1 \qquad (3)$$

This latter formula measures the total number of possible decisions (= $V_{LI}(G)$ - 1) i.e. the number of simple IF statements in the whole program - compound Boolean expressions with m predicates are weighted as m decisions (McCabe, 1976).
The original use intended by McCabe was to take a single flowgraph representing a code module for which the value of V(G) could be calculated. As a consequence, the basis for testing paths could be evaluated. A second use is to apply this approach to the whole program (using either Equation (2) or (3)) to get a value of V(G) or $V_{LI}(G)$ respectively for the whole program and then use this to suggest as an heuristic to divide the program into subroutines following the classic decomposition principles as propounded by Parnas (1972).
However, essentially as a result of asymmetric nestings on these decision branches, the minimum number of decisions taken at runtime may be less than the value suggested from consideration of V(G). Sagri (1989) introduces this notion as the ``operating complexity", $v_o$, calling McCabe's original number the ``rated complexity", v_r. These two numbers represent the minimum and maximum number of possible individual decisions. He argues that the interval (v_o, v_r) provides a more useful assessment of program complexity. However, no account is taken of the distribution of values (i.e. can all values in the interval be realised), a characteristic which might also be of interest (Monarchi, 1992, p.c.).
The value of v_o is a measure probably more related to computational complexity than to structural complexity (Figure 1), suggesting that the interval may not be as useful a measure of structural complexity as originally envisaged. Furthermore, it should be stressed that the number of decisions discussed here is *not* the same as the number of paths through the module, as might be required for constructing testing procedures (see also Nejmeh, 1988). This value may be as high as 2^n where the program contains n binary decisions.

Sagri (1989) analysed five graphs g1--g5 (Figure 2) constructed from different arrangements of three *if-else* statements (two further possible arrangements of three *if-else* statements are included as graphs

g6 and g7) and found the minimum and maximum number of possible decisions that have to be made in each graph (Table 1) to arrive at the following conclusions:

1. During execution, since the number of decisions that have to be made in graphs g2 and g4 is smaller than in graph g1, it is concluded that the nested and concatenated *if - elses* (graphs

g2, g4) are less complex than un-nested *if-elses* (graph g1). The complexity ordering of these graphs is g4 < g2 < g3, g5 < g1.

2. The distinction between rated and operating complexity helps in comparing two alternative programs with the same cyclomatic number.

Graph	No. of run-time	decisions	v(G)
	min+1	max+1	
g1	4	4	4
g2	2	4	4
g3	3	4	4
g4	3	3	4
g5	3	4	4
g6	2	4	4
g7	2	4	4

Table 1: Comparison of v(g) with minimum and maximum decisions taken during execution

Table 1 also suggests that the concepts of rated and operating complexity cannot distinguish graph g3 from g5 or graph g2 from g6 or g7. Sagri (1989) specially noted that in graph g1, it is necessary to evaluate 3 decisions during run time irrespective of the path taken. In other words, it has a constant complexity in the sense that *all* runs make the same number of decisions. This gives it the highest overall operating complexity, reflecting the fact that this is a measure of computational resources used on average and has little to do with comprehension since graphs g2 and g3 are unstructured (Section 3.1) and graphs g4 and g6 contain nested structures, commonly regarded as *adding* complexity (Magel, 1981; Myers, 1977; Piwowarski, 1982) (see also discussion in Section *Modularization and Nesting*).

Whilst relying on ``intuition" to support his case (as do other authors e.g. Myers (1977) cf. Shepperd (1988)), Sagri (1989) fails to comment on three distinct anomalies of the mode of presentation of ``three IFs" in Figure 2. Firstly, graphs g2 and g3 *do not and cannot* represent *structured* code (see below). Secondly, the arguments rely solely on the graphical design resulting from the construction of a DAG from 3 decisions. This is contrary to McCabe's presentation of V(G) in which he stresses the use of V(G) in extracting a DAG from code not from design. This (mis)application of V(G) to the DAG, using it as a surrogate to infer code complexity, was not intended by McCabe (1976). Such analyses of DAGs, made by e.g. Piwowarski (1982) and Sagri (1989), frequently result in apparently well-designed logical structures which are unimplementable in a structured fashion, requiring highly complex Boolean predicates and duplicate code or else the use of unconditional GOTOs (for example, Magel's (1981) example DAG number 9).

