AN INVESTIGATION INTO CHARACTERISTIC POLYNOMIALS AND TSAI'S POLYNOMIALS

by

DEIVAN DURAI

B.Sc.(Ag.)., Tamil Nadu Agricultural University, 1976
M.Sc., Indian Agricultural Research Institute, 1978
Ph.D., Kansas State University, 1985

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

Major Professor

TABLE OF CONTENTS.

LIST OF TABLES.

LIST OF FIGURES.

ACKNOWLEDGMENTS

My sincere thanks to my major professor, Dr. David A. Gustafson, for his guidance and suggestions, and most of all, for his sensitivity to my occasional problems. His input into this report is considerable, to say the least. One would be hard pressed to choose a better major professor and a better friend. My sincere appreciation to Drs. Virgil E. Wallentine and William J. Hankley for their reviews and criticisms. There is one person that deserves a most special mention, Dr. Donald P. Hoyt, director of Planning and Evaluation Services and my current boss. This report would have gotten delayed much more were it not for his encouragement, support, and understanding. I had never dreamed of working under such a congenial and sympathetic overseer. There is much to gain from his acumen and wisdom that I am thrilled to be associated with him.

It would be utterly impossible to fully acknowledge the support and love I have received from my wife Kannaki but she deserves all I could ever give to any one. She is one precious woman in my life and nothing less than dedicating this report to her would do justice. Hence, this report is dedicated to her. Thanks to my children Vijay and Jayanthi for the love they have poured out to me with no knowledge of why I was away from home much of the time.

Chapter 1.

INTRODUCTION

There exists an assortment of software complexity measures in the literature, some of them celebrated. This report describes the examination of two different complexity measures. The first one is a source code complexity measure constructed by Cantone, Cimitile, and Sansone[1] in which the complexity of a program is computed as a polynomial. The second one, also computed as a polynomial, is a data structure complexity measure. This measure was developed by Tsai, Lopez, Rodriguez, and Volovik[2].

Software complexity is made up of two parts, namely, complexity of the source code and that of the data structure. They are called *source code complexity,* and *data structure complexity,* respectively. Discussion of software complexity has

[1]Cantone, Giovanni, Aniello Cimitile, and Lucio Sansone, "Complexity in Program Schemes: The Characteristic Polynomial", *SIGPLAN Notices*, vol. 18, no. 3, March 1983.

[2]Tsai, W.T., M.A. Lopez, V. Rodriguez, and D. Volovik, "An Approach to Measuring Data Structure Complexity", *Proceedings of IEEE Computer Society's Tenth International Computer Software and Applications Conference*, 1986, pp. 240-246.

almost always meant the complexity of the source code and rarely referred to the complexity associated with the data structure.

The source code complexity measure that this study concentrates on and as developed by Cantone and others, is called *characteristic polynomial.* Giving credit to the developers and for lack of better names, the data structure complexity measure will be called *Tsai's data structure complexity measure* or *Tsai's polynomial.*

In addition to a description of the two complexity measures, the inquiry in this study further involves:

1. investigation of the existence of *equivalence classes* and/or *partial orderings* related to characteristic polynomial and to Tsai's data structure complexity,
2. development of a *mapping measure* which maps the characteristic polynomial [and Tsai's polynomial] on to real numbers for straightforward comparison,
3. verification that the mapping measure preserves the partial order of the characteristic polynomial and Tsai's polynomial,
4. validation of both the complexity measures utilizing a complexity measures validation paradigm, and
5. discussion of the possibility of combining/collapsing of characteristic polynomial and Tsai's polynomial to arrive at an *extended software complexity measure.*

1.1. ORGANIZATION OF THE REPORT:

Chapter 2 describes characteristic polynomials as developed by Cantone, Cimitile, and Sansone and describes its properties. Tsai's polynomials constructed by Tsai, Lopez, Rodriguez, and Volovik and their properties are covered in Chapter 3. A few mapping measures that map these polynomial complexity measures to real numbers are introduced in Chapter 4. Investigation of the existence of equivalence classes and/or partial orders with respect to characteristic polynomials and Tsai's polynomials are examined respectively in Chapters 5 and 6. A brief discussion of problems in combining characteristic polynomials and data structure complexity is given in Chapter 7.

Characteristic polynomials, as defined by Cantone, et. al., are obtained using directed graphs whereas Tsai's data structure complexity polynomial that gauges the complexity of a program's data structure is obtained utilizing directed multigraphs. Even though Cantone and others have discussed the development of characteristic polynomials by basing them on directed graphs, their measure is really based on structured flowgraphs. A discussion, therefore, of digraphs, directed multigraphs, flowgraphs, structured flowgraphs, and other related concepts is perceived to be useful. However, to avoid such a discussion from cluttering the main aspects of the study, it is presented in the appendix [see Appendix A]. Additionally, concepts associated with relations, orders, and flowgraph transformations which are used in this study are also discussed in Appendix A.

Appendix B describes a complexity measure validation paradigm as proposed by Baker, Bieman, Gustafson, and Melton[3]. This paradigm is used in validating the above two complexity measures.

[3]Baker, Albert L., James M. Bieman, David A. Gustafson, and Austin C. Melton, "Modeling and Measuring the Software Development Process", *Proceedings of Hawaiian International Conference on Computers and Software*, 1987, pp. 23-30.

Chapter 2.

CHARACTERISTIC POLYNOMIALS

2.1. SOURCE CODE COMPLEXITY:

Complexity measures have traditionally been constructed for programs rather than for data structures. This section describes a source code complexity measure called characteristic polynomial.

Complexity measures, most notably McCabe's cyclomatic number and Halstead's software science measure, do not consider control environments, cycles, selections, and nesting. Cantone, et. al., construct a complexity measure that takes into account the structural characteristics of the control flow of a program and the number of its possible executions. Their construction is called the *characteristic polynomial.* A program's characteristic polynomial is easily constructed from its flowgraph.

2.2. FLOWGRAPHS versus CHARACTERISTIC POLYNOMIALS:

Flowgraphs have been the basis for several complexity measures. Even though characteristic polynomials could be constructed without utilizing flowgraphs, it is much easier to visualize the computation of characteristic polynomials with the aid of flowgraphs. Furthermore, characteristic polynomials are constructed in this study [as well as in Cantone[4] and others' original work] only for structured flowgraphs that avoid flowgraph representation for spaghetti code. Structured flowgraphs/programs have become the norm and hence, computing characteristic polynomials for unstructured programs is not attempted.

Since the complexity of source code is related to the nesting of its control structures, the complexity increases as the level of nesting of the control structures of the program increases. In constructing the characteristic polynomial, therefore, different criteria are applied to obtain the various components of the characteristic polynomial.

2.3. COMPUTATION OF CHARACTERISTIC NUMBERS AND POLYNOMIALS:

The characteristic polynomial of a structured flowgraph is obtained by combining what is termed by Cantone[4] et. al., *characteristic numbers* of its subgraphs and primitive nodes. Understandably, a characteristic number can be associated with each node of a flowgraph. The following criteria are used in computing the characteristic numbers:

---

[4]Cantone, et. al., op. cit.

- for each *primitive node* the characteristic number is *1*.
- for a *structure of the serial type,* the characteristic number is given by the *product of the characteristic numbers* of the structures and of the primitive nodes which are immediately contained in it. This means that if a digraph is made up of a sequence of primitive nodes representing a program with no looping and branching statements, then its characteristic number is the same as that of a single primitive node.
- for a *structure of the selective type,* the characteristic number is given by the *sum of the characteristic numbers* of the structures and of the nodes which are immediately contained in it. This value coincides with the number of total paths from the start node to the terminal node of the program flowgraph.
- for a *cyclic structure,* the characteristic number is given by the *product of a variable c and the characteristic number* of the unique structure or primitive node immediately contained in it. Variable c is used to distinguish the structural characteristics of the control flow of a program/flowgraph and consequently the structure of the characteristic polynomial expression.
- the characteristic number of a *program* is the characteristic number of the level m graph associated with the program.
- the general form of a characteristic polynomial is $\Sigma a_i c^{\alpha_i}$.

2.4. PROPERTIES OF CHARACTERISTIC POLYNOMIALS:

Since the characteristic number of a representative program is non-linear in c, it is called the characteristic polynomial of a program. However, note that the characteristic polynomial of every program is not non-linear in c.

Briefly, the characteristic polynomial of a program without cyclic structures is of degree 0 in variable c and the value of the characteristic polynomial coincides with the total number of paths of the program. A program or subprogram with a cyclic structure with no other nested cyclic structure contained in it is represented by a polynomial of degree 1 in variable c. The coefficient of the term of the first degree coincides with the number of elementary cycles present in the digraph. Cantone, et. al., use the phrase *exemplary executions of a cyclic structure* to denote any execution that involves only the execution of the instructions associated with the nodes of one elementary cycle of the corresponding digraph. A complete characteristic polynomial denoted by $\Sigma a_i \cdot c^{\alpha_i}$, represents a program for which there are Σa_i distinct exemplary executions. In particular, a_n are exemplary executions characterized by n cycles, a_{n-1} are exemplary executions characterized by n-1 cycles, ..., a_0 are exemplary executions without cycles.

AN EXAMPLE:

Structured flowgraph transformations, viz., *composition, alternation,* and *iteration transformations,* discussed in Appendix A, are *incremental and progressively complex* in the sense that if $a \in \mathcal{A}$ is a structured flowgraph [where $\mathcal{A}$ is a set of

flowgraphs] with some measured complexity in any intended component of complexity, then any structured flowgraph transformations exercised on a will not reduce the complexity of a. Therefore, instead of computing the characteristic polynomials for a sample collection of flowgraphs--which will yield non-comparable characteristic polynomials--the computation is made for flowgraphs which are the result of successive employment of structured flowgraph transformations on a given flowgraph. Note that, based on an assumption made in the *software complexity measures paradigm* [see Appendix B], the set $\mathcal{A}$ can be treated as a preordered set.

The application of the structured flowgraph transformations that can be performed on a structured flowgraph will be obvious in the following example. Composition, alternation, and iteration transformations have been used to arrive at the successive flowgraphs. The characteristic polynomial is also computed and shown alongside for each flowgraph. Structured flowgraph transformations modify characteristic polynomials in such a fashion that it becomes difficult to compare them directly. Further, one natural objective in software complexity measures research is to map software complexity measures to real numbers. Hence, measures that reduce the characteristic polynomials to real numbers will be adopted. These measures make it possible to compare the complexity of the flowgraphs directly.

Structured flowgraphs that are increasingly complex due to successive application of structured flowgraph transformations are provided in the following pages. *Table 2.1* presents the characteristic polynomials computed

