A Measure for Module Cohesion

A Dissertation

Presented to

The Graduate Faculty of The

The University of Southwestern Louisiana

In Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Jagadeesh Nandigam

Spring 1995

A Measure for Module Cohesion

Jagadeesh Nandigam

APPROVED:

Arun Lakhotia, Chairman Assistant Professor of Computer Science	Steve P. Landry Director, Research and Sponsored Programs
William R. Edwards Associate Professor of Computer Science	Claude G. Čech Associate Professor of Psychology
Joan T. Cain Dean, Graduate School	

To the memory of

my mother, Savithri Nandigam
my grandparents, Subbarayudu and Neelaveni Nandigam
my father-in-law, Vasudevan G Tekumalla

Abstract

Module cohesion is a property of a module that describes the degree to which actions performed within the module contribute to single behavior/function. The concept of module cohesion was originally introduced by Stevens, Myers, Constantine, and Yourdon. However, the subjective nature of their definitions has made it difficult to compute the cohesion of a module precisely.

In this dissertation, we have proposed a measure for module cohesion that is amenable to algorithmic computation. The proposed measure for module cohesion has been investigated using proper experimental design and analysis methods. A tool implementing the proposed measure for cohesion has been developed for determining the cohesion of functions in a C program.

In the proposed measure for module cohesion, the output variables of a module are interpreted as the processing elements of a module. The associations between the processing elements of a module are defined in terms of control and data dependencies between the variables of a module. Formal rules are developed to compute cohesion between pairs of processing elements, in terms of control and data dependencies between variables of a module. An algorithm to compute the cohesion of a module using the formal rules mentioned above has been proposed.

The proposed measure has been validated using experiments based on two paradigms: *comprehension* paradigm and *production* paradigm. In an experiment based on *comprehension* paradigm, subjects (programmers) were provided stimulus materials (programs) and were asked to classify the cohesion level of functions within a program, using the concepts of module cohesion as originally described by Stevens et al. Their classifications were then compared to the classification of the same programs made by the tool implementing the proposed measure. The classfication data were analyzed using various statistical tests, including binomial test, analysis of variance, and correlation test. The general

conclusion was that subjects appeared to have some agreement amongst themselves in determining relative cohesion of functions, but the tool was unable to predict these data.

In an experiment based on *production* paradigm, subjects (programmers) produced stimulus materials (programs) and the cohesion of functions in those programs were computed using the tool developed. The programs used in this experiment were based on good design. Assuming that the implementations of programs did not deviate much from the design, we expected that the tool should assign high cohesion levels to functions in these programs as well. The results of this experiment showed that the tool, implementing the proposed measure, in fact, displayed above-chance level correlation between the expected cohesion level of functions and the actual cohesion assignment made based on the proposed measure.

One conclusion of this research is that developing objective measures for module cohesion is necessary since cohesion is an important attribute of software quality, if one needs to study the precise effects of cohesion on software quality. Also, any proposed measure for quantifying software attributes should first be empirically validated with proper experimental design and analysis.

Acknowledgments

I wish to express my sincere appreciation to Dr. Arun Lakhotia for his continued guidance, patience, help, and support throughout this research. His persistence has been vital to the completion of this dissertation. Also, I would like to thank him for patiently listening to me on issues other than research, especially during the last semester of my research.

Thanks to Drs. Steve P. Landry, William R. Edwards, and Claude G. Cech for serving on the committee. Their comments, criticisms, and suggestions made this a better dissertation.

Special thanks to Dr. Steve P. Landry for all his help in numerous occasions during my education at USL. He came to my rescue whenever I needed and I will always remember his kindness and helpfulness. Special thanks to Dr. Claude G. Cech for all the help that he has given me in experimental design and analysis. I would also like to thank Dr. Robert McFatter for his advise on experimental analysis.

I would like to thank Mr. Anurag Bhatnagar, Mr. John Gravely, Mr. Anil K Vijendran, Mr. Bharath Nedunchelian, and Mr. Ganesh Sundaresan, who as fellow graduate students, have contributed to this work through many informal, intense, and valuable technical conversations and assistance. My thanks to Dr. Tat W. Chan and Mr. Anurag Bhatnagar for their friendship and willingness to listen to my problems.

I would like to thank Dr. Muddapu Balaram for allowing me to have a flexible teaching schedule which was crucial to the successful completion of this research. I would like to thank Dr. Y. B. Reddy and Dr. J. Alsabbagh for their help as colleagues.

I would like to express my sincere appreciation and love to my uncle and aunt, Dr. Nandigam Gajendar and Mrs. Shobha Gajendar, for their support and encouragement throughout my education, especially during the early stages of my education in the United States. I would not be in this position today if it were not because of them. I also wish to

thank them, on behalf of my whole family, for all help that they have given us over the years. I also thank my uncle for his help as a colleague at Grambling State University.

I want to express my love and special thanks to my father, Mr. Janardhan Nandigam, for all the sacrifices he had made for the our sake when my mother died leaving five young children to his responsibility. He always wanted me to become a medical doctor, but he is certainly happy now to know that I will be addressed, at least, as 'Dr. Nandigam' from now on. I would also like to thank all my family members, both in the United States and India, for their support and love.

Special thanks to my mother-in-law, Mrs. Vasantha Vasudevan, for all the wonderful and great help she has given to me and my family by willing to leave everything behind in India for the sake of my studies. The completion of this research would not have been possible without her help with the household chores and care for her grandsons. She made sure that the whole family is well fed with the mouth-watering items she made every day; we never had to wait for them. My whole family is going miss her, especially her three year old grandson who will not sleep without her next to him, when she leaves for India soon.

I would like to express my deepest appreciation and thanks to my wife, Asha, for her love, support, and understanding during this seemingly interminable project. The thought of quitting the dissertation research in the middle occurred to me so many times, but it is she who never encouraged this crazy idea and provided love and support to pass all those hurdles that came as a part of the package. She is a great mother to my sons. I am lucky to have such a wonderful life partner and my heartfelt thanks and love to her.

Finally my love and thanks to my adorable sons Nikhil and Nishal for all the daddy-less days they had to spend when I was out of town half of the week, every week for almost two years. They both brought good luck in my life, especially this year has been great in many ways, so far. I love you guys!!

Table of Contents

List of Tables

List of Figures

Chapter 1
Research Objectives

1.1 Introduction

The *cohesion* of a module is a property that describes the degree to which actions performed by/within a module contribute to a single behavior/function. The concept of cohesion was originally introduced by Stevens, Myers, Constantine, and Yourdon [Stevens74, Myers75, Myers78, Yourdon78, Yourdon79]. Stevens et al. defined the notion of cohesion subjectively by encoding in English certain rules that may be used to determine a module's cohesion. The cohesion of a module was defined in terms of the relationships among the processing elements of a module. A processing element in a module was defined as a statement, a group of statements, a data definition or a procedure call. Stevens et al. defined an ordinal scale with seven levels of cohesion based on the types of associations among the processing elements of a module. These levels, in decreasing order of cohesion, are *functional, sequential, communicational, procedural, temporal, logical,* and *coincidental*.

Module cohesion has been associated to the quality of a software. Stevens et al. and Page-Jones claimed that cohesion is associated with effective modularity, a desirable quality of software, and has predictable effects on external software quality attributes such as modifiability, maintainability, and understandability [Yourdon78, Page-Jones88]. Karstu indicated that there appears to be a correlation between module cohesion and number of changes made to a module [Karstu94] such that highly cohesive modules are less likely to need changes. It is generally accepted that, with respect to the quality of software, functional cohesion is the most desirable and coincidental cohesion is the least desirable.