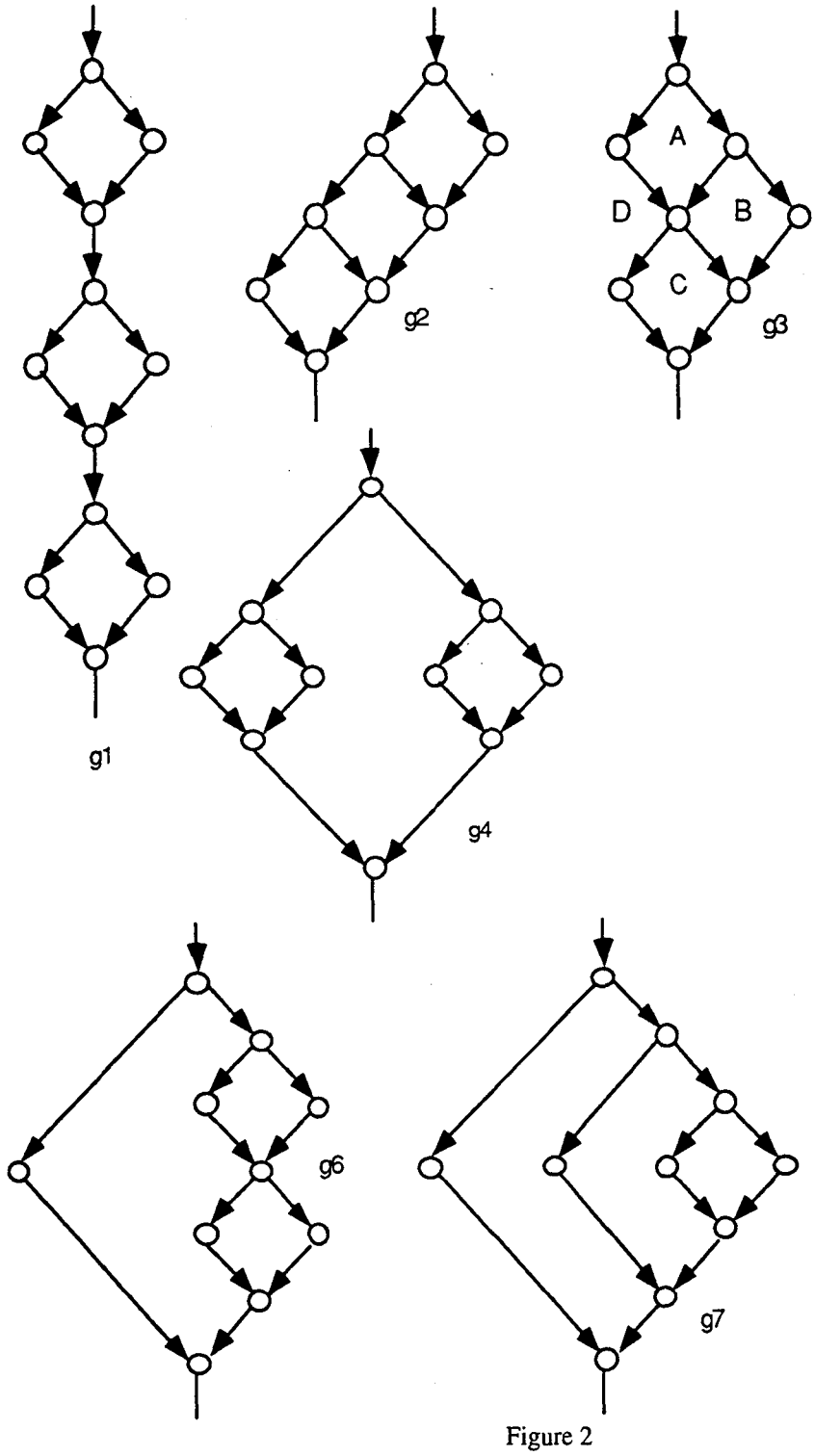

Figure 2

Finally, such discussions (as cited above) which select one graph among many indirectly suggest that it is possible to convert one graph to another. This seems highly unlikely since all the graphs presented implement different functions or algorithms and the question of favouring one among the others does not arise. The cyclomatic number of all the graphs is the same because it is a function of only the total number of *if-else* statements and not of their arrangement. Revising the algorithms and logic embodied in one graph (say g1) to a more nested structure akin to g4, g5 or g7, say, is unlikely to be feasible without changing the overall cyclomatic number (see following section). In other words, the logical constructs of the seven presented DAGs do not map into each other necessarily. This is one of the drawbacks of McCabe's cyclomatic complexity number because design complexities are often dependent on the details of the implementation. In other words, the design use of V(G) may mask

unstructured constructs such that the implemented code (when reverse engineered to a DAG) has a higher value of V(G).