$CP = 4c$

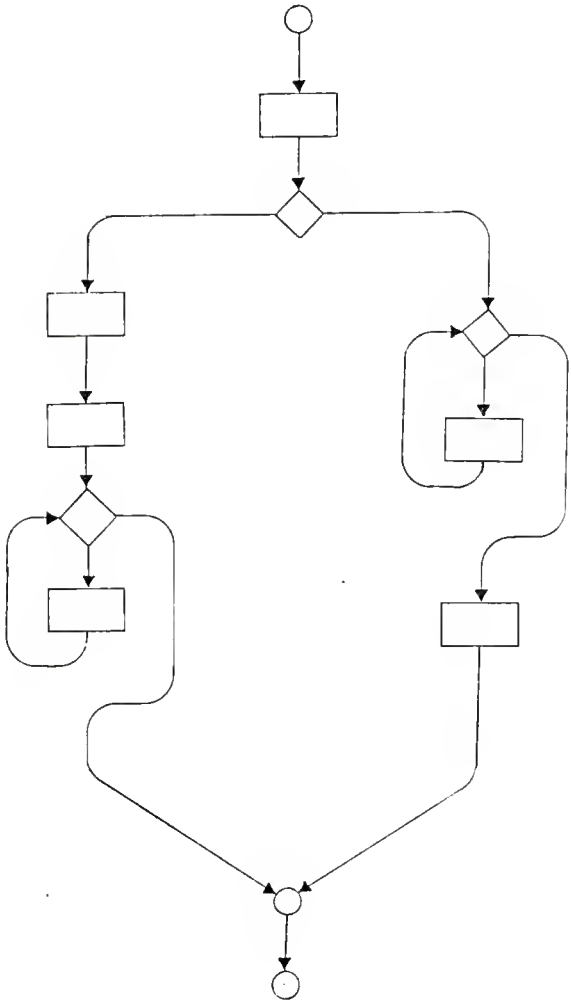

Figure 2.1

$CP = 5c$

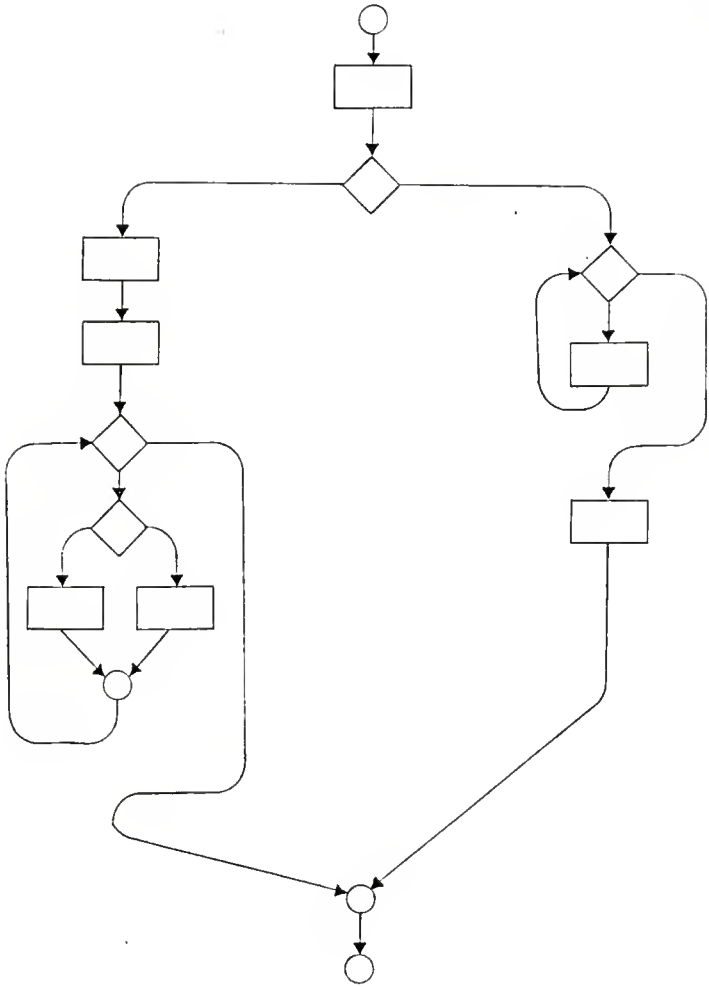

Figure 2.2

$CP = 6c^2 + 2c$

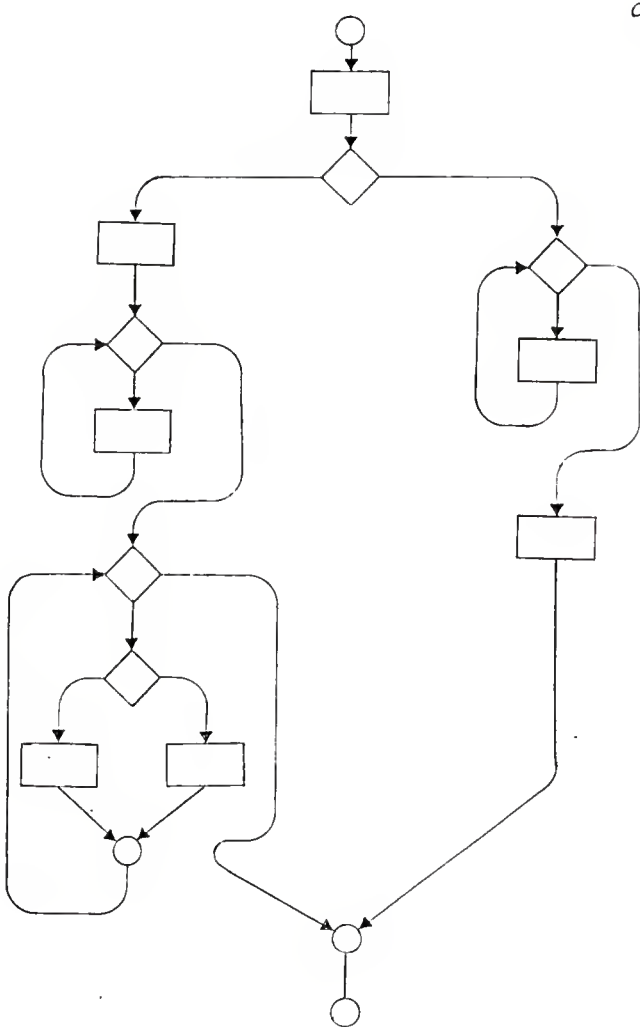

Figure 2.3

$CP = 6c^2 + 2c + 1$

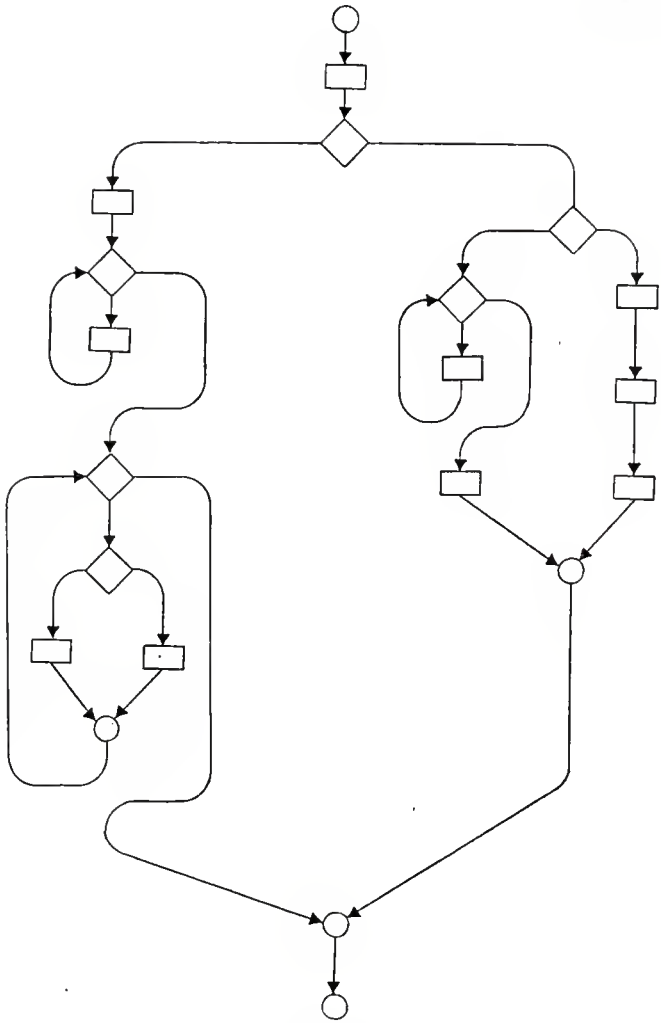

Figure 2.4

$CP = 6c^2 + 4c$

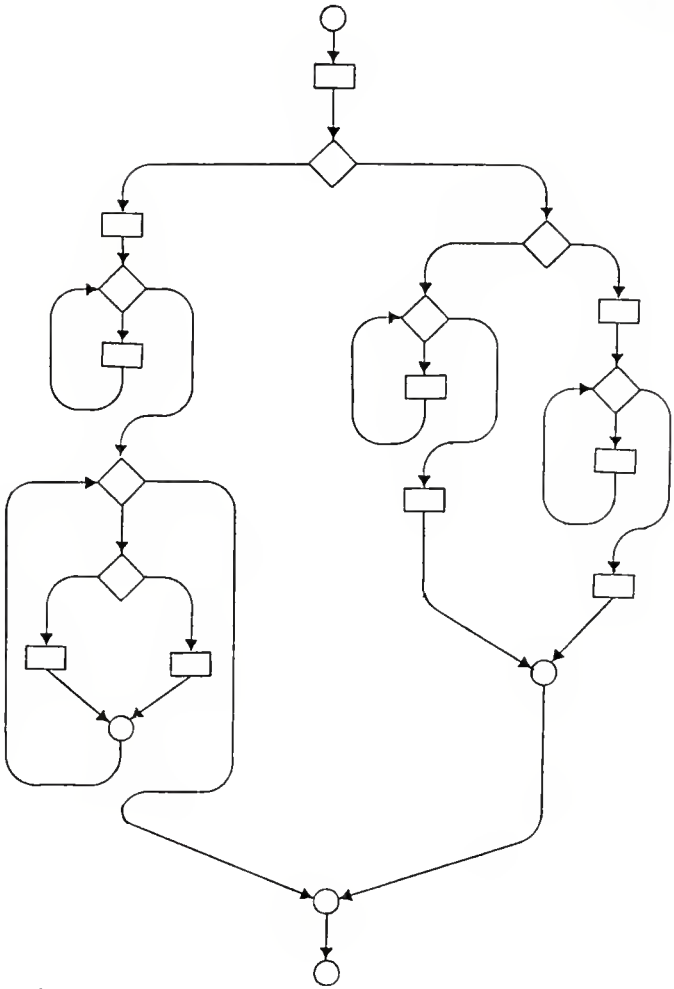

Figure 2.5

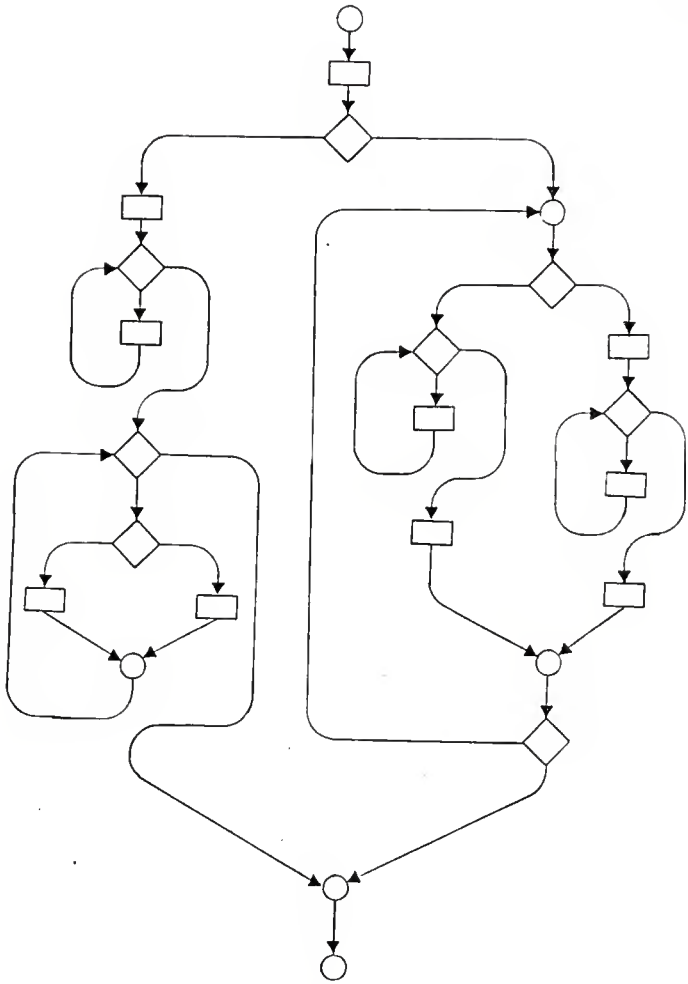

Figure 2.6

$CP = 4c^3 + 8c^2$

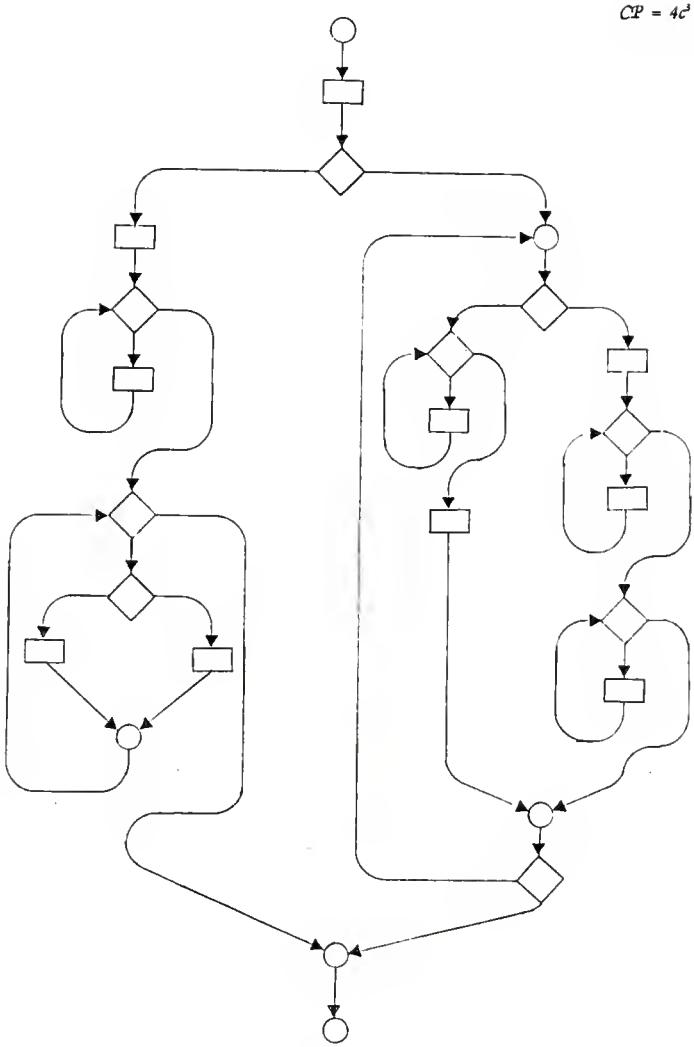

Figure 2.7

$CP = 8c^3 + 6c^2$

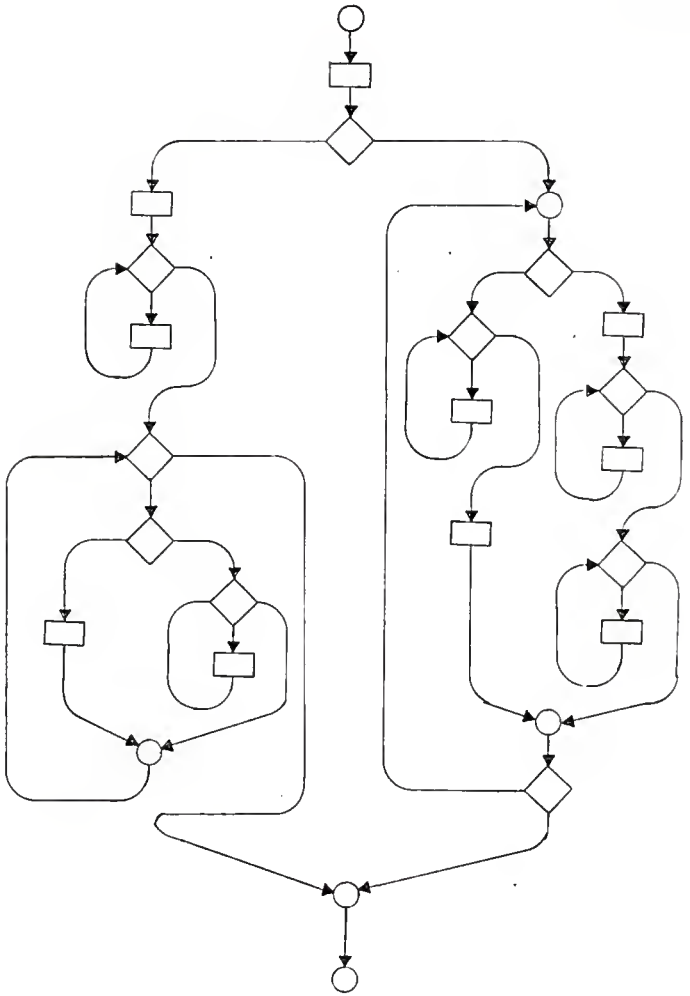

Figure 2.8

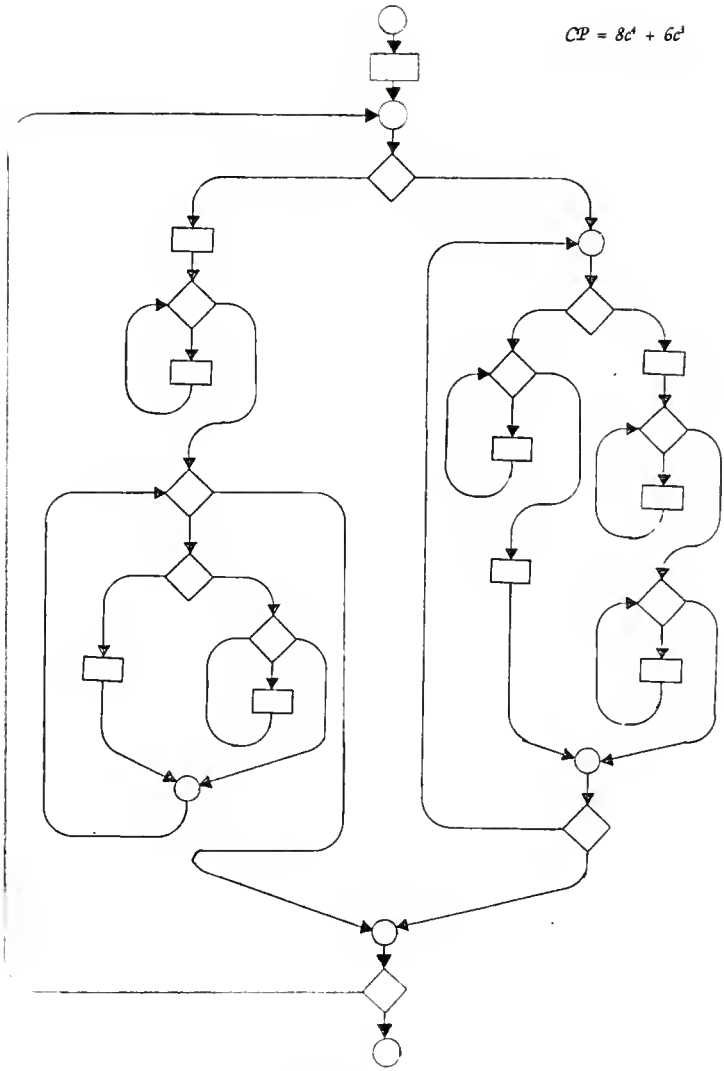

Figure 2.9

Flowgraph	Kind of transformation applied on the predecessor	Characteristic Polynomial
2.1	Initial flowgraph	$4c$
2.2	Alternation	$5c$
2.3	Iteration	$6c^2 + 2c$
2.4	Alternation	$6c^2 + 2c + 1$
2.5	Iteration	$6c^2 + 4c$
2.6	"	$10c^2$
2.7	"	$4c^3 + 8c^2$
2.8	"	$8c^3 + 6c^2$
2.9	"	$8c^4 + 6c^3$

Table 2.1. Transformations and Characteristic Polynomials.

for these structured flowgraphs along with the kind of transformation that modified the previous flowgraph.

Note that many junction nodes in *Figure 2.1* through *Figure 2.9* have been deliberately left out since it reduces the clutter and these junction nodes have no effect of the characteristic polynomials.

It is obvious that the structured flowgraphs become increasingly complex as composition, alternation, and iteration transformations are performed on them. Also apparent is the difficulty in directly comparing their characteristic polynomials. For example, looking at the characteristic polynomials [without the aid of the flowgraph diagrams] for flowgraphs in *Figure 2.4* and *Figure 2.5*, it is difficult to say whether the characteristic polynomial of flowgraph 4 is more complex than that of flowgraph 5 or vice versa.

Prior to coming up with a *mapping measure*[1], let us determine if characteristic polynomials obey either of the two desired orderings in discrete mathematics, viz., equivalence classes and partial orders. Since characteristic polynomials are computed from structured flowgraphs, the job of determining the orderings inherent in characteristic polynomials boils down to confirming the ordering inherent in structured flowgraphs.

---

[1]A measure that maps each characteristic polynomial with a real number in the set Ω.

Chapter 3.

TSAI'S POLYNOMIALS

3.1. DATA STRUCTURE COMPLEXITY:

Characteristic polynomials, discussed above, measure the source code complexity and in so doing try to especially weigh the characteristics associated with the control structures of a program. However, they do not take into account the complexity of data structures which, unquestionably, constitutes a major portion of a program's overall complexity.

A complexity measure of the data structure of a program, as presented here, is taken from Tsai, Lopez, Rodriguez, and Volovik[6]. Hereafter, this data structure complexity will be called *Tsai's data structure complexity measure* or briefly *Tsai's polynomial.* Both Tsai's data structure complexity measure and Cantone-Cimitile-Sansone's source code complexity measure yield similar-looking

[6]Tsai, et. al., op. cit.

polynomials. The similarity of these two measures is one reason behind the investigation of Tsai's data structure complexity measure. Besides the similarity of these polynomials, the real motivation in studying Tsai's data structure complexity along with characteristic polynomial has to do with the desire to discover if these two complexity measures could somehow be combined in arriving at an *extended complexity measure.*

It is a well known fact that complexity measures research has not focused much attention on studying data structure complexity. Equally well known is the fact that software complexity is a function of several factors and data structure is one of them. Note additionally that the data structure of a software system is available from the beginning of software life cycle and hence, it provides a way to estimate and predict software characteristics earlier than other measures based on source code alone. Further, data complexity can be thought of as a harbinger of the software system complexity since the structure of a program is known to follow the structure of its data very closely.

Method of computing Tsai's measure is discussed below. Instead of computing Tsai's measure for a single involved data structure, we begin here with a very simple data definition and add more data items making the data structure progressively complex as we did in the case of characteristic polynomials. Tsai's data structure complexity measure is computed at each stage.