The subjective nature of the Stevens et al.'s definitions of the term *processing element* and the associative principles used to distinguish between the various levels of cohesion make it difficult to determine the cohesion of a module precisely. The subjectivity of their measure also leads to problems when studying the effects of module cohesion on attributes

of software quality. When a measure is not objective, it is difficult to use the measure for predicting purposes.

The goal of the proposed research is to define an objective measure for module cohesion, develop a tool to analyze and determine the cohesion of functions in a C program, and validate the proposed measure using controlled and exploratory experiments.

1.2 Motivations

Over the past decade many changes have taken place in how programs are developed. Emphasis in program development has shifted from *ad hoc* approaches to more systematic approaches. Many software development methodologies, tools and techniques have been proposed as the solution to the so-called *software crisis*. These modern program development methodologies include structured and object-oriented analysis/design/programming, data abstraction, information hiding etc., [Pressman92, Sommerville89, Lewis91]. Many claims have been made in support of these new approaches to software development including enhanced reliability, easier and more thorough testing possibilities, reduced number of bugs in programs, shortened development time, ease of maintenance, and ease of modification etc. Practitioners of these approaches accept these claims without questioning their validity largely because of their intuitive appeal.

But methodological improvements alone do not completely solve the software crisis situation. The field of software engineering needs empirical investigations of the benefits of these methodologies. Unfortunately, the issue of empirical validation, until recently, has been almost totally ignored within the mainstream of software engineering theory and practice. Empirical evaluation of these claims, in most cases, is not possible because of the lack of proper measurement of the benefits of the proposed method. Sound measurement is a prerequisite for sound empirical validation [Fenton91, Baker90]. Successful empirical validation requires proper selection of experiment design, subjects, materials, and measures.

Brooks provides a review of many of the technical problems associated with carrying out effective experimental research [Brooks80].

We are faced with similar problems of proper measurement and sound empirical validation when dealing with attributes of a software. Software attributes that are usually measured are classified into two types: *internal* and *external* [Fenton91, Zuse91, Conte86]. Internal attributes, such as modularity, lines of code, coupling, and cohesiveness, are attributes of a software which can be measured in terms of the software product itself. External attributes, such as reliability, usability, modifiability, and maintainability, are attributes of a software which can only be measured in terms of how the product relates to its environment. External attributes are usually the ones that software managers and software users would like control and predict. However, it is difficult to measure external attributes directly. Internal attributes are usually needed to support the measurement of external attributes. In general, the internal attributes are considered to be the key to improving software quality, i.e., external attributes of a software. In spite of intuitive connections between internal attributes of a software and its external attributes, there have been very few attempts to establish specific relationships. One reason for this is the lack of proper measures for internal attributes of a software. Therefore, it is important to provide accurate and meaningful measures of internal attributes, such as cohesion, of a software.

1.3 Objectives

The major objectives of this research are outlined below:

1) Propose a measure for module cohesion that is objective and algorithmically computable. In our approach, the output variables of a module are interpreted as the module's processing elements. The associations between the processing elements of a module are defined in terms of control and data dependencies between the variables of the module. Associative rules are provided for each cohesion level, except for temporal cohesion. An algorithm to compute the cohesion of a module is proposed.

2) Develop a software tool that is capable of analyzing and determining the cohesion level of functions written in C programming language. The tool should be robust enough to handle industrial-strength programs.

3) Validate the proposed measure for cohesion using a controlled experiment. The intent of this experiment, referred to as *Experiment 1*, is to investigate whether our measure preserves the intent of original definition of module cohesion as provided by Stevens et al. The experiment will involve comparison of cohesion level assignments made by subjects (programmers) using the original definition of cohesion levels with the cohesion level assignments made by the tool implementing the proposed measure.

4) Use the tool developed to analyze large programs obtained from course projects and repositories in the public domain. This is conducted as separate experiments: *Experiment 2* and *Experiment 3*. In *Experiment 2*, multiple releases of each of the two software systems are analyzed to study the distribution of various cohesion levels. In *Experiment 3*, multiple implementations of the same specification/problem by different programmers are analyzed to see how the code-level cohesion differs from the design/specification-level cohesion.

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides the original definition of module cohesion as given by Stevens et al. and a collection of programs used in the rest of the chapters. The sample programs presented in this chapter are used only to demonstrate the various approaches to computing cohesion and are not used for the experimental study. In Chapter 3, we present the proposed approach for computing the module cohesion. Chapter 4 describes the controlled experiment, *Experiment 1*, conducted to validate the proposed measure for module cohesion. Chapter 5 presents the details of the production experiments, *Experiment 2* and *Experiment 3*. Chapter 6 provides a detailed discussion of some recent attempts to make the subjective nature of cohesion more precise.

Chapter 7 provides a summary of research contributions and future research directions. Appendix A describes the implementation details of the *Cohesion Measurement Tool* (*CMT*). Appendix B provides information on processing elements in functions of programs used in *Experiment 1* and *Experiment 3*. Appendix C contains a listing of the materials used in *Experiment 1*.

Chapter 2
The Original Definition of Cohesion

In this chapter the definition of *module cohesion* as originally proposed by Stevens, Myers, Yourdon and Constantine [Stevens74, Myers75, Myers78, Yourdon78, Yourdon79] is presented. Also, a set of programs is presented in Section 2.2 which will be used to illustrate various approaches to computing module cohesion.

2.1 Stevens et al.'s definition of Module Cohesion

Module cohesion is an intramodular measure originally proposed by Stevens, Myers, Yourdon and Constantine [Stevens74, Myers75, Myers78, Yourdon78, Yourdon79]. It is defined as the measure of the strength of functional relatedness among the *processing elements* within a module. A processing element is defined as a statement, a group of statements, a data definition, or a procedure call; that is, it is any piece of code that accomplishes some work or defines some data. Other terms used in the literature to denote the same concept are *module strength*, *module binding*, and *module functionality*.

The original work by Stevens et al. [Stevens74, Myers75, Myers78, Yourdon78, Yourdon79] on module cohesion resulted in identifying three levels of cohesion. This list has been extended and refined to seven levels of cohesion which have become the *de facto* standard. These seven levels of cohesion, in the order of increasing cohesion, are: *coincidental, logical, temporal, procedural, communicational, sequential*, and *functional*. These levels are defined based on certain *associative principles* (properties or characteristics) that relate the processing elements in a module.

The rest of this section contains a discussion on the original seven levels of cohesion as defined by Stevens et al., an algorithmic description of the steps used by them to compute the cohesion of a module, and the subjective methods suggested in the literature to determine the cohesion of a module.

(a) Coincidental Cohesion

Coincidental cohesion occurs when there is little or no meaningful relationship among the processing elements of a module. The following is an example of coincidentally cohesive module:

```
procedure compute_read_write
begin
     A := B + C
     Read  D
     Write F
     If M = 4 then  S = 0
end;
```

(b) Logical Cohesion

Logical cohesion occurs when the processing elements of a module perform a set of related functions, one of which is selected by the calling module at the time of the invocation of the module. The following is an example of a logically cohesive module:

```
procedure process_records (record,code)
begin
     if code = 1 then
      insert record;
     if code = 2 then
      delete record;
     if code = 3 then
      update record;
end;
```

(c) Temporal Cohesion

Temporal cohesion occurs when the processing elements of a module are executed within the same limited period of time during the execution of the system. Typical examples of temporally cohesive modules are for *initialization, termination, housekeeping,* and *clean-up*. An example of a temporally cohesive module is shown below:

```
procedure initialize_all_variables
begin
     x := 0;
     y := 0;
     for i := 1 to n do
      z[i] := 0;
end;
```

(d) Procedural Cohesion

A set of processing elements are procedurally cohesive if they share a common procedural unit. The common procedural unit may be a loop or a decision structure. The module *sum_and_product2* in Figure 2.4 is procedurally cohesive because the only relationship between the processing elements computing *sum* and *prod* is their dependence on the common loop structure.