Code Structuredness

It is perhaps worth reintroducing a third structural complexity measure here: that of ``essential complexity", $v_e$, as defined in McCabe's original paper. This is the complexity remaining after reduction of the DAG by removal of *all* closed cycles. Thus, $v_e$ V(G) and should equal unity for a fully structured program. Perhaps a better measure is the``inessential complexity", v_i, defined as

$$v_i = 1 - \frac{V(G) - v_e}{V(G) - 1} \qquad (4)$$

which has values in the range [0,1], if V(G) is not equal to 1. When $v_e = 1$, the ratio $\frac{V(G) - v_e}{V(G) - 1}$ has a value of unity and thus $v_i = 0$. Since $v_i = 0$ corresponds to a well-structured program, non-zero values of v_i reflect the degree to which a program has unstructured characteristics.

Sagri's graph	g1	g2	g3	g4	g5	new g6	new g7
V(G)	4	4	4	4	4	4	4
v_e	1	4	4	1	1	1	1
v_i	0	1	1	0	0	0	0
KNOTs	0	8-9*	2	0	0	0	0

Table 2: Different Measures of the graphs g1-g7 (*Dependent on coding chosen)

Out of Sagri's five original graphs, two are non-structured by McCabe's definition of containing a branch into a decision (McCabe, 1976, p316) and hence have a non-zero value of v_i (Equation (4)) - see Table 2. Since the McCabe approach is to extract a DAG from code, we must presume that Sagri's graph g3 (used as the example here) was extracted from unstructured code containing at least one GOTO as in Figure 3. As noted by Sagri, this code (and the DAG) has a value of V(G = 4 - calculated either

1. directly from Equation (1);
2. as the number of decisions (3) plus one (either from code or DAG); or
3. as the number of regions (McCabe, 1976; Bollobás,1979) = 4 (labelled A, B, C, D in Figure 2).

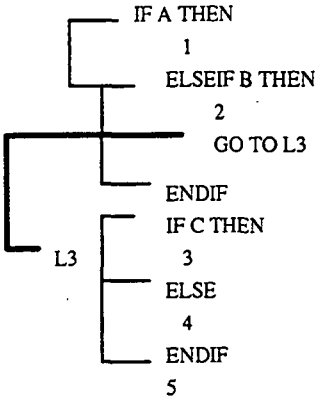

Figure 3

However, what is not stated, perhaps because it is obtainable only from the code and not from the DAGs, is that this program (Figure 3) has a non-zero value for the number of KNOTS[1] (Table 2) - a value long recognized as making programs less maintainable despite the fact that it adds nothing to V(G). In other words, it does not affect the *testability* of the code. Thus Sagri appears to confuse, implicitly, the complexity of testing (as indicated by McCabe's V(G)) with the complexity of comprehension (not discussed here). We should also note that the two measures, KNOTS and v_i, both measure the extent of unstructuredness; albeit on different scales.

Graph g3	Original	I	II	III	IV
no. of single decisions	3	7	4	8	4
V(G)	4	8	5	9	5
KNOTs	2	0	0	0	0
V(G) + KNOTs	6	8	5	9	5

Table 3: Complexity values for the five coded versions for the implementation of the graph g3 of figure 3.

As a further corollary, it is interesting to reverse the normal direction of McCabe's discussion of V(G) and ask how we might implement, for example, Sagri's g3 graph if we were presented with this graph as the design. In addition to the valid (but non-structured) implementation of Figure 3, we might well attempt to implement it in fully structured code with no unconditional GOTO statements. This can be accomplished in many ways (for example, four ways are shown in Figure 4). However, all of these *must* utilize four or more simple predicate decisions (Table 3). Although all have a KNOT value of zero, their cyclomatic complexity V(G) is greater than that of the original unstructured code. A combination f V(G) and KNOTS seems to offer a rationalization insofar as codes I and III have a value of V(G) (=7) identical to the value of [V(G) + KNOTS] for Sagri's original graph.

Whilst acknowledging that the structured/non-structured divide was a more prevalent question at the time of McCabe's original proposition (1976), although still worthy of discussion today (Prather, 1984) the discussion here has highlighted:

1. the important asymmetry in translating from code to DAG compared to implementing a DAG in code;
2. the apparent increase in complexity (if measured solely by V(G)) when moving from unstructured to structured code (also noted by Myers, 1977); and
3. the need to characterize non-structuredness by measures other than V(G) (see for example the large value of KNOTS for graph g2 - Table 2).