3.2. TSAI'S DATA STRUCTURE COMPLEXITY MEASURE:

The algorithm for computing Tsai's polynomial requires that the data structure in any programming language satisfies a data structure description language. This data structure description language is quite general in that it is applicable to the data structures of almost all structured programming languages.

3.2.1. A Data Structure Description Language:

The data structure description language as outlined by Tsai, et. al., assumes that any given data structure $\mathcal{D}$ is of the form

$$\mathcal{D} :: f\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}, \qquad n \geq 1.$$

where f is a data structure building operator, $\mathcal{D}_i$'s are instances of references to the definitions of data structures used to build $\mathcal{D}$, and n is a number of these instances. Note that $\mathcal{D}_i$ can be $\mathcal{D}$ itself, i.e., recursive (self-referential) definitions are allowed. Also note that $\mathcal{D}_i$'s can be atomic data structures, i.e., the ones which are predefined in a language and do not have an explicit definition of the form above.

A finite set Q of data structure definitions can then be written as

$$Q = \{\mathcal{D}_i \mid \mathcal{D}_i :: f\{\mathcal{D}_{j1}, \ldots, \mathcal{D}_{jn}\}\}, \quad j1 \leq jn$$

where f's are data structure building operators and $\mathcal{D}_{ji}$'s are instances of references to data structures.

A data structure definition for which Tsai's polynomial is to be constructed is expected to follow the data structure description language.

3.2.2. Step 1 - Building a Directed Multigraph:

After verifying that the data structure definition of a program follows the data structure description language, a *directed multigraph* is constructed. There is one directed multigraph for each distinct data structure Q in a given program. To build the multigraph, create a unique node d_i for each data structure definition $\mathcal{D}_i$ and a unique node for each kind of atomic data structure referenced by some data structure definition $\mathcal{D}_i$ in Q. For each data structure definition $\mathcal{D}_i$ of the form $\mathcal{D}_i :: f(\mathcal{D}_{j1}, ..., \mathcal{D}_{jn})$ create an edge directed from d_i to d_{ji} for each instance of $\mathcal{D}_{ji}$ referenced in the definition of $\mathcal{D}_i$, reflecting the fact that $\mathcal{D}_i$ is dependent on the definitions of $\mathcal{D}_{ji}$'s when $\mathcal{D}_{ji}$'s are explicitly defined in Q, or on the atomic data structures when $\mathcal{D}_{ji}$'s are atomic. Call this multigraph G.

3.2.3. Step 2 - Checking for Completeness:

The directed multigraph as drawn is checked for *completeness*. A set of definitions is *complete* if and only if there exists a path from any node corresponding to a non-atomic definition to some node corresponding to an atomic definition. Note that checking for completeness can be performed even before building the multigraph. However, the directed multigraph visually facilitates the check.

3.2.4. Step 3 - Eliminating Auxiliary Definitions:

For each node in the directed multigraph, define the *degree of a node* as

degree(d_i) = in-degree(d_i) + out-degree(d_i)

where in-degree(d_i) is the total number of edges going from some node [including d_i] to d_i, and out-degree(d_i) is the total number of edges going from node d_i to some node [including d_i].

Auxiliary nodes in a directed multigraph are those nodes of degree at least two which have exactly one in-edge [or exactly one out-edge] where the only in-edge is different from every out-edge and no out-edge forms a loop of length two with the in-edge, [or the only out-edge is different from every in-edge and no in-edge forms a loop of length two with the out-edge]. All such auxiliary nodes are eliminated.

This amounts to eliminating a node if there is one in-edge to the node and one out-edge from the node with the condition that the out-edge does not point back to node's parent.

After deleting such nodes, replace each pair of edges [the in-edge and the out-edge of the deleted node] by just one edge with the same source as the in-edge and the same destination as the out-edge.

3.2.5. Step 4 - Splitting into Strongly Connected Components:

Two nodes d_i and d_j in a directed multigraph are said to be *strongly connected* if and only if there are paths in a directed multigraph from d_i to d_j and from d_j to d_i. Any node is considered to be strongly connected with itself even when there is no path from the node to itself. Define a strongly connected component of a directed multigraph to be a subgraph $\mathcal{K}$ of $\mathcal{G}$, such that each pair of nodes in $\mathcal{K}$ is strongly connected via some path in $\mathcal{K}$. Notice that $\mathcal{K}$ is *maximal* in a sense that no node in $\mathcal{G}$ which is not in $\mathcal{K}$ already is strongly

connected to any of the nodes in $\mathcal{K}$. Further, a binary relation of being strongly connected partitions the set of nodes of $\mathcal{G}$ into a set of equivalence classes.

3.2.6. Step 5 - Deriving Self-Complexity Monomials:

The self-complexity monomial of a strongly connected component takes into account the number of nodes and edges within the strongly connected component weighing it by the number of simple circular paths within the component.

The self-complexity monomial of a strongly connected component $S(d_i)$ is computed using

$$S(\mathcal{K}) = S(d_i) = (\mathcal{V} + \mathcal{E})\, c^{\mathcal{L}}$$

where $\mathcal{V}$ is the total number of nodes in a component $\mathcal{K}$, $\mathcal{E}$ is the total number of edges in the component $\mathcal{K}$ to which d_i belongs, and $\mathcal{L}$ is the total number of simple circular paths [circular paths which do not include proper circular sub-paths in them] in the component $\mathcal{K}$, and c is a variable.

Each node within a strongly connected component is assigned the same self-complexity monomial. This monomial $S(d_i)$ is a measure of the self-complexity of any of the nodes in the component $\mathcal{K}$.

3.2.7. Step 6 - Deriving Data Structure Complexity Polynomials:

Having computed the self-complexity monomials for each of the strongly connected components, data structure complexity polynomial, denoted $C(\mathcal{K})$, is computed by aggregating the self-complexity monomials.

Data structure complexity polynomial for the strongly connected component $\mathcal{K}$ is then defined as

$$C(\mathcal{K}) = \Sigma[C(\mathcal{K}_i)] + S(\mathcal{K})$$

where the sum is taken over the set of $\mathcal{K}_i$'s such that $\mathcal{K}_i \mathcal{R} \mathcal{K}$ and $\mathcal{K}_i$ is adjacent to $\mathcal{K}_i$ i.e., there is an edge connecting some node in $\mathcal{K}_i$ to some node in $\mathcal{K}_i$ and each summand $C(\mathcal{K}_i)$ is present in the sum as many times as there are different edges directly connecting $\mathcal{K}_i$ to $\mathcal{K}_i$.

Complexity of a node is defined as the complexity of the unique strongly connected component to which this node belongs. Tsai's data structure complexity measure of a data structure is then just a complexity of a node corresponding to this data structure.

AN EXAMPLE:

The next several pages contain data structure definitions, their corresponding multigraphs, and their self-complexity monomials and data structure complexity polynomials. The first figure shows the initial data structure that we started with and the others display the incremental addition to their predecessors. This was done with the conviction that it will indicate the way in which the complexity changes as the data structure definition grows more complex. Moreover, each modification of the data structure can be described in terms of the directed multigraph transformations [Appendix A; Section A.7].

Most of the notations in the figures have been described above. In the multigraphs broken lines encircle each strongly connected component. In all

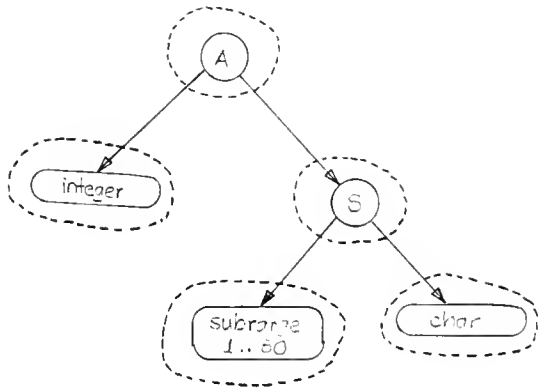

Figure 3.1

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.1.

Data Structure:

```
Type
  A = RECORD
    i1 :  integer;
    S :  array [1 .. 80] of char;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{A}) = S(\text{S}) = 1$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{S}) = C(\text{subrange}) + C(\text{char}) + S(\text{S}) = 3$
$C(\text{A}) = C(\text{integer}) + C(\text{S}) + S(\text{A}) = 5$

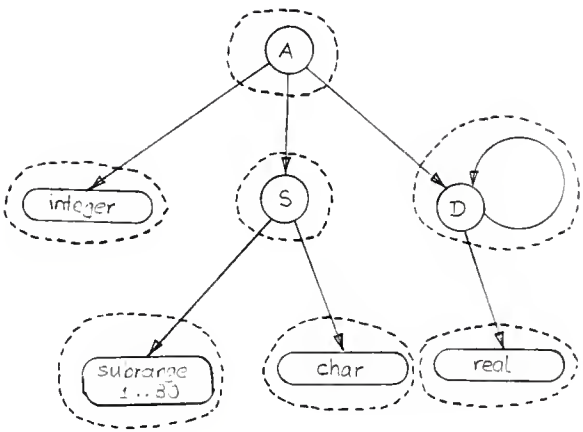

Figure 3.2

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.2.

Data Structure:

```
Type
  A = RECORD
    i1 : integer;
    S  : array [1 .. 80] of char;
    f1 : ^D;
  END;

  D = RECORD
    f1 : ^D;
    r1 : real;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{real}) = S(\text{A}) = S(\text{S}) = 1$
$S(\text{D}) = (1+1)\ X^1 = 2X$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{real}) = S(\text{real}) = 1$
$C(\text{S}) = C(\text{subrange}) + C(\text{char}) + S(\text{S}) = 3$
$C(\text{D}) = C(\text{real}) + S(\text{D}) = 1 + 2X$
$C(\text{A}) = C(\text{integer}) + C(\text{S}) + C(\text{D}) + S(\text{A}) = 1 + 3 + (1+2X) + 1 = 6 + 2X$

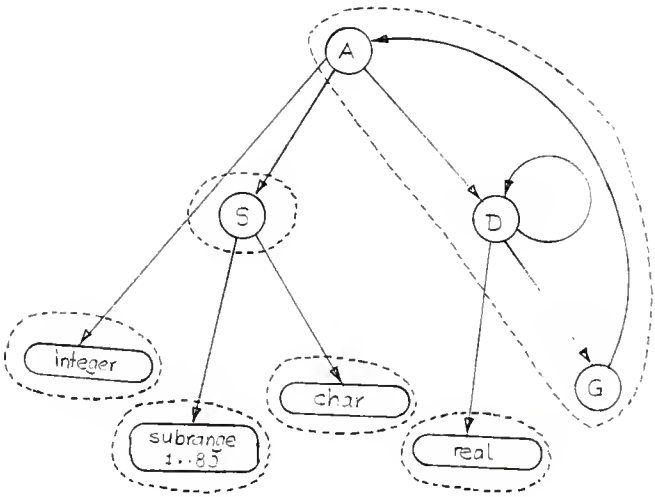

Figure 3.3

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.3.

Data Structure:

```
Type
  A = RECORD
    i1 : integer;
    S : array [1 .. 80] of char;
    f1 : ^D;
  END;

  D = RECORD
    f1 : ^D;
    r1 : real;
    f2 : ^G;
  END;

  G = RECORD
    f1 : ^A;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{real}) = S(S) = 1$
$S(A) = S(D) = S(G) = (3+4)\ X^2 = 7X^2$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{real}) = S(\text{real}) = 1$
$C(S) = C(\text{subrange}) + C(\text{char}) + S(S) = 3$
$C(A) = C(D) = C(G) = C(\text{integer}) + C(S) + C(\text{real}) + S(A)$
$= 1 + 3 + 1 + 7X^2 = 5 + 7X^2$

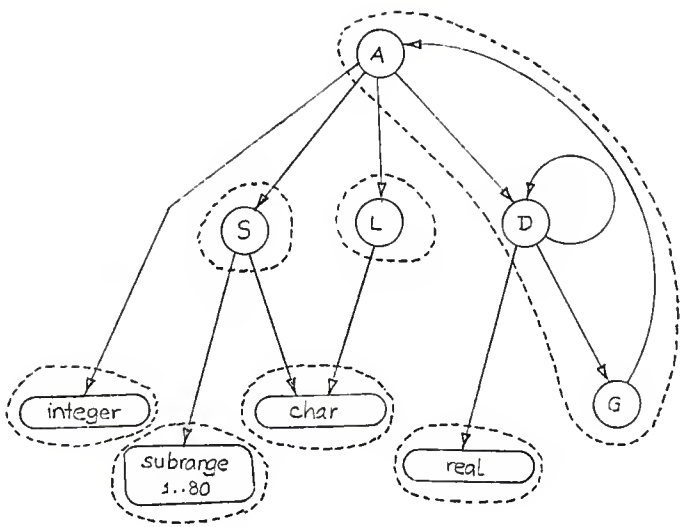

Figure 3.4

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.4.

Data Structure:

```
Type
  A = RECORD
    i1 :  integer;
    S  :  array [1 .. 80] of char;
    f1 :  ^D;
    f2 :  ^L;
  END;

  L = RECORD
    c1 :  char;
  END;

  D = RECORD
    f1 :  ^D;
    r1 :  real;
    f2 :  ^G;
  END;

  G = RECORD
    f1 :  ^A;
  END;
```

Note: The node L can be eliminated as an auxiliary definition and the edges leading to and emanating from L can be replaced by an edge from node A to node char. See Figure 2.5.

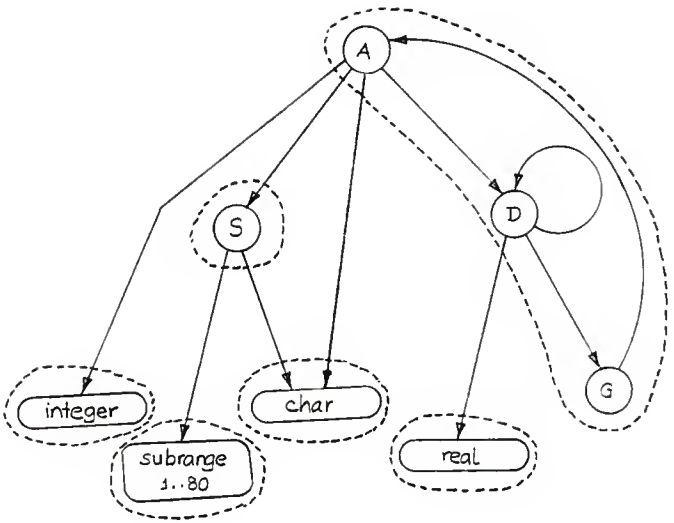

Figure 3.5

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.5.

Data Structure:

```
Type
  A = RECORD
    i1 :  integer;
    S  :  array [1 .. 80] of char;
    f1 :  ^D;
    f2 :  ^L;
  END;

  L = RECORD
    c1 :  char;
  END;

  D = RECORD
    f1 :  ^D;
    r1 :  real;
    f2 :  ^G;
  END;

  G = RECORD
    f1 :  ^A;
  END;
```

Self-Complexity Monomials:

S(integer) = S(subrange) = S(char) = S(real) = S(S) = 1
S(A) = S(D) = S(G) = (3+4) X^2 = $7X^2$

Tsai's Complexity Polynomials:

C(integer) = S(integer) = 1
C(subrange) = S(subrange) = 1
C(char) = S(char) = 1
C(real) = S(real) = 1
C(S) = C(subrange) + C(char) + S(S) = 3
C(A) = C(D) = C(G) = C(integer) + C(S) + C(char) + C(real) + S(A)
= 1 + 3 + 1 + 1 + $7X^2$ = 6 + $7X^2$

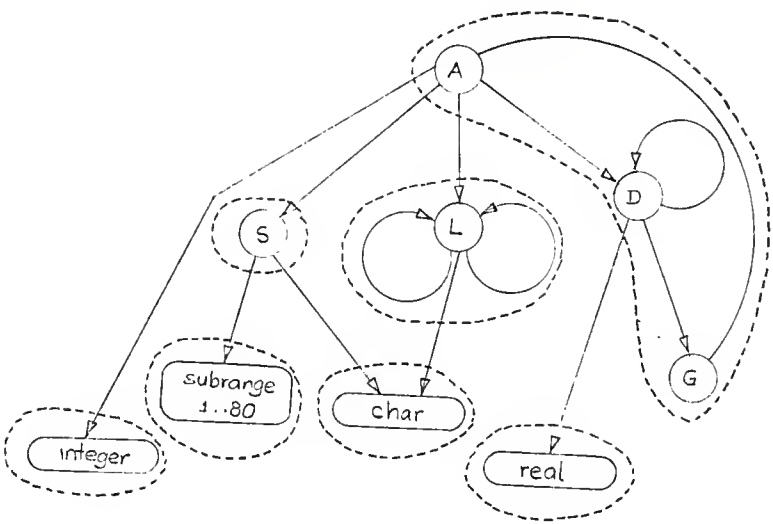

Figure 3.6

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.6.

Data Structure:

```
Type
  A = RECORD                        L = RECORD
    i1 :  integer;                    f1 :  ^L;
    S  :  array [1 .. 80] of char;    f2 :  ^L;
    f1 :  ^D;                         c1 :  char;
    f2 :  ^L;                       END;
  END;

  D = RECORD
    f1 :  ^D;
    r1 :  real;
    f2 :  ^G;
  END;

  G = RECORD
    f1 :  ^A;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{real}) = S(S) = 1$
$S(L) = (1+2)X^2 = 3X^2$
$S(A) = S(D) = S(G) = (3+4)\,X^2 = 7X^2$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{real}) = S(\text{real}) = 1$
$C(S) = C(\text{subrange}) + C(\text{char}) + S(S) = 3$
$C(L) = C(\text{char}) + S(L) = 1+3X^2$
$C(A) = C(D) = C(G) = C(\text{integer}) + C(S) + C(L) + C(\text{real}) + S(A)$
$= 1 + 3 + (1+3X^2) + 1 + 7X^2 = 6 + 10X^2$

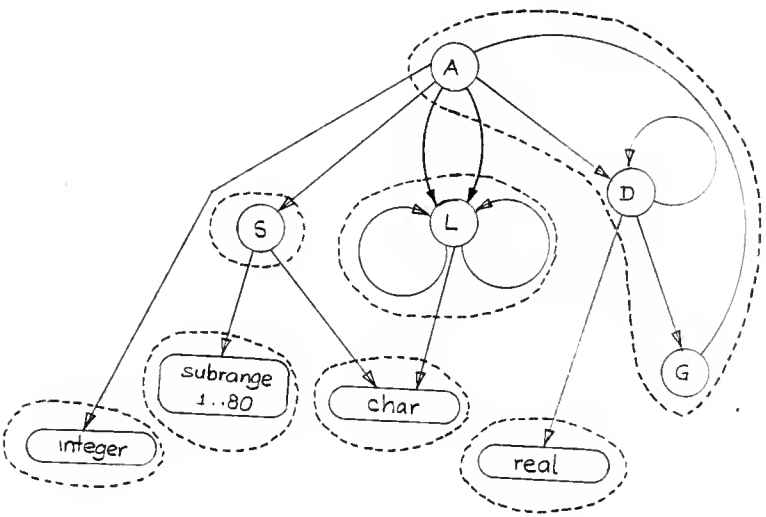

Figure 3.7

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.7.