(e) Communicational Cohesion

A set of processing elements are communicationally cohesive if they reference the same input data and/or produce the same output data. This is the lowest level where processing elements are related to one another by flow of data rather than flow of control. The module *sum_and_product3* in Figure 2.5 is communicationally cohesive because the processing elements that compute *sum* and *prod* reference the same input data item *x*.

(f) Sequential Cohesion

Two processing elements are sequentially cohesive when the output data or results from one processing element serve as input data for the other processing element. The module *sum_and_average* in Figure 2.9 is sequentially cohesive because the output from the computation of summation is input to the computation of mean.

(g) Functional Cohesion

Functional cohesion occurs when all the processing elements of a module contribute to the computation of a single specific result that is returned to the caller of the module. The module *compute_sum* in Figure 2.10 is functionally cohesive because it computes and returns only one value to its caller.

Table 2.1 summarizes the original definitions of cohesion levels by listing for each level of cohesion the associative principle that must hold between a pair of processing elements.

Table 2.1 Associative principles between two processing elements and the corresponding cohesion level

Cohesion	***Associative Principle***
Functional	Both processing elements contribute to a single specific function.
Sequential	The output of one processing element serves as input to the other processing element.
Communicational	Both processing elements reference the same input data and/or output data.
Procedural	Both processing elements belong to the same procedural unit such as a loop or a decision structure.
Temporal	Both processing elements are executed within the same limited time period during the execution of the system.
Logical	One of the processing elements selected at the time of invocation is executed.
Coincidental	None of the other cohesion levels hold between the two processing elements.

Any given module is rarely an example of only one associative principle or cohesion. The processing elements of a module may be related by a mixture of the seven levels of cohesion. A given pair of processing elements can be associated by more than one level of cohesion. The steps suggested by Stevens et al. to determine the cohesion of a module are summarized in Figure 2.1:

Algorithm: *Compute-module-cohesion*
Input: A module's code / design / narrative description
Output: cohesion level of the module
begin
1. Identify the set of processing elements of the module.
2. For each pair of processing elements do
   - Identify the set of associative principles in Table 2.1 that suitably define the association(s) between the pair.
   - The highest level of cohesion corresponding to these principles is the cohesion for the pair.
3. The cohesion of the module is the lowest cohesion that was assigned to any pair of processing elements in step 2.

end

Figure 2.1 The basic steps for computing module cohesion according to Stevens et al. (from [Lakhotia93])

A technique commonly suggested by Stevens et al. and others to determine the cohesion of a module is by writing an English sentence that accurately describes the function

of a module and then examining the sentence structure and keywords in the sentence for an indication of the level of cohesion. Page-Jones [Page-Jones88] provides a decision tree, shown in Figure 2.2, that can be used as an aid in determining the cohesion of a module. Myers [Myers75] provides a table version of decision tree that can be used as an aid in determining the cohesion of a module.

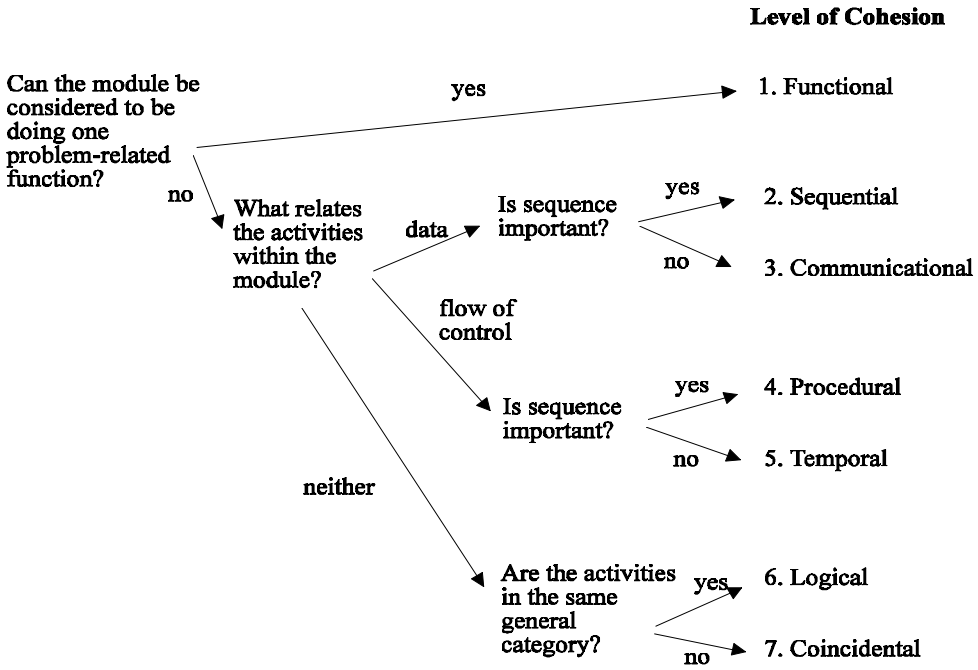

Figure 2.2 Decision tree to determine the cohesion level of a module [Page-Jones88]

2.2 Sample set of programs

A common set of programs is used to illustrate the various approaches to computing module cohesion. These programs are listed below along with their cohesion as per Stevens et al.'s definitions. The procedures *sum_and_product1*, *sum_and_product2*, *sum_and_product3*, and *sum_and_product4* all compute sum and product of a set of numbers, but in different ways. The procedures *sum_or_product1* and *sum_or_product2* compute either sum or product of a set of numbers based on some control flag passed to the

module. For each of the programs, we also give its cohesion level according to Stevens et al.'s definition.

1) The following procedure *sum_and_product1* computes the sum of first *m* natural numbers and the product of first *n* natural numbers using two separate *while* loops.

   Module cohesion: Coincidental

```
 1     procedure sum_and_product1(m,n: integer; var sum,prod: integer);
 2     var i,j: integer;
 3     begin
 4         i := 1;
 5         sum := 0;
 6        while i <= m do begin
 7             sum := sum + i;
 8             i := i + 1;
 9        end;
10        j := 1;
11        prod := 1;
12        while j <= n do begin
13             prod := prod * j;
14             j := j + 1;
15        end;
16     end;
```

   Figure 2.3 Procedure sum_and_product1

2) The following procedure *sum_and_product2* computes the sum and product of first *n* natural numbers using a single *while* loop.

   Module cohesion: Procedural

```
 1     procedure sum_and_product2(n: integer; var sum,prod: integer);
 2     var i: integer;
 3     begin
 4         i := 1;
 5         sum := 0;
 6         prod := 1;
 7         while i <= n do begin
 8              sum := sum + i;
 9              prod := prod * i;
10              i := i + 1;
11         end;
12     end;
```

   Figure 2.4 Procedure sum_and_product2

3) The following procedure *sum_and_product3* first initializes the array *x*. The sum and product of the values in the array *x* are then computed using two separate *while* loops.

Module cohesion: Communicational

```
1       procedure sum_and_product3(n:integer; var x:vector; var sum,prod:integer);
2       var i,j,k: integer;
3       begin
4            i := 1;
5            while i <= n do begin
6                 x[i] := i;
7                 i := i + 1;
8            end;
9            j := 1;
10           sum := 0;
11           while j <= n do begin
12                sum := sum + x[j];
13                j := j + 1;
14           end;
15           k := 1;
16           prod := 1;
17           while k <= n do begin
18                prod := prod * x[k];
19                k := k + 1;
20           end;
21      end;
```

Figure 2.5 Procedure sum_and_product3

4) The following procedure *sum_and_product4* computes the sum and product of the values in an array that is passed as an input. A single *while* loop is used to compute the sum and product.