This discussion also lays the groundwork for our discussion of modularization and nesting (following section) and the role of abstraction, especially in complex structured and object-oriented software systems.

MODULARIZATION AND NESTING

Modularization, by removing one or more control structures and replacing them by module ``CALLs", does not, *per se*, change the value of V(G) of the overall system (as calculated by Equation (1)) (Henderson-Sellers, 1992c). However, it does discretize the program into a larger number of cognitively comprehensible smaller chunks --- an idea to be pursued in our discussion of object-oriented systems in the following section. However, if this process also eliminates duplicate code containing at least one control structure, then the structurally-related value of V(G) will generally decrease (Shepperd, 1988; Henderson-Sellers and Tegarden, 1993), although the cognitive effort of such a process may be different (a discussion of which is beyond the scope of this present paper).

[1]The value of KNOTS (Woodward et al., 1979) is calculated as shown in Figure 3 by drawing lines between IF/ELSE/ENDIF structures and loops and also between GOTO statements and their target

```
I
IF A THEN
        1
        IF C THEN
                3
        ELSE
                4
ELSEIF B THEN
        2 3
        ELSEIF C
                3
ENDIF
IF NOTA AND NOTB AND NOTC THEN
        4
ENDIF
5
```

5 IFs equivalent to 7 decisions
Code block 4 duplicated and block 3 appears 3 times

```
II
IF A THEN
        1
        IF C THEN
                3
        ELSE
                4
        ENDIF
ELSEIF B THEN
        2 3
        ELSEIF C THEN
                3
        ELSE
                4
ENDIF
5
```

4 IFs Code block appears 3 times and block 4 twice

```
III
IF A THEN
        1
ELSEIF B
        2
        3
ENDIF
IF NOTA AND B THEN
        blank
ELSEIF (A AND C) OR (NOTB AND C)
        3
ELSE
        4
ENDIF
5
```

4 IFs equivalent to 8 decisions
Code block 3 duplicated

```
IV
IF A THEN
        1
        IF C THEN
                3
        ELSE
                4
        ENDIF
ELSE IF B
        2 3
        ELSEIF C
                3
        ELSE
                4
                ENDIF
        ENDIF
ENDIF
5
```

4 IFs Code block duplicated and block 3 triplicated

Figure 4

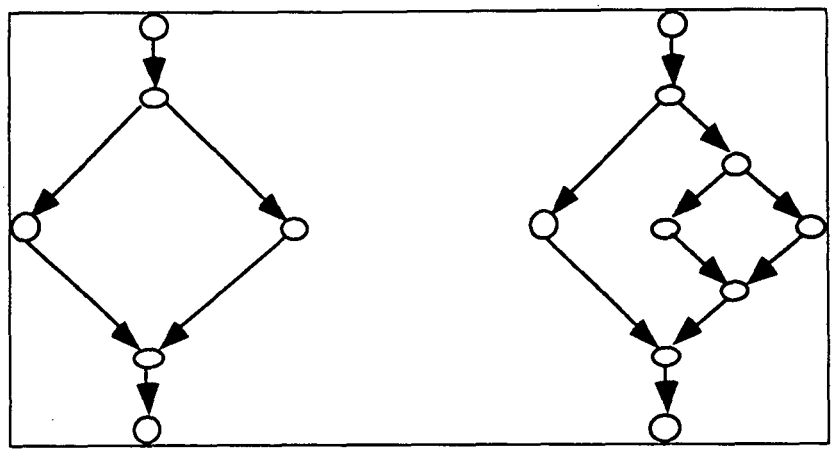

Figure 5 Nested Control Structure adds to the Cyclomatic Complexity

On the other hand, the addition of a nested control structure adds to the cyclomatic complexity (Piwowarski, 1982), whilst leaving the essential complexity value unchanged (Figure 5). Un-nested *if-elses* and loops will be simpler to construct for a programmer than nested *if-elses*/loops, contrary to the conclusions of Sagri (1989), because it is easier mentally to handle one independent *if-else*/loop at a time. In other words, only one level of abstraction needs to be ``remembered" at one time. Thus the control structure forms a single, easily understood ``chunk" (Cant et al., 1992). Embedding a second control structure inside this top-level chunk disrupts the flow of cognition by requiring ``tracing" --- viz. a change of abstraction level --- visualized by the landscape diagram of Cant et al. (1992). Disregarding the computations embedded within the block, it is obvious that one independent *if--else*/loop commonly requires less effort for cognitive processes like chunking and tracing because of constraints of our short-term memory (Cant et al., 1992). On the other hand, nested *if--elses* also contain implicit *and*s, and thus create distinct paths (and naturally more side effects), which the programmer has to take into account while writing *or* modifying programs. Similarly, in a doubly-nested loop structure the programmer/analyst needs to bear in mind the values of 2 control variables - chunking and tracing between the two levels of abstraction continually. It is obvious that the logical structure of nested control structures is difficult to assess fully without the inclusion of programmer cognition. In this present paper, we retain a focus on artefacts measurable independently of cognitive complexity issues (cf. Cant et al., 1992).