Data Structure:

```
Type
  A = RECORD                          L = RECORD
    i1 : integer;                       f1 : ^L;
    S  : array [1 .. 80] of char;       f2 : ^L;
    f1 : ^D;                            c1 : char;
    f2 : ^L;                        END;
    f3 : ^L;
  END;

  D = RECORD
    f1 : ^D;
    r1 : real;
    f2 : ^G;
  END;

  G = RECORD
    f1 : ^A;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{real}) = S(S) = 1$
$S(L) = (1+2)X^2 = 3X^2$
$S(A) = S(D) = S(G) = (3+4)\,X^2 = 7X^2$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{real}) = S(\text{real}) = 1$
$C(S) = C(\text{subrange}) + C(\text{char}) + S(S) = 3$
$C(L) = C(\text{char}) + S(L) = 1+3X^2$
$C(A) = C(D) = C(G) = C(\text{integer}) + C(S) + 2 * C(L) + C(\text{real}) + S(A)$
$= 1 + 3 + 2 * (1+3X^2) + 1 + 7X^2 = 7 + 13X^2$

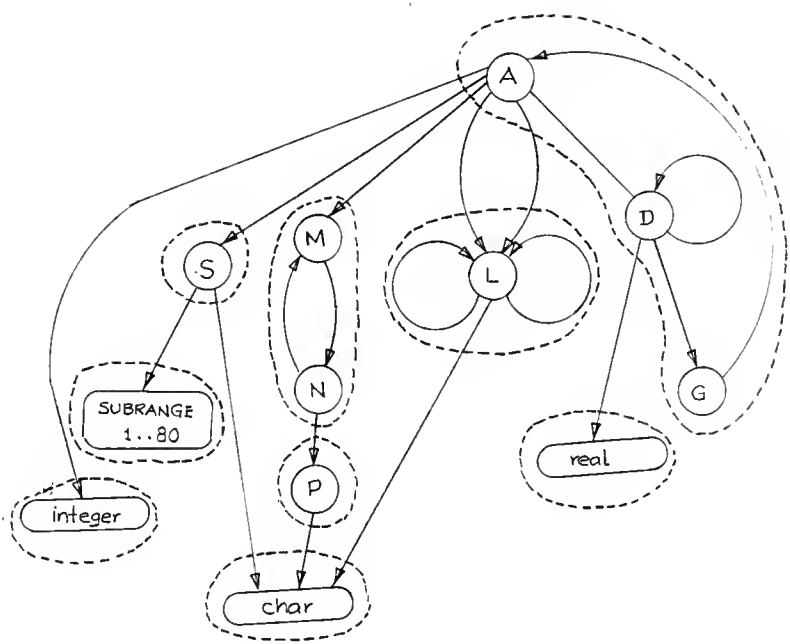

Figure 3.8

DATA STRUCTURE AND TSAI'S POLYNOMIALS FOR FIGURE 3.8.

Data Structure:

```
Type
  A = RECORD
    i1 :  integer;
    S  :  array [1 .. 80] of char;
    f1 :  ^D;
    f2 :  ^L;
    f3 :  ^L;
    f4 :  ^M;
  END;

  D = RECORD
    f1 :  ^D;
    r1 :  real;
    f2 :  ^G;
  END;

  L = RECORD
    f1 :  ^L;
    f2 :  ^L;
    c1 :  char;
  END;

  N = RECORD
    f1 :  ^M;
    p  :  set of char;
  END;