Module cohesion: Communicational

```
 1      procedure sum_and_product4(x: array; n: integer; var sum,prod: integer);
 2      var i: integer;
 3      begin
 4          i := 1;
 5          sum := 0;
 6          prod := 1;
 7          while i <= n do begin
 8               sum := sum + x[i];
 9               prod := prod * x[i];
10               i := i + 1;
11          end;
12      end;
```

Figure 2.6 Procedure sum_and_product4

5) The following procedure *sum_or_product1* uses a single *if* statement to compute the sum of the first *n* natural numbers if the input flag is 1. If the flag is not equal to 1, it computes the product of the first *n* natural numbers.

Module cohesion: Logical

```
1       procedure sum_or_product1(n,flag: integer; var sum,prod: integer);
2       var i,j: integer;
3       begin
4            if flag = 1 then begin
5                i := 1;
6                sum := 0;
7                while i <= n do begin
8              sum := sum + i;
9              i := i + 1;
10               end;
11           end
12           else begin
13                  j := 1;
14                  prod := 1;
15                  while j <= n do begin
16                      prod := prod * j;
17                      j := j + 1;
18                  end;
19             end;
20      end;
```

Figure 2.7 Procedure sum_or_product1

6) The following procedure *sum_or_product2* computes the sum of the first *n* natural numbers if the input flag is 1. If the input flag is 2 then it computes the product of the first *n* natural numbers.

Module cohesion: Logical

```
1       procedure sum_or_product2(n,flag: integer; var sum,prod: integer);
2       var i,j: integer;
3       begin
4            if flag = 1 then begin
5                 i := 1;
6                 sum := 0;
7                 while i <= n do begin
8                sum := sum + i;
9                i := i + 1;
10                end;
11           end
12           if flag = 2 then begin
13                j := 1;
14                prod := 1;
15                while j <= n do begin
16               prod := prod * j;
17               j := j + 1;
18                 end;
19            end;
20      end;
```

Figure 2.8 Procedure sum_or_product2

7) The following procedure *sum_and_average* computes the sum and average of the first *n* natural numbers.

Module cohesion: Sequential

```
1       procedure sum_and_average(n: integer, var sum,average: integer);
2       var i: integer;
3       begin
4           i := 1;
5           sum := 0;
6           while i <= n do begin
7                sum := sum + i;
8                i := i + 1;
9           end;
10          average := sum / n;
11      end;
```

Figure 2.9 Procedure sum_and_average

8) The following procedure *compute_sum* computes the sum of the first *n* natural numbers.

Module cohesion: Functional

```
1       procedure compute_sum(n: integer; var sum: integer);
2       var i: integer;
3       begin
4           i := 1;
5           sum := 0;
6           while i <= n do begin
7                sum := sum + i;
8                i := i + 1;
9           end;
10      end;
```

Figure 2.10 Procedure compute_sum

Chapter 3
The Proposed Measure for Module Cohesion

This chapter presents our measure for computing the cohesion of a module. In our approach, the output variables of a module are interpreted as the module's processing elements. The associations (or relationships) between the processing elements of a module are defined in terms of control and data dependencies between the variables of a module. These dependencies are represented as a directed graph called a *Variable Dependence Graph (VDG)*.

Various levels of cohesion are defined by a set of associative principles or rules that must hold between pairs of output variables. Our approach does not include temporal cohesion as one of the cohesion levels. We believe that the temporal relationships between the processing elements are difficult to obtain from static analysis of code. An algorithm to compute the cohesion of a module is also presented.

Section 3.1 presents a formal definition of *variable dependence graph*. Section 3.2 presents formal definitions of cohesion levels in our approach. Section 3.3 presents an algorithm to compute module cohesion using the proposed definitions of various levels of cohesion. Section 3.4 describes how the dependencies determined through interprocedural analysis are incorporated in the *VDG* of a module. In our approach to computing module cohesion, we assume that variables in a module are canonical, i.e., every variable has a single purpose. Section 3.5 describes an algorithm to canonicalize variables in a module. In Section 3.6, we present how the proposed measure evolved over time.

3.1 Variable Dependence Graph

The *Variable Dependence Graph* (*VDG*) abstracts the data and control dependencies between the variables of a module. The nodes represent the variables of a module and the edges represent the dependencies between the variables. These dependencies are obtained

through data and control flow analysis of the module [Aho86, Hecht77]. Some useful flow analysis definitions are given here:

Definition 1: The *control flow graph*, or simply a *flow graph*, of a program is a directed graph where the nodes correspond to the basic blocks of the program and the edges represent potential transfer of control between two basic blocks [Aho86, Hecht77].

Definition 2: A *basic block* is a group of statements such that no transfer occurs into a group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially [Hecht77].

Definition 3: A *definition-use chain* of variable x is of the form $<x,n_1,n_2>$, where statement n_1 *defines* the variable x and statement n_2 *uses* the variable x, and there exists a path in the flow graph from n_1 to n_2 which does not contain another definition of x.

Definition 4: A variable y has *data dependence* on variable x, denoted $x \xrightarrow{D} y$, if statement n_1 *defines* x and statement n_2 *defines* y and there is a definition-use chain with respect to x from n_1 to n_2.

Definition 5: A variable y has *control dependence* on variable x due to statement n_1, denoted $x \xrightarrow{C(n)} y$, if statement n contains a predicate that uses x and the execution of the statement that defines y is dependent on the value of the predicate in n.

For each distinct variable of a module, there is a node in the VDG of the module that is labeled with that variable. The dependencies between variables (represented as edges in the VDG) are classified into two types: *data dependence* and *control dependence*. The control dependence edges are further classified into two types: *loop-control* and *selection-control*.

Definition 6: A *VDG* contains a *data dependence edge* from node x to node y labeled "d" if $x \xrightarrow{D} y$.

Definition 7: A *VDG* contains a *loop-control dependence* edge from node *x* to node *y*, labeled "*l*(*n*)" if $x \xrightarrow{C(n)} y$, and *n* is a loop statement such as a *while* or *for* statement.

Definition 8: A *VDG* contains a *selection-control dependence* edge from node *x* to node *y* of the form "*s*(*n*,*k*)", if $x \xrightarrow{C(n)} y$, and *n* is an *if* or *case* statement and *y* is defined in the k^{th} branch.

A *variable dependence graph* (VDG) of a module *M*, denoted V_M, is a directed graph with labeled edges defined as follows, where $v(V_M)$ denotes the set of vertices and $\varepsilon(V_M)$ denotes the set of edges of V_M:

$v(V_M) = Var(M)$, the set of variables of module *M*

$\varepsilon(V_M) = \{ e \mid e = (x \xrightarrow{D} y \vee x \xrightarrow{L(n)} y \vee x \xrightarrow{S(n,k)} y) \wedge x \neq y\}$

A control dependence (of loop or selection kind) is given precedence over data dependence between two variables. That is, if both data and control dependence exists between two variables, we only establish control dependence between them. Figure 3.1 shows a sample program and its VDG constructed using the approach described above:

```
1  procedure sum1ton(n: integer;
                     var sum:integer);
2  var i: integer;
3  begin
4      i := 1;
5      sum := 0;
6      while i <= n do begin
7             sum := sum + i;
8             i := i + 1;
9       end;
10  end;
```

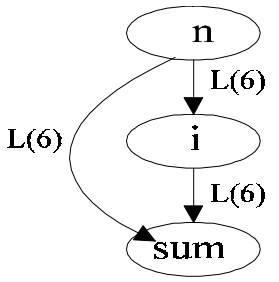

Figure 3.1 A sample program and its Variable Dependence Graph

The control and data flow analysis carried out to construct a variable dependence graph handles the compound data types, such as arrays, structures/records, and pointers as follows: (i) a definition of an array or structure element is considered as a definition to the whole array or structure, (ii) a reference of an array or structure element is considered as a

reference to the whole array or structure, and (iii) pointer variables are treated as regular variables, i.e., a pointer variable such as $*sum$ in C language is treated as a variable *sum*.