In this context, we re-evaluate V(G), v_0 and v_e from the ``McCabe family", together with ``nesting complexity (N)" designed specifically to address the questions of nesting (Magel, 1981; Piwowarski, 1982).

Piworwarski's	Sequential		Nested		Unstructured	
	A	B	C	D	E	F
No. of single decisions	2	2	2	2	2	2
V(G)V*(G)	3	3	3	3	3	3
Operating Complexity (v_0)	3	3	2	3	2	3
Essential Complexity (v_e)	1	1	1	1	3	3
N* (Equation 6)	0	0	1	1	0	0
Piwowarksi's N	3	3	4	4	5	5
KNOTs	0	0	0	0	1	1
P_1	3	3	4	4	5	5
P_2	3	3	4	4	9	9

Table 4: Values for the 6 programs/graphs of Piwowarski. (Here V(G)V*(G)). Also indicated are values of P_1 and P_2 given by equations (7) & (8)

In Table 4 are presented values for these three complexity measures and Piwowarski's (1982) N for his six example DAGs (Figure 6). The value of N is given by

$$N = V^*(G) + N^* \qquad (5)$$

where V*(G) is an adjusted cyclomatic complexity number, in which case/switch structures are treated as a single predicate (similar to Myers' (1977) lower bound) and N* represents the effect of nested control structures and is given by

$$N^* = {}_i P^*(i) \qquad (6)$$

where P*(i) is the nesting depth of the ith predicate defined as the number of predicate node scopes overlapped or contained by the ith predicate node.