  G = RECORD
    f1 :  ^A;
  END;
```

Self-Complexity Monomials:

$S(\text{integer}) = S(\text{subrange}) = S(\text{char}) = S(\text{real}) = S(S) = S(P) = 1$
$S(L) = (1+2)X^2 = 3X^2$
$S(M) = S(N) = (2+2)X = 4X$
$S(A) = S(D) = S(G) = (3+4)\,X^2 = 7X^2$

Tsai's Complexity Polynomials:

$C(\text{integer}) = S(\text{integer}) = 1$
$C(\text{subrange}) = S(\text{subrange}) = 1$
$C(\text{char}) = S(\text{char}) = 1$
$C(\text{real}) = S(\text{real}) = 1$
$C(S) = C(\text{subrange}) + C(\text{char}) + S(S) = 3$
$C(L) = C(\text{char}) + S(L) = 1+3X^2$
$C(P) = S(P) = 1$
$C(M) = C(N) = C(P) + S(M) = 1+4X$
$C(A) = C(D) = C(G)$
$= C(\text{integer}) + C(S) + C(M) + 2 * C(L) + S(A)$
$= 1 + 3 + (1+4X) + 2 * (1+3X^2) + 7X^2 = 7 + 4X + 13X^2$

the following diagrams, $C(\mathcal{A})$ gives the data structure complexity of the component that contains the node $\mathcal{A}$ [also nodes $\mathcal{D}$ and $\mathcal{G}$] as well as the data structure complexity of the entire data structure definition.

Note that in the multigraph in *Figure 3.4*, node $\mathcal{L}$ will be removed in the process of deleting auxiliary definitions, and replaced by an edge from node $\mathcal{A}$ to node "Char". Consequently, $C(\mathcal{A})$ yields $6+7x^2$ rather than $7+7x^2$.

3.3. PROPERTIES OF TSAI'S POLYNOMIALS:

This measure quantifies the structural complexity of data, emphasizing the dynamic part of a data structure over the static part. The number of data elements in the data structure has only a small influence on the complexity measure. It measures the structure of data and not the size of it.

The measure, in the words of Tsai, et. al., is intuitive. Every reference of one definition to another definition (represented by an edge between two nodes), and every instance of such definitions (represented by a node) increases the measured complexity of data by increasing the coefficients of the resulting polynomials. Circular references contribute much more to the measure by also affecting the order of magnitude of the complexity measure.

This measure is consistent, that is, if data structure x is a substructure of a data structure y, then complexity(x) $\leq$ complexity(y). The measure tolerates incomplete information since it is possible to compute the measure at the very start of software design when not all the decisions have been made. Additionally, this measure is insensitive to language-specific details to some degree.

Chapter 4.

A MAPPING MEASURE

4.1. A PROCESS OF ABSTRACTION:

It is well known that most complexity measures attempt to arrive at a real number that represents the complexity of a program in whatever sense they are measured. McCabe's and Halstead's measures are two prominent examples. In studying complexity, an abstraction that generalizes programs is devised as the first step. Complexity measures may be computed on the basis of such an abstraction. For example, the abstraction on which McCabe bases his cyclomatic number is none other than the familiar flowgraphs. However, the interest is in producing measures that either map the abstraction directly or indirectly onto real numbers. Nevertheless, such mapped real numbers do not carry much significance unless they obey some kind of mathematical ordering such as *preorder, partial order,* and *linear order. Figure 4.1*

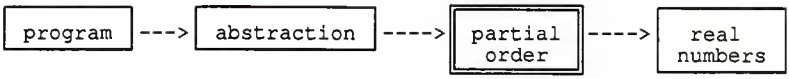

Figure 4.1.

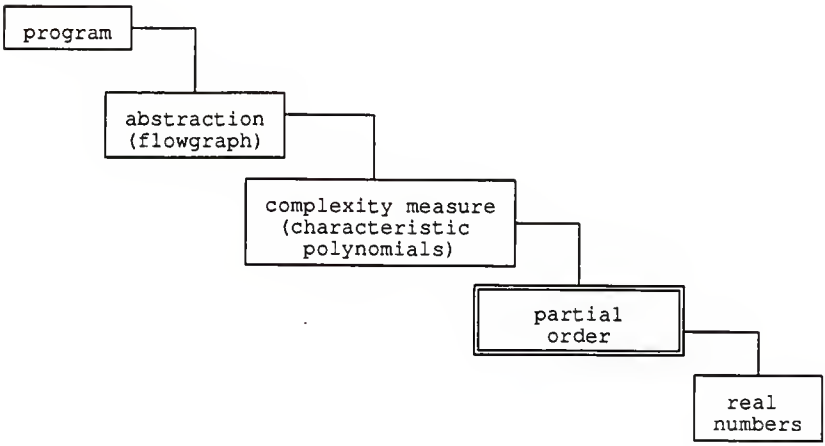

Figure 4.2.

depicts one desired mapping of the abstraction onto real numbers wherein the mapping mechanism preserves partial order.

Orderings of measures make them *comparable*. In case of linear orders, every pair of elements is comparable. However, linear orders are too restrictive to be of much value. Partial orders facilitate comparison of two elements. However, not every pair of elements of a partially ordered set is comparable. Preordering is the least restrictive of the three requiring the relation to be reflexive and transitive.

We started with programs that contained no statements that may have caused them to be unstructured. Secondly, we took the usual route in abstracting programs into flowgraphs. Since the programs are structured the flowgraphs are also structured. We will show later that the set of structured flowgraphs and a relation $\mathcal{F}$ is a partial order [see Chapter 5]. However, characteristic polynomials, which are computed for structured flowgraphs, are complex polynomials and are not members of the set Ω. As noted above, this causes difficulties in directly comparing characteristic polynomials. Therefore, in this study we will extend the manner in which an abstraction of a program is mapped to real numbers by incorporating a *mapping measure* [*Figure 4.2*].

EXAMPLE CONTINUED:

The characteristic polynomials computed for the illustrative flowgraphs in Chapter 2 were found to be difficult to compare directly. In order to enable comparisons between characteristic polynomials, a few simple measures were devised that map the characteristic polynomial expressions to real

numbers. It will be shown later that only one of the four simple mapping measures [$\mathcal{MM}$] constructed in this study preserve the ordering.

The four measures provided in *Table 4.1* map the polynomials to real numbers. Furnishing explanations of the various measures seems to result in reasonable, albeit crude, descriptions.

Σa_i as was indicated earlier, the first measure, namely, Σa_i is the number of exemplary executions. Σa_i is nothing but the *total number of paths* [from the start node to the terminal node] of the digraph.

$\Sigma \alpha_i$ variable c is significant only when there are cyclic control structures in a flowgraph. Hence, variable c along with its exponent indicates the nestedness of a flowgraph. Perhaps, this sum could be treated as the *crude sum of nestedness*. Note that very many different polynomials can produce this real number and, therefore, it is a rather poor mapping measure.

$\Sigma a_i(\alpha_i)$ It is simple to perceive that perhaps $\Sigma \alpha_i$ must be considered in tandem with Σa_i as a mapping measure. This measure is the *number of paths weighted by its nestedness.*

$\Sigma a_i(\alpha_i+1)$ Nearly the same as the measure above with the exception that the exponent of the variable is incremented by unity before using it as a weight to the number of paths.

As noted earlier, the flowgraphs illustrated in Chapter 2 were generated utilizing the flowgraph transformations and, clearly, they were increasingly complex. It is not clear, however, whether the characteristic polynomials built

Characteristic Polynomial	(1) Σa_i	(2) $\Sigma \alpha_i$	(3) $\Sigma a_i(\alpha_i)$	(4) $\Sigma a_i(\alpha_i+1)$
1. $4c$	4	1	4	8
2. $5c$	5	1	5	10
3. $6c^2 + 2c$	8	3	14	20
4. $6c^2 + 2c + 1$	9	3	14	23
5. $6c^2 + 4c$	10	3	16	26
6. $10c^2$	10	2	20	30
7. $4c^3 + 8c^2$	12	5	28	40
8. $8c^3 + 6c^2$	14	5	36	50
9. $8c^4 + 6c^3$	14	7	50	64

Table 4.1. Mapping Measures for Characteristic Polynomials.

for the increasingly complex flowgraphs demonstrate any such gain in source code complexity. That is, it is not always easy to compare two characteristic polynomials--without the aid of related flowgraphs--generated for transformed flowgraphs and remark if one characteristic polynomial is more complex than the other.

It is for this reason why an attempt was made to find measures that would map characteristic polynomials to real numbers and hence facilitate comparison. The anticipation is that any such measure would map increasingly complex characteristic polynomials, due to increasingly complex flowgraphs, onto increasingly larger real numbers. It is important to establish that the mapping also *preserves partial order* if any comparison of the resulting real numbers is desired. Indeed, one of the goals of the software complexity measures research is to find functions [measures] that preserve these orders. Therefore, it would be gratifying to see that more complex flowgraphs are always mapped with correspondingly larger real numbers and, at the same time, the mapping measures preserve partial orders.

Table 4.2 presents a comparative picture of the characteristic polynomials computed for the increasingly complex structured flowgraphs [given in *Figure 2.1* through *Figure 2.9*] in terms of the mapping measures [furnished in *Table 4.1*]. Note that it is useless to compare mapping measures (1) and (2) on the polynomials. Instead, they are combined together to check how they both map the polynomial expressions[7].

[7]This comparison could in turn be performed in terms of vectors. For example, comparing characteristic polynomials 1 and 2 can be written as [2,1] < [3,1] instead of 2 < 3 & 1 = 1, etc.

Characteristic Polynomials	COMPARISONS		
	(1) & (2) $\Sigma a_i \leq \Sigma b_i$ & $\Sigma \alpha_i \leq \Sigma \beta_i$	(3) $\Sigma a_i \alpha_i \leq \Sigma b_i \beta_i$	(4) $\Sigma a_i(\alpha_i+1) \leq \Sigma b_i(\beta_i+1)$
1 & 2. $4c$ vs. $5c$	$4 < 5$ & $1 = 1$	$4 < 5$	$8 < 10$
2 & 3. $5c$ vs. $6c^2+2c$	$5 < 8$ & $1 < 3$	$5 < 14$	$10 < 20$
3 & 4. $6c^2+2c$ vs. $6c^2+2c+1$	$8 < 9$ & $3 = 3$	$14 = 14$	$20 < 23$
4 & 5. $6c^2+2c+1$ vs. $6c^2+4c$	$9 < 10$ & $3 = 3$	$14 < 16$	$23 < 26$
5 & 6. $6c^2+4c$ vs. $10c^2$	$10 = 10$ & $3 \nleq 2$	$16 < 20$	$26 < 30$
6 & 7. $10c^2$ vs. $4c^3 + 8c^2$	$10 < 12$ & $2 \nleq 5$	$20 < 28$	$30 < 40$
7 & 8. $4c^3+8c^2$ vs. $8c^3+6c^2$	$12 < 14$ & $5 = 5$	$28 < 36$	$40 < 50$
8 & 9. $8c^3+6c^2$ vs. $8c^4+6c^3$	$14 = 14$ & $5 < 7$	$36 < 50$	$50 < 64$

[a_i and α_i refer to the coefficient i and exponent i of the first characteristic polynomial and b_i and β_i refer to the coefficient i and exponent i of the second characteristic polynomial of each comparison.]

Table 4.2. Comparisons of Characteristic Polynomials.

Overlooking preservation of partial orders for the time being, it is apparent that the combined mapping measure of (1) and (2) as well as (3) do not consistently produce larger real numbers for more complex characteristic polynomials. [Compare the characteristic polynomials 5 and 6 as well as 8 and 9 with respect to the combined measure (1) & (2) and the polynomials 3 and 4 with respect to the mapping measure (3).] Mapping measure (4), on the other hand seems to achieve the mapping properly. The fact that this mapping measure does map characteristic polynomials of more complex flowgraphs onto larger real numbers will be mathematically established later. Additionally, it preserves the partial order as well, as will be established below.

Chapter 5.

EQUIVALENCE CLASSES, PARTIAL ORDERS, AND CHARACTERISTIC POLYNOMIALS

Definition 5.1: Let $\mathcal{A}$ be a set of structured flowgraphs. For any two structured flowgraphs, $\mathcal{G}_1 \in \mathcal{A}$ and $\mathcal{G}_2 \in \mathcal{A}$ there exists a relation $\mathcal{G}_1 \mathcal{F} \mathcal{G}_2$, if $\mathcal{G}_2$ can be obtained from $\mathcal{G}_1$ by a finite, $k \geq 0$, sequence of applications of structured flowgraph transformations, viz., composition, alternation, and iteration transformations.

5.1. STRUCTURED FLOWGRAPH TRANSFORMATIONS AND EQUIVALENCE CLASSES:

Given characteristic polynomials, the consideration is regarding the existence of any equivalence classes and/or partial orders related to characteristic polynomials. The outcome of structured flowgraph transformations on equivalence classes is examined below.

Starting with a structured flowgraph and applying the composition transformation, it is easy to find that the transformed flowgraph does not belong in the same equivalence class. Composition destroys the symmetry property. Given two structured flowgraphs $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$, composition results in a new structured flowgraph, $G_3 = (N_1 \cup N_2, E_1 \cup E_2, s_1, t_2)$ with the restriction that the exit node of G_1 and the entry node of G_2 become a single new junction node. Although both G_1 and G_2 comply with reflexive and transitive properties, they do not satisfy the property of symmetry since while an edge such as $(s_2, x) \in G_2$ where $x \in N_2$, $(s_2, x) \notin G_1$. We can, therefore, conclude that flowgraph transformations do not result in preserving equivalence ordering. In other words, the relation $\mathcal{F}$ as defined above is not an equivalence ordering.

5.2. STRUCTURED FLOWGRAPH TRANSFORMATIONS AND PARTIAL ORDERS:

Though the relation $\mathcal{F}$ is not an equivalence ordering, it will be shown below that it is a partial ordering. Proving $\mathcal{F}$ is a partial ordering involves showing that reflexivity, antisymmetry, and transitivity properties are still

met after any of the three possible transformations. Assume three structured flowgraphs $G_1 = (N_1, E_1, s_1, t_1)$, $G_2 = (N_2, E_2, s_2, t_2)$, and $G_3 = (N_3, E_3, s_3, t_3)$ where G_1 is combined with G_2 using composition, alternation, or iteration transformations resulting in a new structured flowgraph, $G_{1,2}$, and G_2 is combined with G_3 producing a flowgraph, $G_{2,3}$. Employing similar notations, combining G_{ij} with G_{kl} will be written $G_{ij,kl}$.

Clearly $G_1 \mathcal{F} G_1$ and $G_2 \mathcal{F} G_2$ immaterial of the application of composition, alternation, or iteration transformation. Therefore, reflexivity property holds.

With any of the three structured flowgraph transformations, unless when $k = 0$, $G_1 \mathcal{F} G_{1,2}$ but $G_{1,2} \mathcal{F} G_1$. When $k = 0$, $G_1 \mathcal{F} G_{1,2}$ and $G_{1,2} \mathcal{F} G_1$ implying that $G_1 = G_{1,2}$. Therefore, relation $\mathcal{F}$ is antisymmetric.

For observing transitivity property assume that the structured flowgraph G_1 is combined using any of the structured flowgraph transformations with G_2 producing $G_{1,2}$. Further assume that the structured flowgraphs $G_{1,2}$ and G_3 are combined producing a new flowgraph $G_{12,3}$. We have argued above that $G_1 \mathcal{F} G_{1,2}$ and $G_{1,2} \mathcal{F} G_{12,3}$ for any finite sequence of structured flowgraph transformations. It is easy to see that if G_1 can be transformed into $G_{1,2}$ using G_2 in some finite sequence of transformations and $G_{1,2}$ can similarly be transformed into $G_{12,3}$ using G_3 in some finite applications of transformations, then G_1 can be transformed into $G_{12,3}$ using G_2 and G_3 in some finite transformations. Therefore, $G_1 \mathcal{F} G_{12,3}$.

This implies that if $G_1 \mathcal{F} G_{12}$ and $G_{12} \mathcal{F} G_{123}$ then $G_1 \mathcal{F} G_{123}$ and hence $\mathcal{F}$ is transitive.

Since $\mathcal{F}$ is reflexive, antisymmetric, and transitive it is a partial order. In other words, structured flowgraph transformations preserve the partial order of the relation $\mathcal{F}$. Summarizing the results one can arrive at the following definition:

Definition 5.2: The relation $\mathcal{F}$ on a set of structured flowgraphs $\mathcal{A}$ is a partial order since $\mathcal{F}$ is reflexive, antisymmetric, and transitive. In other words, $(\mathcal{A},\mathcal{F})$ is a *poset.*

5.3 PROOF THAT CHARACTERISTIC POLYNOMIALS PRESERVE PARTIAL ORDER:

Definition 5.3: Assume a set of structured flowgraphs $\mathcal{A}$ that were generated using structured flowgraph transformations. Further, let $\mathcal{CP}$ be the set of characteristic polynomials computed for $\mathcal{A}$. Then, the relation $\mathcal{F}$ on $\mathcal{CP}$ is a partial ordering and $(\mathcal{CP},\mathcal{F})$ is a *poset.*

In order to prove that the characteristic polynomials preserve the ordering we would show that the characteristic polynomial of a flowgraph is contained in the characteristic polynomial of the transformed flowgraph. Let CP, be the characteristic polynomial of a flowgraph [the characteristic polynomial of the transformed flowgraph is denoted CP,]. Then, it can be written as:

$$CP_o = a_1 c^{\alpha_1} + a_2 c^{\alpha_2} + \ldots + a_k c^{\alpha_k} + \ldots + a_n c^{\alpha_n} \qquad (1)$$

$$= \sum_{i=1}^{n} a_i c^{\alpha_i} \qquad \text{where both } a \geq 0 \text{ and } \alpha \geq 0 \text{ and are both integers.}$$

COMPOSITION TRANSFORMATION:

By applying a composition transformation, a composite structure is obtained by sequentially combining two structured flowgraphs. The exit node of one structured flowgraph is made to coincide with the entry node of another structured flowgraph. Each of these structured flowgraphs has a characteristic polynomial of its own. The computation of characteristic polynomial measure dictates that the characteristic polynomial of the transformed flowgraph will be the product of the characteristic polynomials of the two separate structured flowgraphs. If the characteristic polynomials of the two individual structured flowgraphs are assumed to be

$$CP_1 = \sum_{i=1}^{n} a_i c^{\alpha_i} \qquad \text{and} \qquad (2)$$

$$CP_2 = \sum_{j=1}^{m} b_j c^{\beta_j} \qquad \text{where } a \geq 0,\ b \geq 0,\ \alpha \geq 0, \text{ and } \beta \geq 0 \text{ and are all integers.} \qquad (3)$$

then the characteristic polynomial of the composed flowgraph is:

$$CP_t = \left[\sum_{i=1}^{n} a_i c^{\alpha_i} \right] \cdot \left[\sum_{j=1}^{m} b_j c^{\beta_j} \right]. \qquad (4)$$

which is larger than *(1)*. Hence, $CP_o \leq_c CP_t$ that is, the characteristic polynomial of the initial flowgraph is contained in the characteristic polynomial of the transformed flowgraph.

ALTERNATION TRANSFORMATION:

Alternation transformation lets us combine two structured flowgraphs by permitting them to be successors of a common condition node and also letting them have a common exit node. Once again, assume that the two flowgraphs have the characteristic polynomials indicated in *(2)* and *(3)* respectively. Then, using the fact that the characteristic polynomial is given by the sum of the characteristic numbers of the structures and nodes which are contained in it, the new characteristic polynomial of the transformed flowgraph is given by:

$$CP_t = \sum_{i=1}^{n} a_i c^{\alpha_i} + \sum_{j=1}^{m} b_j c^{\beta_j} \qquad (5)$$

which implies that $CP_o \leq_c CP_t$.

ITERATION TRANSFORMATION:

The transformed structured flowgraph is organized such that an original structured flowgraph is placed within a *loop* with a condition node at its beginning or end to govern the loop's termination. Assuming that the characteristic polynomial of the initial flowgraph is given by *(2)*, the new characteristic polynomial is given by either:

$$CP_t = \left[\sum_{i=1}^{n} a_i c^{\alpha_i} \right] * c \qquad (6)$$

if it is a repeat-until structure, or by:

$$CP_t = [\ \sum_{i=1}^{n} a_i c^{\alpha_i} + 1\ ]\ ^{*}\ c \qquad (7)$$

if it is a while-do structure.

This indicates that $CP_o \leq_c CP_t$ meaning that the characteristic polynomial of the initial flowgraph is contained in the new characteristic polynomial following an iteration transformation.

5.4 PROOF THAT $\Sigma a_i(\alpha_i+1)$ PRESERVES THE PARTIAL ORDER:

In order to provide a proof that $\Sigma a_i(\alpha_i+1)$ preserves the order, we will borrow the results from the previous section. Showing that the mapping of the characteristic polynomial to real numbers using the mapping measure $\Sigma a_i(\alpha_i+1)$ preserves partial order is, again, a simple mathematical exercise. This exercise is also carried out in terms of composition, alternation, and iteration transformations.

As we know, the generalized characteristic polynomial for the original structured flowgraph is given by:

$$CP_1 = \sum_{i=1}^{n} a_i c^{\alpha_i} \qquad (2)$$

and that of another structured flowgraph with which the original flowgraph is composed is given by:

$$CP_2 = \sum_{j=1}^{m} b_j c^{\beta_j} \qquad \text{where } a \geq 0,\ b \geq 0,\ \alpha \geq 0, \text{ and } \beta \geq 0 \text{ and are all integers.} \qquad (3)$$

and the mapping measure of the original flowgraph is given by:

$$MM = \sum_{i=1}^{n} a_i (\alpha_i + 1) \qquad (8)$$

Following the composition of these two structured flowgraphs, the new characteristic polynomial becomes *(4)*, which is:

$$CP_t = \left[\sum_{i=1}^{n} a_i c^{\alpha_i} \right] \cdot \left[\sum_{j=1}^{m} b_j c^{\beta_j} \right]. \qquad (4)$$

This characteristic polynomial *(4)* can be shown to have the following mapping measure:

$$MM = \sum_{i=1}^{n} a_i \cdot \sum_{j=1}^{m} b_j \, (\alpha_i + \beta_j + 1) \qquad (9)$$

which is larger than the mapping measure for the original structured flowgraph as indicated in *(8)*.

In case of an alternation transformation, the new characteristic polynomial is given by *(5)* and its new mapping measure is given by:

$$MM = \sum_{i=1}^{n} a_i \, (\alpha_i + 1) + \sum_{j=1}^{m} b_j \, (\beta_j + 1) \qquad (10)$$

which is larger than *(8)* by its second term.

Iteration transformation also raises the size of the mapping measure. Note that an iteration transformation alters the characteristic polynomial to either *(6)* or *(7)*. The mapping measure corresponding to these new characteristic polynomial expressions are given by *(11)* and *(12)* respectively:

$$MM = \sum_{i=1}^{n} a_i \, (\alpha_i + 2) \qquad (11)$$

$$MM = \sum_{i=1}^{n} a_i \, (\alpha_i + 2) + 2 \qquad (12)$$

each of which is larger than the mapping measure of the original structured flowgraph *(8)*. Therefore, any increase in complexity of the original flowgraph

due to composition, alternation, and/or iteration transformations results in a larger real value and hence proves that the mapping measure *(8)* does preserve partial order.

5.5 SYNOPSIS:

We noted that if a structured flowgraph G_2 can be obtained from structured flowgraph G_1 using structured flowgraph transformations, then the set of structured flowgraphs and the relation $\mathcal{F}$ [see the start of Chapter 5 for a definition of $\mathcal{F}$] is a partial order.

Characteristic polynomials were then constructed for structured flowgraphs and were shown to preserve the order. Since the polynomials did not lend themselves to direct comparison, a mapping measure *(MM)* was devised. As we have shown above, the mapping measure, *MM*, preserved the partial order while facilitating direct comparison.

It must be noted that $MM(G_1)$ is comparable to $MM(G_2)$ or that CP_1 is comparable to CP_2 [assuming that CP_1 and CP_2 are the characteristic polynomials constructed for the structured flowgraphs G_1 and G_2, respectively] if and only if the structured flowgraph G_2 is constructed by applying structured flowgraph transformations on the structured flowgraph G_1. For two unrelated structured flowgraphs, G_1 and G_2, it is not possible to say if CP_1 and CP_2 are comparable.

In order to validate characteristic polynomials utilizing the software complexity measure validation paradigm given in Appendix B, let m be a [hypothetical] software complexity measure which is strictly monotone with

respect to the $\mathcal{F}$ partial order. Then if a and b are programs with structured flowgraphs G_1 and G_2, respectively, and $m(a) \leq m(b)$, then we cannot have that $\mathcal{MM}(G_1) > \mathcal{MM}(G_2)$. That is, characteristic polynomials, as a complexity measure, is strictly monotone with respect to the $\mathcal{F}$ preordering of all structured flowgraphs.

Chapter 6

EQUIVALENCE CLASSES, PARTIAL ORDERS, AND TSAI'S POLYNOMIALS

As was with characteristic polynomials, the usefulness of the data structure complexity polynomial depends on whether it preserves any mathematical ordering. In order to check the existence of equivalence classes and partial orders in Tsai's polynomials, we can once again begin with the definition of a relation [see Definition 6.1 below] similar to the one employed for flowgraphs in Chapter 5. Note that Definition 6.1 is expressed in terms of multigraph transformations discussed in Appendix A. It may be recalled

that Definition 5.1 was stated in terms of structured flowgraph transformations.

Definition 6.1: Let $\mathcal{A}$ be a set of directed multigraphs. For any two multigraphs, $\mathcal{M}_i \in \mathcal{A}$ and $\mathcal{M}_j \in \mathcal{A}$ there exist a relation $\mathcal{M}_i \; \mathcal{G} \; \mathcal{M}_j$ if $\mathcal{M}_i$ can be obtained from $\mathcal{M}_i$ by a finite, $k \geq 0$, sequence of applications of directed multigraph transformations, viz., node addition, node replication, and edge addition transformations.

6.1 MULTIGRAPH TRANSFORMATIONS AND EQUIVALENCE CLASSES:

Node addition transformations are possible only from non-atomic nodes. The added node may be either atomic or non-atomic. However, in order to be complete, if the added node is non-atomic path(s) must be provided from the new non-atomic node to atomic node(s). In any case, a node addition transformation modifies the directed multigraph and the new multigraph no longer belongs in the same equivalence class.

A node replication transformation applied on a multigraph generates a directed multigraph with a new *auxiliary* definition contained in it. As noted from the method of computing Tsai's polynomial we know that any such node(s) do not contribute to the data structure complexity and can effectively be removed. Hence, the original directed multigraph and the new multigraph with one or more node replication transformations applied on it are equivalent. Therefore, node replication transformation retains the directed multigraph in the same equivalence class.

The effect of edge addition transformation on directed multigraphs is more complex. Since a path from an atomic definition to a non-atomic definition will have no correspondence to a possible data structure definition, edge additions will be restricted between non-atomic nodes or from non-atomic nodes to atomic nodes.

Edge addition transformations can be distinguished into *forward edge addition transformation* and *backward edge addition transformation*. In case of forward edge additions, an edge which originates in a non-atomic node points at an atomic node or the edge originates in a node which must have been declared before the node which the edge is pointing to. On the other hand, backward edge addition transformation is one where the edge originates at a node whose declaration was made later than the node it points at. In essence, backward edge addition transformation causes a circular definition of data. Note that there can not be a backward edge addition transformation from an atomic node to any other node.

Edge additions between non-atomic nodes may be either forward edge addition transformation or backward edge addition transformation whereas edge addition transformation from a non-atomic node to an atomic node is always forward edge addition. Forward edge addition does not change the number of strongly connected components but it increases references [non-cyclic] to other data structures. In the case of backward edge addition, new cyclic references are created. This may reduce the number of strongly connected components in the entire data structure. Note that the reduction in the number of strongly connected components is due to the merger of two

or more previously strongly connected components. In either case, edge addition transformations create multigraphs which do not belong in the same equivalence class.

The relation $\mathcal{G}$, as defined in Definition 6.1, hence is not an equivalence ordering. Node replication transformation places the new multigraph in the same equivalence class and satisfies the reflexivity, symmetry, and transitivity properties. However, the other two flowgraph transformations do not. This result is quite the same as in the case of transformations applied on flowgraphs.

6.2 MULTIGRAPH TRANSFORMATIONS AND PARTIAL ORDERS:

The relation $\mathcal{G}$ is, nevertheless, a partial ordering. The results obtained here, again, are comparable to the preservation of partial order by structured flowgraph transformations applied on flowgraphs. Assume three directed multigraphs $\mathcal{M}_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathbf{f}_1)$, $\mathcal{M}_2 = (\mathcal{V}_2, \mathcal{E}_2, \mathbf{f}_2)$, and $\mathcal{M}_3 = (\mathcal{V}_3, \mathcal{E}_3, \mathbf{f}_3)$ where multigraph $\mathcal{M}_1$ is transformed into $\mathcal{M}_2$ using any of the directed multigraph transformations and similarly multigraph $\mathcal{M}_3$ from $\mathcal{M}_2$.

Clearly $\mathcal{M}_1 \; \mathcal{G} \; \mathcal{M}_1$, $\mathcal{M}_2 \; \mathcal{G} \; \mathcal{M}_2$ and $\mathcal{M}_3 \; \mathcal{G} \; \mathcal{M}_3$ immaterial of the application of node addition, node replication, or edge addition transformations. Therefore, reflexivity property holds.

Node replication transformation was shown to retain the multigraphs in the same equivalence class, i.e., $\mathcal{M}_1$ is the same as $\mathcal{M}_2$ since the replicated node will be removed. Hence, $\mathcal{M}_1 = \mathcal{M}_2$, which implies that node replication is antisymmetric. With any of the other two multigraph transformations, unless

when $k = 0$, $\mathcal{M}_1 \mathcal{G} \mathcal{M}_2$ but $\mathcal{M}_2 \not\mathcal{G} \mathcal{M}_1$. When $k = 0$, $\mathcal{M}_1$ and $\mathcal{M}_2$ are one and the same. Therefore, relation $\mathcal{G}$ is antisymmetric.

If $\mathcal{M}_1 \mathcal{G} \mathcal{M}_2$ and $\mathcal{M}_2 \mathcal{G} \mathcal{M}_3$ then $\mathcal{M}_1 \mathcal{G} \mathcal{M}_3$ must be proved to show the property of transitivity. This can be done by stating that if $\mathcal{M}_1$ is transformed into $\mathcal{M}_2$ and $\mathcal{M}_2$ into $\mathcal{M}_3$ each in some finite sequence of multigraph transformations, then it is possible to transform $\mathcal{M}_1$ into $\mathcal{M}_3$ in some finite sequence of transformations and hence $\mathcal{M}_1 \mathcal{G} \mathcal{M}_3$.

Since $\mathcal{G}$ is reflexive, antisymmetric, and transitive it is a partial order. In other words, multigraph transformations preserve the partial order of the relation $\mathcal{G}$. Summarizing the results we obtain the following definition:

Definition 6.2: The relation $\mathcal{G}$ on a set of directed multigraphs $\mathcal{A}$ is a partial order since $\mathcal{G}$ is reflexive, antisymmetric, and transitive. In other words, $(\mathcal{A}, \mathcal{G})$ is a *poset*.

It is still necessary to show that Tsai's data structure complexity measure preserves the ordering. Although Tsai's polynomials resemble characteristic polynomials, preservation of ordering needs to be proved since the method with which Tsai's polynomials are constructed differs from that for characteristic polynomials.

6.3 PROOF THAT TSAI'S POLYNOMIALS PRESERVE PARTIAL ORDER:

Definition 6.3: Assume a set of directed multigraphs $\mathcal{A}$ that were generated using directed multigraph transformations. Further, let $C(\mathcal{A})$ be the set of Tsai's polynomials computed for $\mathcal{A}$. Then, the relation $\mathcal{G}$ on $C(\mathcal{A})$ is a partial ordering and $(C(\mathcal{A}), \mathcal{G})$ is a *poset.*

Tsai's data structure polynomials, as computed from directed multigraphs, can be written as follows:

$$C(\mathcal{A}) = C(\mathcal{A}_1) + C(\mathcal{A}_2) + \ldots + C(\mathcal{A}_n) + S(\mathcal{A}) \qquad (13)$$

or

$$C(\mathcal{A}) = \sum_{i=1}^{n} C(\mathcal{A}_i) + S(\mathcal{A}) \qquad (14)$$

where $C(\mathcal{A})$ refers to the complexity polynomial of the vertex that corresponds to the entire data structure $\mathcal{A}$, $C(\mathcal{A}_i)$ to the complexity polynomial of the strongly connected component $\mathcal{A}_i$ of the data structure, and $S(\mathcal{A})$ to the self-complexity monomial of the data structure. Data structure $\mathcal{A}$ is assumed to contain n strongly connected components. Let $C_o(\mathcal{A})$ be the Tsai's polynomial for an initial directed multigraph and $C_t(\mathcal{A})$, that of the transformed multigraph by employing some multigraph transformations on the initial multigraph. The fact that multigraph transformations preserve the partial order can be shown by demonstrating that the Tsai's polynomial of the initial multigraph is contained in the Tsai's polynomial of the transformed multigraph.

NODE ADDITION TRANSFORMATION:

No node addition could be made from an atomic node and is always made from a non-atomic node. Node addition transformation always creates a new strongly connected component [n+1'th component] with a self-complexity monomial and complexity polynomial of unity in addition to other strongly connected components that are already present. That is,

$$C_t(\mathcal{A}) = \sum_{i=1}^{n} C(\mathcal{A}_i) + 1 \qquad (15)$$

and hence, $C_s(\mathcal{A}) \leq_t C_t(\mathcal{A})$.

NODE REPLICATION TRANSFORMATION:

As noted earlier, the consequence of a node replication transformation on a directed multigraph is nil since the auxiliary definition elimination step of the Tsai's polynomial construction process eliminates any nodes added through the node replication transformation.

Hence, the Tsai's data structure complexity measures of the initial directed multigraph and the transformed multigraph are the same and consequently, $C_s(\mathcal{A}) \leq_t C_t(\mathcal{A})$.

EDGE ADDITION TRANSFORMATION:

We have already noted that edge addition transformations can not be made from an atomic node to a non-atomic node without destroying the features of the programming language. As before, the discussion is carried on in terms of forward edge addition and backward edge addition transformations. Furthermore, unlike structured flowgraphs, a node can have an arrow pointing back to itself. Note that this is how a self-reference of a data

structure to itself is depicted. Such circular definitions, where a node has an arc pointing back to itself--i.e., a circular path of length one--are grouped under backward edge addition transformations.

FORWARD EDGE ADDITION TRANSFORMATION:

Case 1 -- Between Two Strongly Connected Components: No new strongly connected components are generated. Since the edge does not fall within a single strongly connected component, this addition has no impact on either the self-complexity monomial or on the complexity polynomial. Therefore, $C_e(\mathcal{A}) \leq_e C_i(\mathcal{A})$.

Case 2 -- Within A Strongly Connected Component: Addition of one edge raises the coefficient of the self-complexity monomial of the strongly connected component by unity. The polynomial of the new directed multigraph, therefore, is larger in the term corresponding to the strongly connected component. Hence, $C_e(\mathcal{A}) \leq_e C_i(\mathcal{A})$.

BACKWARD EDGE ADDITION TRANSFORMATION:

Case 1 -- Between More Than One Strongly Connected Components: These strongly connected components merge into a single strongly connected component. The self-complexity monomial of the merged component is freshly calculated and is not the sum of the self-complexity monomials of the individual components that merged into one. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two strongly connected components with self-complexity monomials $S(\mathcal{A}_i)$ and $S(\mathcal{A}_j)$:

$$S(\mathcal{A}) = (\mathcal{V}_i + \mathcal{E}_i)\ c^{\mathcal{L}_i} \qquad \text{for } i = 1, 2. \tag{16}$$

where $\mathcal{V}_i$ is the number of nodes, $\mathcal{E}_i$ the number of edges, and $\mathcal{L}_i$ is the number of simple circular paths in the ith strongly connected component. Assume that these two components, viz., $\mathcal{A}_1$ and $\mathcal{A}_2$ are linked by a backward edge addition transformation. Now the self-complexity monomial of the new strongly connected component, call it $\mathcal{S}(\mathcal{A}_1 \cup \mathcal{A}_2)$, becomes:

$$\mathcal{S}(\mathcal{A}_1 \cup \mathcal{A}_2) = [(\mathcal{V}_1 + \mathcal{V}_2) + (\mathcal{E}_1 + \mathcal{E}_2 + e)] \cdot c^{(\mathcal{L}_1+\mathcal{L}_2+1)} \qquad (17)$$

where $\mathcal{V}_i$, $\mathcal{E}_i$ and $\mathcal{L}_i$ for $i=1, 2$, have the same meanings as before but e is the number of forward pointing edges from $\mathcal{A}_1$ to $\mathcal{A}_2$ or from $\mathcal{A}_2$ to $\mathcal{A}_1$ which were uncounted because $\mathcal{A}_1$ and $\mathcal{A}_2$ were two separate strongly connected components, and c is raised by one more than $\mathcal{L}_1+\mathcal{L}_2$ since the two components are now linked by a new backward edge.

Comparing expressions *(16)* and *(17)*, the self-complexity monomials of the new merged component has a higher order than either of the self-complexity monomials of $\mathcal{A}_1$ and $\mathcal{A}_2$ individually or as a sum of them. It is also true that the coefficient of the monomials of the merged component is larger than that of either of $\mathcal{S}(\mathcal{A}_1)$ and $\mathcal{S}(\mathcal{A}_2)$. Therefore, the complexity polynomial of the transformed data structure is larger than that of the data structure before transformation.

Note that a backward edge addition transformation may merge more than two previously strongly connected components. However, the argument can be extended to cover such cases. The self-complexity monomial of a merged structure when more than two previously strongly connected components were coalesced together can be written as:

$$S\left(\bigcup_{i=1}^{n} \mathcal{A}_i\right) = \left[\sum_{i=1}^{n} \mathcal{V}_i + \sum_{i=1}^{n} \mathcal{E}_i + e\right] \cdot c^{\left(\sum_{i=1}^{n} \mathcal{L}_i + 1\right)} \qquad (18)$$

where $\mathcal{V}_i$, $\mathcal{E}_i$ and $\mathcal{L}_i$ for i=1, 2, ..., n have the same meanings as before but e is the number of forward pointing edges between every two previously strongly connected components $\mathcal{A}_i$ which were uncounted because $\mathcal{A}_i$ were separate strongly connected components. The variable c is raised by one more than $\Sigma\mathcal{L}_i$ since all these components are now linked by a new backward edge.

Case 2 -- Within A Strongly Connected Component: The self-complexity monomial of the strongly connected component is raised in two ways: i) the order of the monomial goes up by one because the backward edge indicates a self-reference to itself or reference to a parent node, and ii) the coefficient increases because there is one more edge now than before the transformation. Therefore, $C_s(\mathcal{A}) \leq_s C_i(\mathcal{A})$.

Case 3 -- Backward Edge Addition From A Node To Itself: The effect on the self-complexity monomial in this case is identical to Case 2.

Hence, we have shown that Tsai's polynomial preserves the order.

6.4 SYNOPSIS:

We noted that if a directed multigraph $\mathcal{M}_2$ can be obtained from the directed multigraph $\mathcal{M}_1$ using directed multigraph transformations, then the set of directed multigraphs and the relation $\mathcal{G}$ [see the start of Chapter 6 for a definition of $\mathcal{G}$] is a partial order.

Tsai's polynomials were then constructed for directed multigraphs and were shown to preserve the order. These polynomials, like characteristic polynomials, do not lend themselves to direct comparison. The mapping measure, $\mathcal{MM}$, discussed in Chapter 4 can be utilized to map Tsai's polynomials to real numbers.

In order to validate characteristic polynomials utilizing the software complexity measure validation paradigm given in Appendix B, let m be a [hypothetical] data structure complexity measure which is strictly monotone with respect to the $\mathcal{G}$ partial order. Then if a and b are data structures with associated directed multigraphs $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, and if $m(a) \leq m(b)$, then we cannot have that $\mathcal{MM}(\mathcal{M}_1) > \mathcal{MM}(\mathcal{M}_2)$. That is, Tsai's polynomials, as a complexity measure, is strictly monotone with respect to the $\mathcal{G}$ preordering of all directed multigraphs.

Chapter 7

AN EXTENDED COMPLEXITY MEASURE?

7.1 COMBINING SEVERAL DATA ITEMS WITHIN A SINGLE SUBPROGRAM:

Tsai's method yields as many polynomials as there are disparate data items within a subprogram. It is still possible to compare the complexities of different data structures by utilizing lexicographical ordering on the space of polynomials in one variable. Consider the following situation. There are two programs that perform the same function essentially. Their source code and data structure differ considerably. Further, assume that the source code complexity has somehow been computed for each program and have been mapped to some real numbers. However, each program has very dissimilar data structures, each with several segregated data items. Computation of Tsai's polynomials, therefore, results in two *sets* of data structure complexity polynomials for each program making it impossible to perform any comparison

between the two programs' data structure complexities. Note that each set of data structure complexity polynomials may contain differing number of elements. It might be useful, therefore, if we could come up with a measure that encompasses the data structure complexity of a given program no matter how many distinct data definitions the program possesses.

Assume two programs whose data structure complexity we want to compare. Consider further that the following [Pascal-like] data definitions make up their data structures:

Data Structure 1