3.2 Our Definitions of Cohesion Levels

In our approach, the output variables of a module are treated as the processing elements of the module. The relationship between two output variables is defined using certain associative principles or rules. The associative principles are based on the existence of data and control dependencies between the variables, both output and non-output, of the module.

Our associative principles for designating cohesion between pairs of output variables are stated in Table 3.1 as associative rules AR_i, i = 1..5. Given a pair of output variables x and y of a module M, if associative rule AR_i evaluates to *true* then the variables x and y are said to have the cohesion given by the table. Sometimes it is possible to have more than one type of cohesion between a given pair of output variables. In such a case, the highest level of cohesion that applies is considered to be the cohesion between the pair of the output variables.

We do not include temporal cohesion in our list of cohesion levels because we believe that temporal relationships between processing elements are difficult to obtain from static analysis of code. Functional cohesion is not listed in the Table 3.1, although not excluded from our list of cohesion levels, since functional cohesion is only defined on modules with one output (variable).

Table 3.1 Associative principles between two processing elements

i	*Cohesion* C_i	*Associative Rules* $AR_i: Var \times Var \rightarrow Boolean$
1	Coincidental	$\neg(\wedge \forall_{i,i\in\{2..5\}} AR_i(x,y))$
2	Logical	$\exists z(z \xrightarrow{S(*,*)} x \wedge z \xrightarrow{S(*,*)} y)$
3	Procedural	$\exists z,n,k(z \xrightarrow{L(n)} x \wedge z \xrightarrow{L(n)} y) \vee (z \xrightarrow{S(n,k)} x \wedge z \xrightarrow{S(n,k)} y)$
4	Communicational	$\exists z(z \xrightarrow{D} x \wedge z \xrightarrow{D} y) \vee (x \xrightarrow{D} z \wedge y \xrightarrow{D} z)$
5	Sequential	$x \rightarrow y \vee y \rightarrow x$

1) *Sequential Cohesion*: $x \rightarrow y \vee y \rightarrow x$

Two output variables *x* and *y* are *sequentially* cohesive if variable *x* is dependent (data or control) on variable *y* or variable *y* is dependent (data or control) on variable *x*.

2) *Communicational Cohesion*: $\exists z(z \xrightarrow{D} x \wedge z \xrightarrow{D} y) \vee (x \xrightarrow{D} z \wedge y \xrightarrow{D} z)$

Two output variables *x* and *y* are *communicationally* cohesive if variables *x* and *y* are data dependent on a common variable *z* or the common variable *z* is data dependent on the variables *x* and *y*.

3) *Procedural Cohesion*: $\exists z,n,k(z \xrightarrow{L(n)} x \wedge z \xrightarrow{L(n)} y) \vee (z \xrightarrow{S(n,k)} x \wedge z \xrightarrow{S(n,k)} y)$

Two output variables *x* and *y* are *procedurally* cohesive if variables *x* and *y* are computed within the same loop originating at statement *n* with a predicate containing variable *z* or variables *x* and *y* are computed within the same branch of an *if* or *case* statement originating at statement *n* with a predicate containing variable *z*.

4) *Logical Cohesion*: $\exists z(z \xrightarrow{S(*,*)} x \wedge z \xrightarrow{S(*,*)} y)$

Two output variables *x* and *y* are *logically* cohesive if variables *x* and *y* are dependent on a common variable *z* through some form of selection-control dependence. This includes the cases where (i) variable *x* is defined in *true* branch of *if* statement and variable *y* is defined in *false* branch of the same *if* statement, (ii) variable *x* is defined in one branch of *case* statement and variable *y* is defined in another branch of the same *case* statement, and (iii) variable *x* is defined in some branch of *if* or *case* statement and variable *y* is defined in

some branch of a different *if* or *case* statement. In each case, the selection statement(s) in consideration involves a predicate with the variable *z*.

5) *Coincidental Cohesion*: $\neg(\wedge \forall_{i,i\in\{2..5\}} AR_i(x,y))$

Two output variables *x* and *y* are *coincidentally* cohesive if they are not sequentially, communicationally, procedurally, or logically cohesive. That is, two output variables are coincidentally cohesive if there is neither data nor control relationships between them.

3.3 Algorithm for Computing Module Cohesion

The steps required to compute the cohesion of a module are presented in the algorithm shown in Figure 3.2. In this algorithm, after the cohesion between each output variable (processing element) pair is determined by applying the associative principles listed in Table 3.1, the cohesion of the module as a whole is determined using the principle: *The cohesion of a module is coincidental if all pairs of processing elements are coincidentally cohesive; otherwise it is the lowest cohesion, excluding coincidental, of all pairs of processing elements in the module.*

The rest of this section verifies our measure using a set of sample programs. The approach used to verify our measure proceeds as follows: i) hand-craft a set of programs with specific cohesion, and ii) apply our measure to the hand-crafted programs and examine if the cohesion levels assigned to modules are same as those assigned by Stevens et al.

Figure 3.3 shows the variable dependence graph of the module *sum_and_product1* of Figure 2.3. According to Stevens et al., this module has coincidental cohesion because the computation of *sum* and *prod* do not have anything in common. Our approach also assigns coincidental cohesion to this module since AR_i(*sum*,*prod*) is false for i = 2..5 and the cohesion is not functional since it has more than one output.

```
Algorithm Compute-Module-Cohesion
Input:   VDG of module M
Output:Cohesion of module M
begin
  X ← {output variables in M};
  if |X| = 0 then Cohesion ←  'undefined'
     else if |X| = 1 then Cohesion ← 'functional'
             else begin
                     cohesion_between_pairs ← {};
                     for all x and y in X and x ≠ y do begin
                             cohesion_between_pairs  ←
                             cohesion_between_pairs ∪ max{C_i | i ∈ {1..5} ∧ AR_i(x,y) };
                     end for;
                     if  (∀_i i ∈  cohesion_between_pairs ∧ i = coincidental) then
                             Cohesion ← coincidental;
                     else
                             Cohesion ← min(cohesion_between_pairs - {coincidental});
                     end;
                  end
  return Cohesion
end Compute-Module-Cohesion
```

Figure 3.2 Algorithm for computing the cohesion of a module

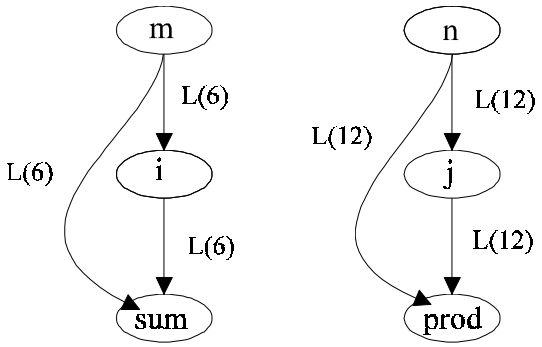

Figure 3.3 VDG of module *sum_and_product1*

Figure 3.4 shows the VDG of the module *sum_or_product2* of Figure 2.8. According to Stevens et al.'s classification, this module has logical cohesion because it computes only one output when invoked depending upon the flag information sent by its caller. Our approach also assigns logical cohesion since the only associative principle that is true is AR_2(*sum*,*prod*).