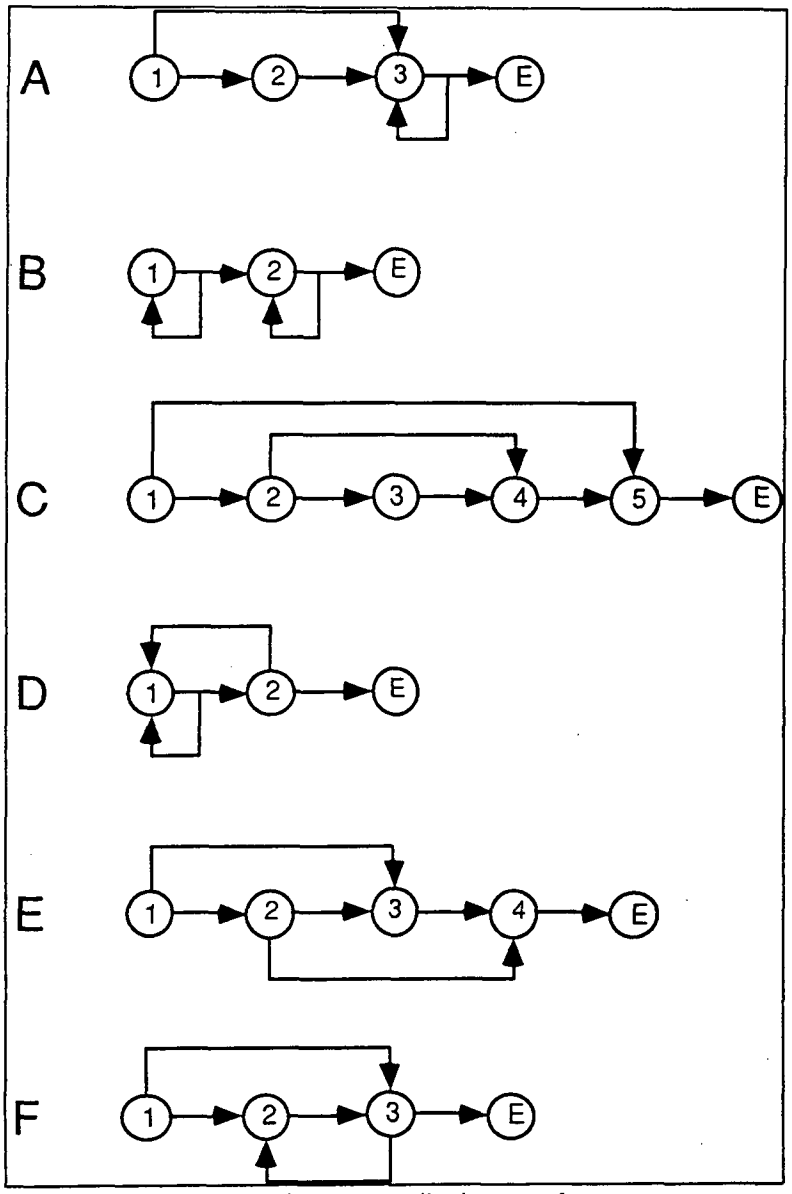

Figure 6 Piwowarski's six examples

Piwowarski's (1982) intentions are to derive a measure which gives high values for unstructured programs, intermediate values for structured programs with nested control structures and lowest values for sequential code with only top level control structures embedded. As such, it penalizes programs, such as E and F in Figure 6, in which KNOTS exist. Thus N reflects not only the cyclomatic complexity but also the nesting levels and the degree of unstructuredness. The measure could alternatively be expressed

$$N = V^*(G) + N^* + 2K \qquad (7)$$

(where N* is redefined as

$$N^* = {}_i P(i) \qquad (8)$$

where P(i) is the straightforward nesting depth of the ith predicate and K is the knot value), i.e. the sum of the adjusted cyclomatic complexity, allowing for nesting (each nesting level in each control structure adding one to N), and twice the KNOT value. The former provides an additional weighting to evaluate nesting complexity whereas the latter addresses the very different issue of unstructuredness (see above). We recommend maintaining these two descriptions of disparate characteristics as separate values:

1. V*(G) modified by true nesting values (P(i) not P*(i)) and

2. KNOTS.

Sagri's graph	g1	g2	g3	g4	g5	g6 (new)	g7 (new)
V(G)V*(G)	4	4	4	4	4	4	4
V*(G) + (Pi)	4	4	4	6	5	6	7
KNOTs	0	8-9*	2	0	0	0	0
Piwowarski's N	4	20-22	8	6	5	6	7
v_e	1	4	4	1	1	1	1
v_o	0	1	1	0	0	0	0
P_1	4	7	7	6	5	6	7
P_2	4	16	16	6	5	6	7

Table 5: Values of various complexity metrics for the seven graphs based on Sagri's (1989) disussion (*Dependent on coding chosen)

Applying this philosophy to Sagri's graphs gives the values shown in Table 5. It can be seen that the ``least complex" of the three nested graphs (g4--g7) is identified, using N, as g5; that the two unstructured DAGs have large, non-zero values for KNOTS and a large (fractional --- actually unity) value for v_i; and that the basic cyclomatic complexity number reflects the number of decisions.

As a tentative indication of the modelling approach we intend to pursue, we note that the factors v_e, N* (Equation (8)) and V(G) all appear to be monotonic increasing functions of ``complexity". An appropriate additive model might therefore be

$$P_1 = a_1 V(G) + a_2 N^* + a_3 (v_e - 1) \qquad (9)$$

(the subtraction of 1 from v_e being necessary since $v_e = 1$ is a well-structured program adding no additional complexity); a multiplicative model might be

$$P_2 = a_3 v_e [a_1 V(G) + a_2 N^*] \qquad (10)$$

where the weights, a_1, a_2, a_3 will depend upon cognitive comprehension issues, as well as the task being performed. That this is a reasonable initial step (until dimensionality, meaning and cognitive concerns can be adequately evaluated) is seen in the last two lines of Tables 4 and 5 where both ``product complexity metrics", P_1 and P_2 are given with the (unjustified) simplistic approach that $a_1 = a_2 = a_3 = 1$. The overall distribution is seen (Figure 7) to reflect the broad differences we might anticipate between nested, sequential, structured and unstructured programs.

3 decisions (table 5)
Piwowarski (table 4)

unstructured					2		2
nested				2	1	2	1
sequential			2	1			
	1	2	3	4	5	6	7

unstructured							2			
nested		2	1	2	1					
sequential	2	1								
	3	4	5	6	7	8	9	10	...	16

Figure 7

The Role of Abstraction

Abstraction provides a powerful tool in both analysis and design, as reflected in the recent surge of interest in the use of abstract data types (ADTs) and object-oriented methodologies (Henderson-Sellers, 1992b; McGregor and Sykes, 1992). The use of abstraction permits the rationalization of complexity by permitting the designer to concentrate *sequentially* on the design at different abstraction levels. This is closely associated with and in some sense opposite to the process of ``prime decomposition" (e.g. Fenton, 1991, p175) and ties in closely with the chunking concepts of the cognitive complexity model (Cant et al., 1992).