```
var
   x : integer;
   y : real;
   z : array [1..10] of char;
```

Tsai's measure:
{1, 1, 3}

Data Structure 2

```
var
   a : RECORD
        x : integer;
        y : real;
        z : array [1..10] of char;
   END;
```

{6}

The data structures of both the programs contain similar data items except for the obvious difference that the second program uses the RECORD concept to integrate the data items. The fact remains that the complexity of individual components of the RECORD data structure in the second program are the same as that of the data items in the first program. The grouping results in the summation of the individual complexities plus a little more. That little more was added to take into account the incorporation of the RECORD data type.

It might, therefore, be less objectionable to say that the data structure complexity of the first program can be thought of as the sum of the com-

plexities of the various data items, viz., 1+1+3 = 5 rather than the vector {1, 1, 3}. If this argument is agreeable, then it is possible to say that the data structure of the second program is more complex than that of the first. Otherwise, no direct comparison of the two Tsai's polynomials can be made! Unless there are cycles involved in the data structure, this appears reasonable!

7.2 COMBINING CHARACTERISTIC POLYNOMIALS AND DATA STRUCTURE COMPLEXITY:

Combining a source code complexity and a data structure complexity and arriving at a hybrid measure that measures the complexity of a software is an ultimate goal. In that sense, the interest in this paper is to see if Tsai's data structure complexity measure and characteristic polynomial could somehow be collapsed into a single measure. They were both shown to be preserving the partial orders and in their final form they were both polynomials. It appears that combining them is a possibility. However, several problems have to be tackled before that:

- Characteristic polynomials were computed for only one program. Issues relating to computing characteristic polynomials for real-world software programs with several subroutines within a single module and several modules within a single program need to be resolved.
- The issues with respect to obtaining a Tsai's data structure complexity measure for a software system are more multifarious since i) disparate data structures within a single subroutine yield

as many Tsai's polynomials as there are disparate data structures and hence issues relating to computing Tsai's polynomial for a subroutine must be resolved; ii) Issues concerning computing a data structure complexity measure for the software with several modules and subroutines with their own data structures will then have to be resolved.

- There is then the fundamental question of the validity of combining a source code complexity which is based on digraphs and a data structure complexity which is based on multigraphs.
- We have separately validated both complexity measures as shown in the earlier sections. We have to come up with a measure for validating any such extended complexity measure.

7.3 CONCLUSIONS:

Two different complexity measures were detailed in this paper. One of the measures, named, characteristic polynomials, measures the complexity associated with source code and the other, Tsai's polynomial tries to measure data structure complexity. Existence of equivalence classes and partial orderings with respect to both measures were investigated. Such investigation was done in terms of structured flowgraph transformations in case of characteristic polynomials and in terms of directed multigraph transformations in case of Tsai's polynomials.

Four different mapping measures that map the complexity measures to real numbers were tested and one of them was shown to preserve the order.

The complexity measures were then validated using a new complexity measure validation paradigm.

Future work will focus on devising ways to combine source code and data structure complexity measures. However, as pointed out earlier in this chapter, several related issues need to solved in order to achieve a hybrid or extended measure of complexity.

Appendix A.

GRAPHS, RELATIONS, ORDERS, FLOWGRAPHS, AND FLOWGRAPH TRANSFORMATIONS

GRAPHS, RELATIONS, AND ORDERINGS:

More details on these and other concepts can be found in Mott, Kandel, and Baker[8] and in Kolman and Busby[9] as well in countless other discrete mathematics texts.

---

[8]Mott, Joe L., Abraham Kandel, and Theodore P. Baker, *Discrete Mathematics for Computer Scientists*, Reston Publishing Co., Reston, VA, 1983.

[9]Kolman, Bernard and Robert C. Busby, *Discrete Mathematical Structures for Computer Science*, 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

A.1. DIRECTED GRAPHS:

Directed graphs, also called *digraphs,* are a way of viewing relations on sets. A pair of sets $\mathcal{G} = (\mathcal{V},\mathcal{E})$, where the elements of $\mathcal{V}$ are called *vertices* and the elements of $\mathcal{E}$ are called *edges,* is a digraph if $\mathcal{E}$ is a subset of $\mathcal{V} \times \mathcal{V}$. An edge (a,b) is said to be from vertex a to vertex b and is represented in a diagram by an arrow emanating from vertex a and pointing at vertex b. Such an edge is said to be **incident from** a, **incident to** b, and **incident on** both a and b. [Edges have no directions in case of *non-directed graphs* and an edge in non-directed graphs is represented $\{a,b\}$.] The number of edges incident from a vertex is called the *out-degree* of the vertex and the number of edges incident to a vertex is called the *in-degree.*

An edge from a vertex to itself is called a *loop.* A *path* in $\mathcal{G}$ is a sequence of zero or more edges $e_1, \ldots, e_n$ in $\mathcal{E}$ such that $e_i = (v_{i-1},v_i)$ for each $1 \leq i \leq n$. A path is *simple* if all edges and vertices on the path are distinct, except that v_0 and v_n, where v_0 is the vertex at which the path originates and v_n is the vertex where the path terminates, may be equal. A path of length ≥ 1 from a node to itself [where the end nodes are the same] is a *cycle* or *circuit.* A cycle traversing distinct nodes, except for the end nodes, is said to be a *simple cycle.* A digraph is *acyclic* if it contains no cycle and it is *cyclic* if it contains at least one cycle.

Notice also that for any digraph $(\mathcal{V},\mathcal{E})$, $\mathcal{E}$ is a binary relation [see below for a brief discussion on relations] on $\mathcal{V}$. Similarly, any binary relation $\mathcal{R}$

which is a subset of $\mathcal{A} \times \mathcal{B}$ may be viewed as a digraph $\mathcal{G} = (\mathcal{A} \cup \mathcal{B}, \mathcal{R})$. In this sense, the notion of binary relation on a set and the notion of digraph are equivalent.

A.2. RELATIONS:

A relation $\mathcal{R}$ *from* a non-empty set A *to* another non-empty set $\mathcal{B}$ is a subset of $\mathcal{A} \times \mathcal{B}$. If $(a,b) \in \mathcal{R}$ where $a \in \mathcal{A}$ and $b \in \mathcal{B}$, then a is said to be *related to* b by $\mathcal{R}$. It is also common to utilize the form $a \mathcal{R} b$ to denote the relation. If $\mathcal{R}$ is a subset of $\mathcal{A} \times \mathcal{A}$, then $\mathcal{R}$ is expressed as a *relation on* $\mathcal{A}$ [i.e., $a \mathcal{R} a$], instead of writing it as a relation from $\mathcal{A}$ to $\mathcal{A}$.

A.2.1. SOME PROPERTIES OF RELATIONS:

Binary relations, as defined above, exhibit the following special properties among others: *reflexivity, irreflexivity, symmetry, asymmetry, antisymmetry,* and *transitivity.* [Please refer to the references listed in footnotes 10 and 11 for additional details.]

PROPERTY OF REFLEXIVITY:

A relation $\mathcal{R}$ on a set $\mathcal{A}$ is *reflexive* if $(a,a) \in \mathcal{R}$ for *every* $a \in \mathcal{A}$. That is, for all $a \in \mathcal{A}$, $a \mathcal{R} a$.

PROPERTY OF IRREFLEXIVITY:

A relation $\mathcal{R}$ on a set $\mathcal{A}$ is *irreflexive* if $(a,a) \notin \mathcal{R}$ for *every* $a \in \mathcal{A}$. That is, for all $a \in \mathcal{A}$, $a \mathcal{R} a$.

PROPERTY OF SYMMETRY:

A relation $\mathcal{R}$ is *symmetric* if *whenever* $(a,b) \in \mathcal{R}$ then $(b,a) \in \mathcal{R}$. That is, for all $a \in \mathcal{A}$ and $b \in \mathcal{A}$, if $a \mathcal{R} b$ then $b \mathcal{R} a$.

PROPERTY OF ASYMMETRY:

A relation $\mathcal{R}$ on a set $\mathcal{A}$ is *asymmetric* if $(a,b) \in \mathcal{R}$ then $(b,a) \notin \mathcal{R}$. That is, for all $a \in \mathcal{A}$ and $b \in \mathcal{A}$, if $a \mathcal{R} b$ then $b \not\mathcal{R} a$.

PROPERTY OF ANTISYMMETRY:

A relation $\mathcal{R}$ on a set $\mathcal{A}$ is *antisymmetric* if *whenever* $(a,b) \in \mathcal{R}$ and $(b,a) \in \mathcal{R}$ then $a = b$. That is, for all $a \in \mathcal{A}$ and $b \in \mathcal{A}$, if $a \mathcal{R} b$ and $b \mathcal{R} a$ then $a = b$. Utilizing the contra-positive of the above implication, it can be reformulated as, if $a \neq b$, then either $a \not\mathcal{R} b$ or $b \not\mathcal{R} a$.

PROPERTY OF TRANSITIVITY:

A relation is *transitive* if *whenever* $(a,b) \in \mathcal{R}$ and $(b,c) \in \mathcal{R}$ then $(a,c) \in \mathcal{R}$. That is, if $a \mathcal{R} b$ and $b \mathcal{R} c$ then $a \mathcal{R} c$.

A.2.2. EQUIVALENCE RELATIONS:

Equivalence relations are the primary tools employed in the process of abstraction, or selectively ignoring differences which are irrelevant to the purpose at hand. Within a given context, we say that two things are equivalent if the differences between them do not matter.

In terms of formal mathematics, a binary relation is an equivalence relation if it is reflexive, symmetric, and transitive. Given a set $\mathcal{A}$, an

equivalence relation $\mathcal{R}$ on $\mathcal{A}$ is defined to be a binary relation on $\mathcal{A}$ that satisfies the properties of reflexivity, symmetry, and transitivity.

This property of equivalence expresses important aspects of being the same which are ordinarily taken for granted and are usually obvious for specific equivalence relations. Another way of looking at equivalence relations is as a means of dividing things into classes. Any time a set is *partitioned* [see the next section for a definition of partition] into disjoint subsets an equivalence relation is involved. The notions of equivalence relations and equivalence classes are interchangeable.

A.2.2.1. Partitions:

Given a set $\mathcal{A}$, a *partition* of $\mathcal{A}$ is a collection $\mathcal{P}$ of disjoint subsets whose union is $\mathcal{A}$. That is,

1. for any $\mathcal{A}_i \in \mathcal{P}$, $\mathcal{A}_i$ is a subset of $\mathcal{A}$;
2. for any $\mathcal{A}_1 \in \mathcal{P}$ and $\mathcal{A}_2 \in \mathcal{P}$, $\mathcal{A}_1 \cap \mathcal{A}_2 = \phi$, or $\mathcal{A}_1 = \mathcal{A}_2$; and
3. for any $a \in \mathcal{A}$, there exists $\mathcal{A}_i \in \mathcal{P}$ such that $a \in \mathcal{A}_i$.

A.2.2.2. Equivalence Classes:

Given any set $\mathcal{A}$ and any equivalence relation $\mathcal{R}$ on $\mathcal{A}$, the *equivalence class* $[a]$ of each element a of $\mathcal{A}$ is defined $[a] = [b \in \mathcal{A} \mid a \mathcal{R} b]$. Note that we can have $[a] = [b]$, even if $a \neq b$, provided $a \mathcal{R} b$. That is, this notation does not give a *unique name* to each equivalence class.

Digraphs can be treated as a special case of a more general kind of graph, called *directed multigraph*, just as binary relations are a special case of

n-ary relations. [Directed multigraphs are discussed in a following subsection.] However, note that multigraphs are not all relations, just as n-ary relations are not graphs except when n = 2.

A.3. ORDERINGS:

A.3.1. PREORDER:

Given a set $\mathcal{A}$, a relation $\mathcal{R}$ is said to be a *preordering* if set $\mathcal{A}$ satisfies the properties of reflexivity and transitivity.

A.3.2. PARTIAL ORDER:

Given a set $\mathcal{A}$, a *partial order* $\mathcal{R}$ on the set $\mathcal{A}$ is defined to be a binary relation on $\mathcal{A}$ that satisfies the properties of reflexivity, antisymmetry, and transitivity.

The word *partial* in partial ordering comes from the fact that the axioms of reflexivity, antisymmetry, and transitivity do not guarantee that for every pair $(a,b) \in \mathcal{A} \times \mathcal{A}$ at least the relations $a \mathcal{R} b$ or $b \mathcal{R} a$ must hold. That is, partial order does not guarantee that every pair of elements is always comparable. The set $\mathcal{A}$ together with the partial order $\mathcal{R}$ is called a *partially ordered set,* or *poset* and is denoted by $(\mathcal{A},\mathcal{R})$.

A.3.3. TOTAL ORDER:

If set $\mathcal{A}$ is a partially ordered set satisfying relation $\mathcal{R}$ and if for every a and b in $\mathcal{A}$, either $a \leq b$ or $b \leq a$, then $\mathcal{R}$ is called a *total ordering.* It is also called *linear ordering.*

A.4. MONOTONICITY:

Let $a \in \mathcal{A}$ and $b \in \mathcal{A}$ and $\mathcal{A}$ be a preordered set. Further, assume that f is a function from the preordered set $\mathcal{A}$ to a partially ordered set $\mathcal{B}$. The function f is *monotone* if whenever $a \leq b$ then $f(a) \leq f(b)$. f is said to be *strictly monotone* if whenever $a \leq b$ and $a \neq b$, then $f(a) < f(b)$.

A.5. DIRECTED MULTIGRAPHS:

A multigraph is a triple $(\mathcal{V},\mathcal{E},\mathbf{f})$, consisting of a set of vertices $\mathcal{V}$, a set of edges $\mathcal{E}$, and an *incidence function* $\mathbf{f}$ that maps each edge in $\mathcal{E}$ to a pair of vertices in $\mathcal{V}$. For a *directed multigraph*, the incidence function maps edges to ordered pairs of vertices. In this case an edge e is said to be from vertex a to vertex b if and only if $\mathbf{f}(e) = (a,b)$. [For *non-directed multigraphs* the incidence function maps edges to unordered pairs of vertices. In this case, an edge e is said to be between vertex a and vertex b if and only if $\mathbf{f}(e) = \{a,b\}$.]

Note that directed multigraphs [as well non-directed multigraphs] are multigraphs because they may contain more than one distinct edge between two vertices. This does not satisfy the regular definition of graph since graphs allow only one distinct edge between any two vertices. This implies that edges of directed (non-directed) multigraphs can not be expressed as a set of ordered (unordered) pairs of vertices.

A.6. MULTIGRAPH TRANSFORMATIONS:

Most of the material reviewed in the appendixes can be found in the literature elsewhere. This section, however, was developed by the author.

The insight for the material presented here comes from Baker[10], et. al. The need for this material will be seen in the example given as part of Tsai's polynomials where multigraph representing data structures are transformed for the purpose of computing Tsai's polynomials. These three transformations enable adding to or enhancing a multigraph.

For the following transformations assume a directed multigraph $\mathcal{M} = (V,E,\mathbf{f})$ where V denotes the set of vertices, E, the set of edges, and $\mathbf{f}$, an incidence function that maps each edge in E to a pair of vertices in V, i.e., an edge e is said to be from vertex a to vertex b if and only if $\mathbf{f}(e) = (a,b)$.

A.6.1. NODE ADDITION TRANSFORMATION:

Given a directed multigraph $\mathcal{M} = (V,E,\mathbf{f})$, a new directed multigraph $\mathcal{M}' = (V \cup v, E \cup e, \mathbf{f}')$ can be obtained from $\mathcal{M}$ using a node addition transformation if

1. $v \notin V$,

---

[10]Baker, et. al., op. cit., where they describe threa transformations that can be performed on flowgraphs, viz., *node replication, edge addition,* and *path addition transformations*. Assuma a flowgraph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$, whare $\mathcal{N}$ denotas tha set of nodas, $\mathcal{E}$, tha set of edges, s, tha start noda, and t, tha terminal node. Then tha following three transformations can ba defined on a flowgraph.

NODE REPLICATION TRANSFORMATION:

Given a flowgraph $\mathcal{G} = (\mathcal{N},\mathcal{E},s,t)$, a naw flowgraph $\mathcal{G}' = (\mathcal{N} \cup b, \mathcal{E}', s, t)$ can ha obtained from $\mathcal{G}$ using a noda replication transformation if

1. $b \notin \mathcal{N}$,
2. there is ona edge $(a,b) \in \mathcal{E}'$ whara $a \in \mathcal{N}$,
3. $\mathcal{E}'$ includas one edge (b,c) for aach edge of tha form (a,c) in $\mathcal{E}$,
4. $\mathcal{E}'$ doas not includa any of tha adges in $\mathcal{E}$ of the form (a,c), and
5. except for tha changes indicated in items 3. and 4. abova, $\mathcal{E}'$ is tha sama as $\mathcal{E}$.

EDGE ADDITION TRANSFORMATION:

Givan a flowgraph $\mathcal{G} = (\mathcal{N},\mathcal{E},s,t)$, then $\mathcal{G}' = (\mathcal{N},\mathcal{E} \cup e, s, t)$ is ohtained from $\mathcal{G}$ hy an edge addition transformation if e is an edge from noda a to noda b, whare $a \in \mathcal{N} \wedge b \in \mathcal{N} \wedge e \notin \mathcal{E}$.

PATH ADDITION TRANSFORMATION:

Givan a flowgraph $\mathcal{G} = (\mathcal{N},\mathcal{E},s,t)$, then $\mathcal{G}' = (\mathcal{N} \cup c, \mathcal{E} \cup \{(a,c),(c,b)\}, s, t)$ is obtained by path addition transformation if $c \notin \mathcal{N} \wedge a \in \mathcal{N} \wedge b \in \mathcal{N}$.

2. $e \notin \mathcal{E}$, i.e., there does not exist in $\mathcal{M}$ either $\mathbf{f}(e) = (x,y)$ or $\mathbf{f}(e) = (y,x)$ for any $x, y \in \mathcal{V}$, and
3. For every $\mathbf{f}(a) = (x,y)$ for any $x, y \in \mathcal{V}$, where $a \in \mathcal{E}$ there exists $\mathbf{f}'(a) = (x,y)$. In addition, there exists in $\mathcal{M}'$ either $\mathbf{f}'(e) = (x,b)$ or $\mathbf{f}'(e) = (b,x)$ for some $x \in \mathcal{M}$.

A.6.2. NODE REPLICATION TRANSFORMATION:

Given a directed multigraph $\mathcal{M} = (\mathcal{V},\mathcal{E},\mathbf{f})$, a new directed multigraph $\mathcal{M}' = (\mathcal{V} \cup b,\mathcal{E}',\mathbf{f}')$ can be obtained from $\mathcal{M}$ using a node replication transformation if

1. $v \notin \mathcal{V}$,
2. there exists one edge $e1$, $\mathbf{f}'(e1) = (a,v)$, belongs to $\mathcal{E}'$ where $a \in \mathcal{V}$,
3. $\mathcal{E}'$ includes one edge $e2$, $\mathbf{f}'(e2) = (v,b)$ for each edge e of the form $\mathbf{f}(e) = (a,b)$ in $\mathcal{E}$,
4. $\mathcal{E}'$ does not include any of the edges in $\mathcal{E}$ of the form $\mathbf{f}(e) = (a,b)$, and
5. except for the changes indicated in items 3. and 4. above, $\mathcal{E}'$ is the same as $\mathcal{E}$.

A.6.3. EDGE ADDITION TRANSFORMATION:

Given a directed multigraph $\mathcal{M} = (\mathcal{V},\mathcal{E},\mathbf{f})$, a new directed multigraph $\mathcal{M}' = (\mathcal{V},\mathcal{E} \cup e,\mathbf{f}')$ can be obtained from $\mathcal{M}$ using an edge addition transformation if