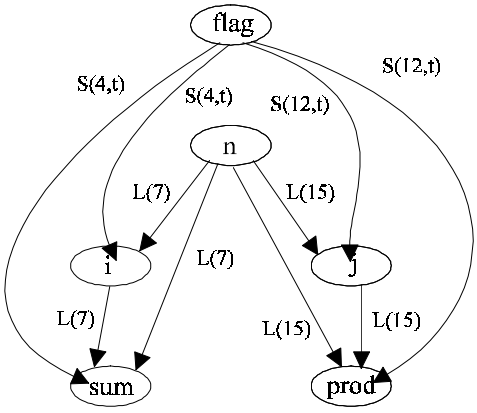

Figure 3.4 VDG of module *sum_or_product2*

As an example of a module with procedural cohesion, consider the Figure 3.5 that shows the VDG of the module *sum_and_product2* of Figure 2.4. According to Stevens et al., this module has procedural cohesion because the processing elements *sum* and *prod* share the common while loop. Our approach also assigns procedural cohesion to this module because the only associative principle that is true is AR_3(*sum*,*prod*).

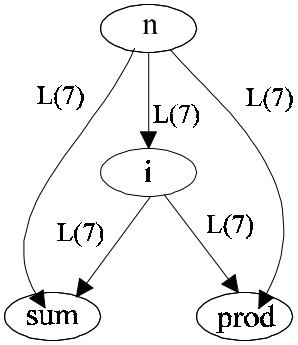

Figure 3.5 VDG of module *sum_and_product2*

Figure 3.6 shows the VDG of the module *sum_and_product4* of Figure 2.6. Stevens et al. classify this module as having communicational cohesion. The reason for this is as follows: The processing elements *sum* and *prod* share the common while loop and also are data dependent on the same input array *x*. Therefore, these processing elements are both procedurally and communicationally cohesive. When there is more than one relationship between two processing elements, the highest level of relationship is applied to the pair.

Thus, the cohesion of the pair is communicational which in turn implies that the module is communicationally cohesive. Our approach also assigns communicational cohesion to this module because AR_3(*sum,prod*) = *true* and AR_4(*sum,prod*) = *true* and the cohesion of the pair is the higher of these two, which is communicational, which in turn makes the module communicationally cohesive.

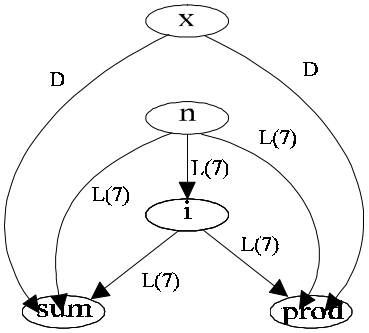

Figure 3.6 VDG of module *sum_and_product4*

Figure 3.7 shows the VDG of the module *sum_and_average* of Figure 2.9. Stevens et al. classify this module as having sequential cohesion because the output of one processing element, *sum*, is used as input to the computation of the processing element *average*. Our approach also assigns sequential cohesion to this module because the only associative principle that is *true* is AR_5(*sum,average*).

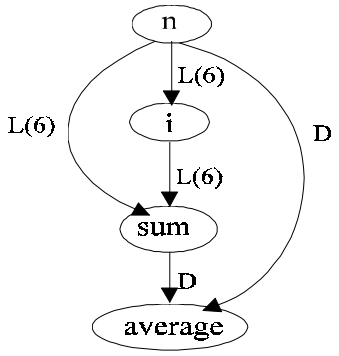

Figure 3.7 VDG of module *sum_and_average*

Figure 3.8 shows the VDG of the module *compute_sum* of Figure 2.10. This module has functional cohesion as per Stevens et al. because the module performs one specific function, i.e., summation. Our approach also assigns functional cohesion to this module because the module produces only one output.

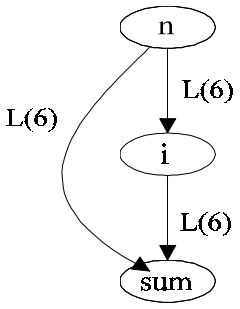

Figure 3.8 VDG of module *compute_sum*

So far, the examples we have illustrated have at most two output variables. The next example illustrates the computation of module cohesion for a module with more than two output variables. Consider the module *sum_sumsqures_product* and its VDG shown in Figure 3.9. The module takes two inputs, *flag* and *n*. If the value of *flag* is 1, then it computes the sum of the first *n* positive integers. Otherwise, it computes both the sum of squares and product of the first *n* positive integers.

The module performs three functions: summation, summation of squares, and multiplication. It performs either summation or summation of squares and multiplication. It has logical cohesion between the processing elements of the first function and the other two functions. The processing elements of the latter two functions are procedurally cohesive because they are performed in a common while loop. The cohesion of the module is logical, i.e., the smaller of logical and procedural.

Our approach also assigns logical cohesion to this module and is computed using associative principles as follows:

$AP_2(sum, sumsquares) = true$

$AP_2(sum, prod) = true$

AP_3(*sumsquares, prod*) = *true*

The cohesion of the module is the smaller of {*logical, procedural*}, which is *logical*.

```
1     procedure sum_sumsquares_product(n,flag: integer;
                        var sum,sumsquares,prod: integer);
2     var i,j: integer;
3     begin
4         if flag = 1 then begin
5             i := 1;
6             sum := 0;
7             while i <= n do begin
8                 sum := sum + i;
9                 i := i + 1;
10            end
11        end
12        else begin
13           j := 1;
14           sumsquares := 0;
15           prod := 1;
16           while j <= n do begin
17                sumsquares := sumsquares + j * j;
18                prod := prod * j;
19                j := j + 1;
20           end
21        end;
22    end;
```

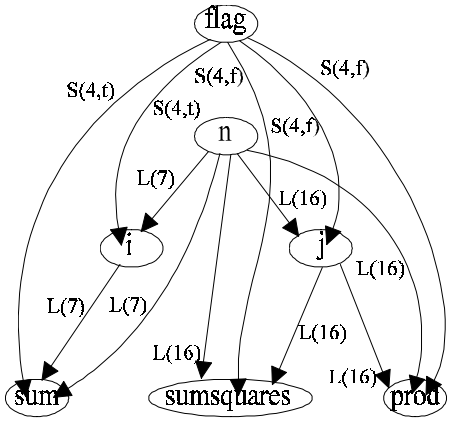

Figure 3.9 VDG of module *sum_sumsquares_product*

3.4 Constructing Variable Dependence Graphs

This section describes the approach used to construct the variable dependence graph of a module containing procedure calls. The construction of a VDG for a module in the presence of procedure calls is similar to the construction of a VDG for a module without

procedure calls with one exception. In the presence of procedure calls, any dependencies between the formal parameters of the called procedure are interpreted as data dependencies between the corresponding actual parameters at the call site. If data dependencies already exist between the actual parameters, established by intraprocedural dependence analysis of the calling module, the dependencies discovered from interprocedural analysis are simply ignored. Otherwise, the discovered dependencies are mapped as data dependencies between the actual parameters at the call site by adding appropriate edges in the variable dependence graph of the calling module.

In presence of procedure calls, it is recommended that the task of construction of VDGs of all functions in a program be finished before cohesion analysis of modules is started. This recommendation is, in fact, too strong. The minimum requirement is that the cohesion analysis of a module should wait until after the VDGs of all its subordinate modules are fully constructed and any possible dependencies due to the calls to these subordinate modules are incorporated in the VDG of the calling module.

An iterative work-list algorithm is used to determine the dependencies between actual parameters of a procedure call due to dependencies between the corresponding formal parameters. This algorithm works by first making a work-list of all possible pairs of formal parameters in a program such that there is either data or control dependence between a pair of formal parameters. The formal parameters of a pair must always be the parameters of the same function.

Once this initial work-list is constructed, a pair of formal parameters is selected. This pair is deleted from the work-list and any dependence (data or control) between the pair is propagated to the corresponding actual parameters at all the calling sites of the function that correspond to the formal parameter pair. The propagation of dependence is achieved by inserting a data edge, if not already present, between the appropriate nodes in the variable dependence graph of the calling function. If the actual parameters at the call site of function are also the formal parameters of the calling function, then a new pair of formal parameters

is constructed and added to the work-list. This process continues until we exhaust all the elements in the work-list.