For example, the un-nested *if-elses* represents a series of simple abstractions, while the outer *if-else* in a nested *if-else* (e.g. graph g4 of Figure 2) can be considered to be at a higher level of abstraction subsuming, as one of its immediate lower-level abstraction, a single *if-else*. Figure 2 (graph g7) adds one more level of nesting so that this graph can be viewed at any of three abstraction levels. The innermost control block can be considered independently; then at the next higher level of abstraction, this whole block can be viewed as a single node (Figure 8); and so on. Such a utilization of abstraction levels elaborates on the process, described by McCabe (1976) for reducing a flow graph to evaluate its essential complexity, v_e.

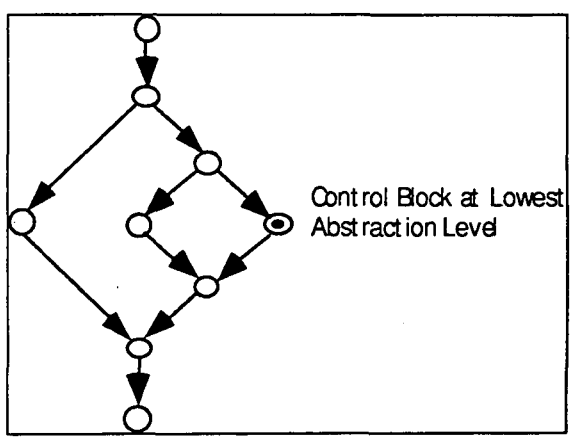

Figure 8 The Innermost Control Block can be considered independently

In object-oriented systems, abstractions are manifest in the form of encapsulated object classes connected through networks. Indeed, object modelling focusses on the use of various levels or degrees of abstractions as a crucial tool (Henderson-Sellers, 1992a; Selic et al., 1992). As we move to higher levels of abstraction, we tend to concern ourselves with progressively smaller volumes of information, and fewer overall items. As we move to lower levels of abstraction, we reveal more details, typically encountering more individual items, thus increasing the volume of information with which we must deal. Further research on object-oriented structural complexity is clearly warranted, although we note that initial investigation of the usefulness of an unmodified McCabe cyclomatic number (Tegarden et al., 1992) suggests that direct application of V(G) can at least highlight the impact of polymorphism

and inheritance although its use for detecting problems in these areas awaits evaluation since V(G) and $V_{LI}(G)$ are best suited to describing control flow structures not architectural configurations.

SUMMARY AND CONCLUSIONS

One of the major problems in complexity research is that of finding an objective assessment of the term complexity. Whilst it is increasingly recognized (e.g. Fenton, 1991) that the term *cannot* have a single, all-encompassing interpretation, it may be possible to define the term in very specific domains. For instance, the use of flow graphs discussed here can be evaluated with respect to numbers of decisions and control flow but cannot evaluate complexity due to convoluted variable plans, to linear algorithms nested between the decision nodes of the flowgraph or due to cognitive difficulties.
We can conclude, therefore, that flow graph complexity measures, such as those of McCabe (1976), Piwowarski (1982), Sagri (1989) and Henderson-Sellers (1992c) can measure decision routes and hence may be applicable to maintenance and testing - as originally envisaged by McCabe (1976). McCabe's (1976) original cyclomatic complexity, measuring decisions, supplemented by metrics for nesting and knots, has been rationalized in terms of new ``product cyclomatic" metrics, P_1 and P_2. However, the application of cognitive complexity models (Cant et al., 1992) may prove useful in this context and is the subject of ongoing research (Cant et al., 1992). Flowgraph-based complexity measures may also have some validity in design, to the extent and with the caveats indicated above. However, in design, a graph such as g5, g6 or g7 (Figure 2) with extensive use of nesting reflects more accurately recent software enginering techniques based on abstraction techniques which are reflected in the increasingly popular object-oriented paradigm.
Further work on complexity, including acknowledgement of psychological considerations and human factors, is required to clarify the operational role of complexity measures in both a structured and object-oriented software development environment.

ACKNOWLEDGEMENTS

We wish to acknowledge the support of a Small Research Grant (grant number C450.301) from the Australian Research Council. We also wish to thank Graham Tate and David Monarchi for their useful comments on the draft of this manuscript.