1. the incidence function $\mathbf{f}'$ is such that for every $\mathbf{f}(e) = (a,b)$ there exists $\mathbf{f}'(e) = (a,b)$ for all $a,b \in \mathcal{V}$,
2. however, $\mathbf{f}'(e) = (a,b)$ does not mean that there exists $\mathbf{f}(e) = (a,b)$, and

3. the number of edges of the form (a,b) in $\mathcal{M}'$ is one greater than the number of edges of the same form in $\mathcal{M}$.

A.7. FLOWGRAPHS:

A flowgraph is basically a digraph. However, to be treated as flowgraphs, additional restrictions are imposed on digraphs. Notations used also differ somewhat. Vertices of digraphs correspond to *nodes* in flowgraphs and edges of digraphs are sometimes also known by the term *arcs*.

1. a single vertex $s \in \mathcal{V}$ is considered the *start node* through which entry into the flowgraph occurs;
2. a single vertex $t \in \mathcal{V}$ is considered the *terminal node* through which an exit is made out of the flowgraph;
3. there exist path(s) from the start node to all the other nodes in the flowgraph;
4. there exist path(s) from any node to the terminal node;
5. except for the above four restrictions, the flowgraph is essentially a digraph.

A flowgraph, therefore, can be defined as $\mathcal{G} = (\mathcal{N},\mathcal{E},s,t)$, where $\mathcal{N}$ is a set of nodes, $\mathcal{E}$ is a set of edges, and s and t are start and end nodes respectively.

A.8. STRUCTURED FLOWGRAPHS:

The foregoing flowgraph definition [and the three flowgraph transformations as discussed in footnote 12] will aid in drawing a flowgraph for any program including non-structured ones. One structured flowgraph [see below for a definition of structured flowgraph] can be transformed into a new, more

complex structured flowgraph by utilizing a set of node replication, edge addition, and/or path addition flowgraph transformations. While the original flowgraph and the transformed flowgraph--after several likely transformations--may be structured, the intermediate flowgraphs might not have retained the structuredness property. Since characteristic polynomials are computed only for structured flowgraphs, there are several intermediate transformations of flowgraphs for which no characteristic polynomial can be computed. Discussion, therefore, is made in terms of structured flowgraphs and structured flowgraph transformations. This subsection describes structured flowgraphs and transformations that retain the structuredness of the flowgraphs intact.

A structured flowgraph, first of all, obeys the restrictions imposed on flowgraphs [see Section A.8]. It can be easily shown that those restrictions are not sufficient to guarantee structured flowgraphs. We will use the following well-known ideas in defining structured flowgraphs. [The use of these ideas results from the formal demonstration by Böhm and Jacopini[11] that any well-behaved computer program, i.e., one without an infinite loop could be constructed using only the *if/then/else* and *while/do* control statements and the appropriate sequential operations.]

A structured flowgraph is, as the name suggests, a flowgraph and hence is a digraph. However, the nodes can be differentiated depending on the basis of their location and the degrees. [The differences among the nodes is often shown by using different shapes and it is a common knowledge that

---

[11]Böhm, Corrado and Giuseppe Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM*, vol. 9, no. 5, May 1966, pp. 366-371.

such diagrams are called *flowcharts*.] The nodes are classified into *action* nodes, *condition* nodes, and *junction* nodes[12]. Action nodes have an in-degree of 1 and an out-degree of 1. Condition nodes have an in-degree of 1 and an out-degree of more than 1. Condition nodes having an out-degree of 2 is also called a *predicate* node. Junction nodes possess an in-degree greater than 1 and an out-degree of 1. There are two exceptions to junction nodes: the *entry* node has an in-degree of 0 and an out-degree of 1 and the *exit* node has in-degree greater than 0 but out-degree equal to zero. As we have noted earlier, a flowgraph must have an entry node and an exit node.

Using the conventional pictorial representations, action nodes are drawn as rectangles, condition nodes as diamonds, and junction nodes as circles. Nevertheless, flowgraphs are still digraphs.

The simplest nontrivial flowgraph [*Figure A.1(a)*] consists of a single action node having one predecessor node [the entry node] and one successor node [the exit node]. Let this flowgraph be called *basic flowgraph*. Given this basic flowgraph, more complex flowgraphs can be constructed by applying what will be called *structured flowgraph transformations*.

Definition of Structured Flowgraphs:

1. a structured flowgraph satisfies the restrictions of flowgraphs as indicated in Section A.8.,
2. the *basic flowgraph* is a structured flowgraph, and

[12]For more on this topic see Art Lew, Computer Science: A Mathematical Introduction, Prentice-Hall, Englewood Cliffs, 1985.

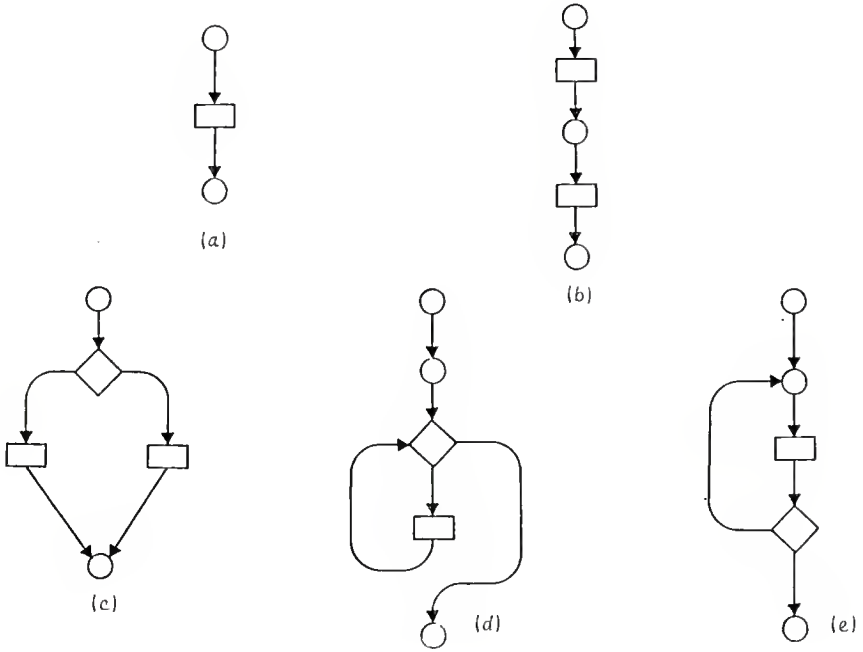

Figure A.1

3. a structured flowgraph on which any or all of the following three transformations were applied is still a structured flowgraph.

A.8.1. STRUCTURED FLOWGRAPH TRANSFORMATIONS:

The basic structured flowgraph [*Figure A.1(a)*] consists of a single action node along with an entry node and an exit node. More complex structured flowgraphs can be constructed by combining simpler flowgraphs in one of the following three ways. Such methods are labeled *structured flowgraph transformations* in this paper.

A.8.1.1. Composition:

Flowgraphs are combined *sequentially* by letting the exit node of one structured flowgraph coincide with the entry node of another, resulting in a *composite* structure. This is the simplest of the three transformations and is exhibited in *Figure A.1(b)*.

A.8.1.2. Alternation:

This transformation results in flowgraphs that depict if-then-else and case structures. In this case, flowgraphs are combined in parallel by letting two or more structured flowgraphs be successors of a common predicate node and also letting them have a common exit node, resulting in an alternation structure. *Figure A.1(c)* shows the resulting structured flowgraph for an if-then-else structure. A flowgraph for a case structure can similarly be drawn easily.

A.8.1.3. Iteration:

Flowgraphs can be transformed by placing flowgraphs within a loop with a predicate node at its beginning or end to govern its termination,

resulting in an iterative structure. Depending on the location of the predicate node, the resulting structured flowgraph represents the familiar while-do or repeat-until structure [*Figure A.1(d)* and *Figure A.1(e)*].

Despite the fact that we have discussed structured flowgraph transformations in terms of composition, alternation, and iteration, they are still in essence node addition, edge addition, and path addition flowgraph transformations [see Footnote 12]. However, these higher level transformations prevent the creation of a non-structured flowgraph from a structured flowgraph.

Appendix B.

A VALIDATION PARADIGM

B.1. A SOFTWARE COMPLEXITY MEASURE VALIDATION PARADIGM:

The contents of this section is ascribed to the work by Baker[13], et. al. Let $\mathcal{A}$ be a set [of, perhaps, programs] and let Ω be the set of real numbers. Now one can assume defining a software complexity measure m which gauges some component of complexity of the elements in $\mathcal{A}$ by mapping $\mathcal{A}$ into Ω.

A *typical validation paradigm* for software complexity measures research would proceed to answer the question: Does the complexity measure m accurately quantify the intended component of software complexity? The paradigm would accomplish that by trying to obtain some ranking of $\mathcal{A}$ which is obtained independently from m but which can still be used to establish the worth of m. Note that this paradigm assumes the existence of a ranking by complexity of $\mathcal{A}$. However, since it is impossible to rank all the elements of $\mathcal{A}$, the ranking

[13]Baker, et. al., op. cit.

of $\mathcal{A}$ is usually approximated by first choosing a subset $\mathcal{A}'$ of $\mathcal{A}$ and then by selecting and using some process data [pd] for each element of the subset $\mathcal{A}'$. Collection of the process data might be done in several possible ways: by using an experimental population to directly obtain a ranking of the elements of $\mathcal{A}'$ either by subjective rankings or by other experiments; by compiling data such as lines of code, size of the compiled code, average length of execution time, person-hours used to produce each element of $\mathcal{A}'$, number of reported/corrected errors or changes, etc.

The process data $pd(a_i)$ for all $a_i \in \mathcal{A}'$ is selected with the intention [or assumption] that if $pd(a_i) < pd(a_j)$ then pd ranks the set of elements $\mathcal{A}'$ and hence this ranking is significant. The values for $pd(a_i)$ for all $a_i \in \mathcal{A}'$ facilitates the examination of the quantitative relationship between the set of values for pd and m on the elements of $\mathcal{A}'$. If the way that pd ranks $\mathcal{A}'$ is sufficiently similar to the way that m ranks $\mathcal{A}'$, then m is concluded to rank the elements of $\mathcal{A}$.

Baker, et. al., note three problems with the above paradigm for validation of software complexity measures:

1. it is unknown that the presumed complexity ranking of the domain set $\mathcal{A}$ exists;
2. it is unknown that the subset $\mathcal{A}'$ of $\mathcal{A}$ is, in some rigorous sense, representative of the domain set $\mathcal{A}$; and
3. it is unknown that the selected process data pd is an adequate approximation of the elusive complexity ranking.

The ranking of the elements of $\mathcal{A}$ that is desired to validate the complexity measure is referred to in discrete mathematics as *orders* on the domain set $\mathcal{A}$. The prime goal of software complexity measures research is to find functions [measures] which preserve these orders. Validating complexity measures can be thought of as a secondary goal. Specifically, validating characteristic polynomials and Tsai's polynomials is one of the missions of this paper.

An assumption in the software complexity measures paradigm is that $\mathcal{A}$ is at least a preordered set and that m is a monotone function from $\mathcal{A}$ to Ω which is, of course, linearly ordered. $\mathcal{A}$ can not be assumed to be linearly ordered since $\mathcal{A}$ is not known to be a partially ordered set. Note, however, that it is difficult to produce preorders for each possible set $\mathcal{A}$ since it is an unknown.

In the complexity measures validation paradigm that was labeled *typical*, validation of complexity measures is attempted by working with a subset $\mathcal{A}'$ of the domain set $\mathcal{A}$. In order to sidestep some of the problems inherent in working with a subset $\mathcal{A}'$ of $\mathcal{A}$, Baker, et. al., suggest avoiding $\mathcal{A}'$ altogether and working with a relation on the domain $\mathcal{A}$ which is weaker [see below for a definition of *weaker ordering*], but more practical, than the elusive preorder on $\mathcal{A}$. Thus, to validate a software complexity measure they try to show that the measure is a monotone function from the more weakly (pre)ordered set to Ω. The particular ordering considered by Baker, et. al., is called *containment ordering*, $\leq_c$.

B.2. WEAKER ORDERING:

An ordering $\mathcal{R}$ is *weaker* than an ordering $\mathcal{T}$ if $\mathcal{R}$ is a subset of $\mathcal{T}$, i.e., whenever, two elements are comparable in $\mathcal{R}$, they are also comparable--with the same relative positions--in $\mathcal{T}$. This can also be written as follows: an ordering $\mathcal{R}$ is weaker than an ordering $\mathcal{T}$ if for all x, y, if $(x,y) \in \mathcal{R}$ then $(x,y) \in \mathcal{T}$.

B.3. CONTAINMENT ORDERING:

Let $\mathcal{A}$ be a set of versions of a program and let $a \in \mathcal{A}$ and $b \in \mathcal{A}$. The versions a and b are related by *containment ordering*, i.e., $a \leq_c b$ if and only if a is contained in b. Informally, this can be taken to mean that a can be found within b. For example, $a \leq_c b$ may mean that every statement that is in a is also in b or that the flowgraph for a is a subgraph of the flowgraph for b.

Note that the relation $\leq_c$ is a partial order on the set $\mathcal{A}$ since $a \leq_c b$ satisfies reflexivity, antisymmetry, and transitivity properties for all $a \in \mathcal{A}$ and $b \in \mathcal{A}$. Hence, $\leq_c$ is also a preorder on $\mathcal{A}$. Note also that containment ordering leads to a family of relations since containment can be defined in different ways. It is the authors' contention that the following two statements are true:

1. every software complexity measure on $\mathcal{A}$ should be monotone from $\mathcal{A}$, preordered by $\leq_c$, to Ω, and

2. $\leq_c$ is nontrivial, i.e., there is enough structure given $\mathcal{A}$ by $\leq_c$ that requiring a measure to be monotone with respect to $\leq_c$ does indeed say a lot about the measure.

B.4. VALIDATION OF A SOFTWARE COMPLEXITY MEASURE:

If $a \in \mathcal{A}$ and $b \in \mathcal{A}$ [say, two versions of a program] and if $a \leq_c b$ by some well-defined sense of containment, then, in general, any complexity inherent in a is also inherent in b. Therefore, for any complexity measure m defined on $\mathcal{A}$, what is desired is that $m(a) \leq m(b)$. It is possible that $m(a) = m(b)$ or that a and b are not comparable at all with respect to $\leq_c$, but if $a \leq_c b$, then $m(a) \leq m(b)$ must hold.

The above result leads to the following validation procedure: suppose that validation of a new software complexity measure m is desired. Assume m is intended to measure a particular component of complexity in the elements of $\mathcal{A}$. As indicated earlier, it is not possible to determine the true associated complexity preordering on $\mathcal{A}$, but it is possible to verify that m is monotone with respect to $\leq_c$ for one or more definitions of containment.

Baker, et. al., argue that m being monotone with respect to $\leq_c$ really says a lot about m with respect to the elusive preordering. One could easily verify that m can distinguish between the lack of this component of complexity and a small amount of it. Let $a \in \mathcal{A}$ be a simple version with none of the component of complexity. If $a \leq_c b$, then b must have at least as much of the component of complexity as a. In such a case, the measure m must be such that $m(a) \leq m(b)$. This sort of comparison can be performed on every pair of

versions which are comparable under $\leq_e$. Whenever $a \leq_e b$, the measure m must rank a and b such that $m(a) \leq m(b)$.

Note, however, that if a and b are *not comparable* under $\leq_e$, then verifying that a measure m is monotone with respect to $\leq_e$ does not say anything about how m will work on a and b. Showing that m is monotone with respect to $\leq_e$ only shows that m is monotone with respect to an ordering which is weaker than the elusive preordering and the weaker ordering is not trivial. Thus showing m is monotone with respect to $\leq_e$ can be thought of as a good "disaster check" for m and we can have confidence that m will do a reasonably good job of ordering a and b.

REFERENCES

[1] Baker, Albert L., James M. Bieman, David A. Gustafson, and Austin C. Melton, "Modeling and Measuring the Software Development Process", *Proceedings of Hawaiian International Conference on Computers and Software*, 1987, pp. 23-30.

[2] Böhm, Corrado and Giuseppe Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM*, vol. 9, no. 5, May 1966, pp. 366-371.

[3] Cantone, Giovanni, Aniello Cimitile, and Lucio Sansone, "Complexity in Program Schemes: The Characteristic Polynomial", *SIGPLAN Notices*, vol. 18, no. 3, March 1983.

[4] Kolman, Bernard and Robert C. Busby, *Discrete Mathematical Structures for Computer Science*, 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[5] Lew, Art, *Computer Science: A Mathematical Introduction*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[6] Mott, Joe L., Abraham Kandel, and Theodore P. Baker, *Discrete Mathematics for Computer Scientists*, Reston Publishing Co., Reston, VA, 1983.

[7] Tsai, W.T., M.A. Lopez, V. Rodriguez, and D. Volovik, "An Approach to Measuring Data Structure Complexity", *Proceedings of IEEE Computer Society's Tenth International Computer Software and Applications Conference*, 1986, pp. 240-246.

AN INVESTIGATION INTO CHARACTERISTIC POLYNOMIALS AND TSAI'S POLYNOMIALS

by

DEIVAN DURAI

B.Sc.(Ag.)., Tamil Nadu Agricultural University, 1976
M.Sc., Indian Agricultural Research Institute, 1978
Ph.D., Kansas State University, 1985

AN ABSTRACT OF A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Complexity associated with a software program is made up of two components, namely, source code complexity and data structure complexity. There is a multitude of complexity measures that try to assess these two components. This report enumerates a source code complexity measure called *characteristic polynomials* and a data structure complexity measure called *Tsai's polynomials*. The method of computation of each measure is detailed.

Characteristic polynomials are computed for the structured flowgraph representations of programs. A new relation is defined on structured flowgraphs in terms of *structured flowgraph transformations*. This relation on the structured flowgraphs is shown to be a partial ordering but not an equivalence ordering. This argument is extended to show a partial ordering associated with characteristic polynomials.

On the other hand, Tsai's polynomials are computed for directed multigraph representations of data structures. A set of transformations called *directed multigraph transformations* is devised and a new relation is defined on directed multigraphs in terms of directed multigraph transformations. This new relation on directed multigraphs is again demonstrated to be a partial ordering but not an equivalence ordering. Further, Tsai's polynomials are shown to exhibit a partial ordering.

A measure that maps these two polynomials to reals is devised. This *mapping measure* facilitates direct comparison of a set of characteristic polynomials or Tsai's polynomials. Some issues involved in obtaining a hybrid complexity measure by combining source code complexity and data structure complexity is stated.