The algorithm to propagate the dependence between the formal parameters to the corresponding actual parameters is outlined in Figure 3.10 and the algorithm to construct the initial work list is outlined in Figure 3.11. The algorithms presented in Figure 3.10 and Figure 3.11 refer to some functions whose names are indicative of the function performed by these modules and not presented here. For example, the function *determine_call_sites_of_function* takes a function definition and returns a set containing information about all the calls to this function. Similarly, *formal_parameter?* is a function that takes the names of a variable as input and determines whether the variable is a formal parameter or not.

```
Algorithm determine_interprocedural_dependencies_of_vdg
   Input:   VDGs of functions with intraprocedural dependence edges
   Output: VDGs of functions with interprocedural dependence edges added
begin
    for f over functions-defined(program) do
        call-sites(f) ← determine_call_sites_of_function(f);
    end for
    worklist ← initialize_worklist();
    while ~empty(worklist) do
        select a pair of formal parameters, <x,y>, from worklist;
        delete <x,y> from worklist;
        function_called ← function with <x,y> as formal parameters;
        for c over call-sites(function_called) do
           calling_function ← function containing call-site c;
           actual_param1 ← {set of variables in actual parameter corresponding to x};
           actual_param2 ← {set of variables in actual parameter corresponding to y};
           vdg_of_calling_function ← vdg(calling_function);
           if (i ∈ actual_param1) ∧ ( j ∈ actual_param1) ∧
                                ( i →d j ∉ vdg_of_calling_function) then
                 i →d j ∈ vdg_of_calling_function;
                 if i and j are formal parameters of calling_function then
                     add <i,j> to worklist
                   end if
            end if
          end for
   end while
end determine_interprocedural_dependencies_of_vdg
```

Figure 3.10 Algorithm to compute interprocedural dependencies in a VDG

Algorithm *initialize_worklist*
Input: VDGs of functions with intraprocedural dependence edges
Output: worklist
begin
worklist ← {};
for *v* over vdgs of program **do**
if $x \in$ nodes(v) $\wedge$ formal_parameter?(x) $\wedge$
$y \in$ nodes(v) $\wedge$ formal_parameter?(y) $\wedge$ ($x \rightarrow y$) **then**
worklist ← worklist with <x,y>
end if
end for
return worklist
end

Figure 3.11 Algorithm to initialize the worklist of pairs of formal parameters

Now, we will demonstrate, using an example program containing several procedures, the construction of VDGs in the presence of procedure calls. Consider the set of procedures in the program listed in Figure 3.12.

```
1  procedure sum_average_product(n:integer,var sum,prod,average: integer);
2  begin
3      sum_and_average(n,sum,average);
4      product(n,prod);
5  end;

1  procedure sum_and_average(n: integer; var sum,average: integer);
2  begin
3      compute_sum(n,sum);
4      compute_average(n,sum,average);
5  end;

1  procedure compute_sum(n: integer; var sum: integer);
2  var i: integer;
3  begin
4      i := 1;
5      sum := 0;
6      while i <= n do begin
7           sum := sum + i;
8           i := i + 1;
9       end;
10 end;

1  procedure compute_average(n,sum: integer; var average: integer);
2  begin
3      average := sum / n;
4  end;
```

```
1  procedure compute_product(n: integer; var prod: integer);
2  var i: integer;
3  begin
4      i := 1;
5      prod := 1;
6      while i <= n do begin
7           prod := prod * i;
8           i := i + 1;
9      end;
10 end;
```

Figure 3.12 A sample program to illustrate construction of interprocedural dependencies

First, we will construct the VDGs of the procedures in the program of Figure 3.12 with only intraprocedural dependence edges. These VDGs are shown in Figures 3.13 through 3.17.

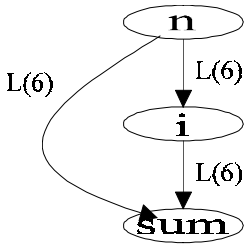

Figure 3.13 VDG for module *compute_sum*

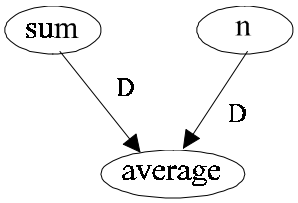

Figure 3.14 VDG for module *compute_average*

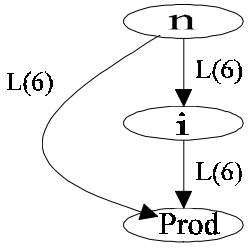

Figure 3.15 VDG for module *compute_product*

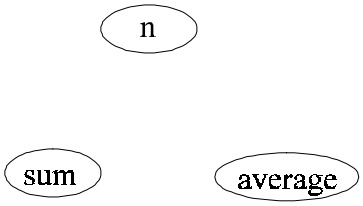

Figure 3.16 VDG for module *sum_and_average*

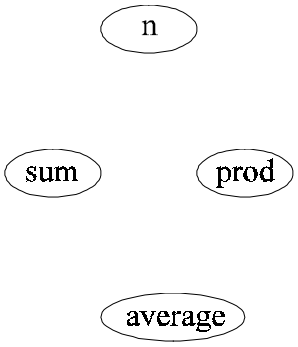

Figure 3.17 VDG for module *sum_average_product*

The VDGs of functions *sum_and_average* and *sum_average_product* have no dependence edges after intraprocedural dependence analysis. The reason for this is that the body of these functions contain only calls to other functions. The dependencies between the actual parameters, if any, will only be established after interprocedural dependence analysis.

Now, let us use the algorithms presented in Figures 3.10 and 3.11 to construct the interprocedural dependencies due to procedural calls. The *initialize_worklist* algorithm will initialize the *worklist* to contain the following pairs of formal parameters by analyzing the existing VDGs of functions:

worklist = {<n,sum,compute_sum> ,<sum,average,compute_average>,
<n,average,compute_average> , <n,prod,compute_product>}

Now, the algorithm of Figure 3.10 will select an arbitrary tuple from the worklist, delete it from the worklist, and propagate the dependence to the call site of the function. The processing of tuples <n,sum,compute_sum> will add a data dependence edge from the variable *n* to the variable *sum* in the VDG of function *sum_and_average*. Since these

variables are also the formal parameters of the function *sum_and_average*, a new tuple of the form <n,sum,sum_and_average> is added to the worklist. The processing of the tuples <sum,average,compute_average> will add a data dependence edge from the variable *sum* to the variable *average* in the VDG of function *sum_and_average*. Similarly, the processing of the tuple <n,average,compute_average> will add a data dependence edge from the variable *n* to the variable *average* in the VDG of function *sum_and_average*, respectively. The processing of these two tuples will add two new tuples of the form <n,average,sum_and_average> and <sum,average,sum_and_average> to the worklist. The VDG of function *sum_and_average* after the addition of the interprocedural dependence edges is shown in Figure 3.18. Also, the worklist will now contain the following tuples:

worklist = {<n,sum,sum_and_average> ,<n,average,sum_and_average>
<sum,average,sum_and_average>,<n,prod,compute_product>}

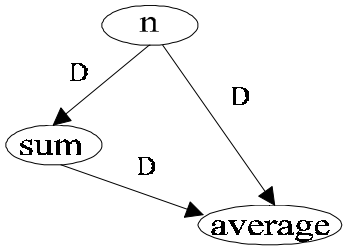

Figure 3.18 VDG for module *sum_and_average*

The algorithm of Figure 3.10 processes the remaining four tuples in a similar manner. The processing of these tuples will add four new data dependence edges to the VDG of function *sum_average_product*. The resulting VDG of this function is shown in Figure 3.19.