REFERENCES

Baker, A. L., Bieman, J. M., Fenton, N., Gustafson, D. A., Melton, A., Whitty, R., (1990), A philosophy for software measurement, **J. Syst. Soft.**, vol. 12, no. 3, pp. 277--281

Basili, V. R., (1980), Qualitative software complexity models: a summary, **in Tutorial Models and Methods for Software Management and Eng**, IEEE Press

Bollobás, B., (1979), **Graph theory: an introductory course**, Springer--Verlag, New York, p. 180

Brooks, F. P., (1987), No silver bullet - essence and accidents of software engineering, **IEEE Computer**, vol. 20, no. 4, pp. 10--19

Cant, S. N., Jeffery, R. D., Henderson-Sellers, B., (1992), A conceptual model of cognitive complexity of elements of the programming process, Centre for Information Technology Research Report No. 57, 1992, University of New South Wales, Sydney, Australia (also submitted for publication)

Conte, S. D., Dunsmore, H. E., and Shen, V. Y., (1986), **Software engineering metrics and models** Benjamin/Cummings, Menlo Park, CA, p. 386

Curtis, B., (1979), In search of software complexity, **Workshop on Quantitative Software Models** pp. 95--106

Fenton, N. E., (1991), **Software metrics: a rigorous approach**, Chapman and Hall, 337pp

Henderson-Sellers, B., (1992a), Object-oriented information systems: an introductory tutorial, **Australian Computer J.**, vol. 24, no. 1, pp. 12--24

Henderson-Sellers, B., (1992b), **A Book of Object-Oriented Knowledge**, Prentice Hall, Sydney, 297pp

Henderson-Sellers, B., (1992c), Modularization and McCabe's cyclomatic complexity, **Commun. ACM**, vol. 35, no. 12, pp. 17--19

Henderson-Sellers, B. and Tegarden, D. P., (1993), The application of cyclomatic complexity to multiple entry/exit modules (submitted for publication)

Magel, K., 1981, Regular expressions in a program complexity metric, **ACM SIGPLAN Not** vol. 16, no. 7, pp. 61--65

McCabe, T. J., (1976), A complexity measure, **IEEE Trans. Soft. Eng.**, vol. 2, no. 4, pp. 308--320

McGregor, J. D. and Sykes, D. A., (1992), **Object-oriented software development: engineering software for reuse** Van Nostrand, Reinhold, New York, p. 352

Melton, A. C., Gustafson, D. A., Bieman, J. M., and Baker, A. L., (1990), A mathematical perspective for software measures research, **Software Eng. J.**, vol. 5, pp. 246--254

Myers, G. J., (1977), An extension to the cyclomatic measure of program complexity, **ACM SIGPLAN Not** vol. 12, no. 10, pp. 61--64

Nejmeh, B. A., (1988), NPATH: A measure of execution path complexity and its applications, **Commun. ACM** vol. 31, no. 2, pp. 188--200

Parnas, D. L., (1972), On the criteria to be used in decomposing systems into modules, **Commun. ACM** vol. 15, no. 12, pp. 1053--1058

Piwowarski, P., (1982), A nesting level complexity measure, **ACM SIGPLAN Not.**, vol. 17, no. 9, pp. 44--50

Prather, R. E., (1984), An axiomatic theory of software complexity measure, **Comp. J.**, vol. 27, no. 4, pp. 340--347

Sagri, M., (1989), Rated and operating complexity of program - an extension to McCabe's theory of complexity measure, **ACM SIGPLAN Not.**, vol. 24, no. 8, pp. 8--12

Selic, B., Gullekson, G., McGee, J., and Engelberg, I., (1992), ROOM: An object-oriented methodology for developing real-time systems, in **CASE'92 Fifth Int. Workshop on Computer-Aided Software Eng.** Montreal, Quebec, Canada

Shepperd, M., (1988), A critique of cyclomatic complexity as a software metric, **Software Eng. J** vol. 3, pp. 30--36

Tegarden, D. P., Sheetz, S. D., Monarchi, D. E., (1992), Effectiveness of traditional software metrics for object-oriented systems, **HICSS-92, IEEE**, San Diego pp. 359--368

Woodward, M., Hennell, M., and Hedley, D., (1979), A measure of control flow complexity in program text **IEEE Trans. Soft. Eng.**, vol. 5, no. 1, pp. 45--50

Zuse, H., (1991), **Software complexity measures and models**, de Gruyter and Co.