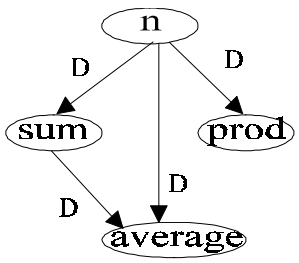

Figure 3.19 VDG for module *sum_average_product*

Now, the worklist will be left with the following four tuples.

worklist = {<n,sum,sum_average_product> ,<n,average,sum_average_product>
<sum,average,sum_average_product>,
<n,prod,sum_average_product>}

The above worklist will be processed in a similar manner by the algorithm of Figure 3.10, assuming there is a main program that calls the function *sum_average_product*. The resulting worklist after such processing will be empty.

3.5 Algorithm for Canonicalization of Variables

A variable is canonicalized if all definitions of the variable are related, i.e., they are defined to achieve a single purpose. A module in which every variable is canonicalized is said to be a canonicalized module.

The variable canonicalizer algorithm is presented in Figure 3.21. The algorithm first places each of the definitions of the variable to be canonicalized into different subsets. The subsets are repeatedly merged such that all definitions belonging to a set are reachable at one or more uses of that variable. At the end of this process, there will be one or more subsets of definitions for a variable. A variable is canonicalized if there exist only one such subset for the variable. A variable is not canonicalized if there exist several subsets of definitions for the variable. Once this partitioning of definitions of variables is done, a variable can be easily canonicalized by replacing each occurrence of the variable from a set with a unique name. This replacement will not affect the functionality of the module. The replacement of variables with unique names is not necessarily done by changing the source code, but can be accomplished by maintaining the information as to which set a particular definition belongs to. This information can be used to tell whether two definitions are related or not.

As an example of variable canonicalization, consider the C function in Figure 3.20, in which the variable *i* has more than one purpose, especially: the computation of sum and the computation of product.

```
1     compute_sum_and_prod(int m,int n,int *sum,int *prod)
2     {
3           int i;
4
5           *sum = 0;
6           for (i = 1; i <= m; i++)
7             *sum = *sum + i;
8           *prod = 1;
9           for (i = 1; i <= n; i++)
10            *prod = *prod * i;
11    }
```

Figure 3.20 An example C function that computes sum and product of numbers

In the above module, the variable *i* involved in the computation of sum is not related to the variable *i* involved in the computation of product. We can safely replace every occurrence of the variable *i* involved in the computation of product, for example, with a unique variable name without affecting the functionality of the module.

Algorithm *variable_canonicalizer*
Input: *D*, set of definitions of a variable *v*
Output: partitioning of input definitions set of a variable into one or more subsets such that the definitions of a subset are related.
begin
for *i* over *size*(*D*) **do**
$d_i \leftarrow \{D_i\}$;
end for
for *j* over *uses*(*v*) **do**
if ($x \in D$) $\wedge$ ($y \in D$) $\wedge$ (*x* reaches *j*) $\wedge$ (*y* reaches *j*) **then**
if ($x \in d_m$) $\wedge$ ($y \in d_n$) $\wedge$ ($m \neq n$) **then**
delete the subset d_m;
merge the definitions of d_m with the subset d_n;
end for
Introduce a dummy *use* for every variable at the end statement
to merge subsets of variables for which there are no uses in the module
end *variable_canonicalizer*

Figure 3.21 Algorithm to canonicalize variables

The introduction of a *dummy* use is needed in some cases where definitions are really related, but the definitions endup in different subsets at the end of the canonicalization process mainly because there are no uses of the definitions within the module. This situation is illustrated with the simple program in Figure 3.22:

```
1     compute_example(int m, int n, int flag, int *x)
2     {
3
4           if (flag == 1)
5              *x = m + n;
6           else
7              *x = m * n;
8     }
```

Figure 3.22 An example to show the need for *dummy* use of variable

In the example of Figure 3.22, there are two definitions of the variable *x*. The canonicalization algorithm will initially place each of these definitions into different subsets. Since there is no *use* of the variable *x* in the module, the algorithm will not merge these definitions of *x* into a single subset, leading to the incorrect conclusion that these two definitions of *x* are not related. The introduction of a dummy use at the *end* of the module will force the algorithm to merge the subsets of definitions into one subset. This action of merging definitions with the use of a dummy variable will identify the correct canonical variables.

3.6 Evolution of the measure

Fenton describes that defining a measure of an attribute of an entity is an iterative process [Fenton91]. One usually begins by identifying the attribute to be measured and developing a means of measuring it. Then data using the defined measure is collected. Analysis of the data usually leads to the clarification and re-evaluation of the attribute. This in turn leads to improvements in the definition of the measure. This iterative process continues until a well-defined measure is developed. In our study of measurement of module cohesion, we were involved in a similar iterative process for capturing the concept of cohesion.

Our measure for module cohesion is an extension of the work done by Lakhotia [Lakhotia91b, Lakhotia93]. Analysis of the data collected from the application of his measure to programs have led us to redefine and/or simplify his definitions for certain levels of cohesion. The definitions of *logical* and *communicational* levels of cohesion have been

simplified. We have applied the new measure to determine the cohesion of modules of a number of large programs. Analysis of the data collected from applying the measure to these large programs in turn led us to refine the formalization of some levels of cohesion used, as described in the rest of this section.

Our experience with processing and analysis of the spreadsheet *SC* and editor *UEMACS* systems, see Chapter 5, have helped formalization of various cohesion levels, especially those of *sequential* and *logical* cohesion levels. Initially, we defined two output variables to be sequentially cohesive if one is data dependent on the other. With this definition of sequential cohesion, a large number of functions in *SC* and *UEMACS* systems were assigned coincidental cohesion. Examination of these functions showed that they contained output variables with one output variable dependent on another output variable through meaningful control dependence. This prompted us to change the definition of sequential cohesion to include control dependence as well as data dependence between two output variables.

Similarly, the definition of *logical* cohesion has been changed to include any type of control dependence of type *selection* between two output variables through a third variable. This change has reclassified some functions that were assigned *coincidental* cohesion to the level of logical cohesion when two output variables were dependent on a third variable with a meaningful control (selection type) dependence.

Our analysis of the data from *Experiment 3*, discussed in Chapter 5, also prompted us to change the algorithm that assigns cohesion level to a module. In our algorithm, a module is assigned coincidental cohesion only if every pair of output variables is coincidentally cohesive. This eliminated the problem with an earlier algorithm that assigned coincidental cohesion to a module as long as there was at least one pair of output variables that was coincidentally cohesive, even when majority of the output variables were highly cohesive.

Chapter 4
Empirical Validation of the Proposed Measure for Cohesion

This chapter presents the results of an experiment, referred to as *Experiment 1*, conducted to validate the proposed measure for cohesion. The objective of this experiment was to investigate correlation between our measure for cohesion and the original definition of cohesion as defined by Stevens et al. [Stevens74]. This was done by asking several graduate students to determine the cohesion of functions in a set of programs and comparing their responses with those of *CMT*, a tool that implements our measure, using both nonparametric statistical tests for nominal data and parametric tests under a stronger assumption that the data constitute an ordered scale. *Experiment 1* can be viewed as using a *comprehension* paradigm whereby subjects (programmers) classify the stimulus materials (programs) and their classifications are compared with those made by *CMT*.

4.1 Subjects

Fifteen students from a graduate level course in software engineering at the University of Southwestern Louisiana voluntarily participated as the subjects for the experiment. Information about their educational background and relevant programming experience is summarized in Tables 4.1 and 4.2.

Table 4.1 Subjects' background information

Subject Background Variable	Low	Mean	High
Years of programming	2	4.6	11
Number of computer science courses in BS	0	10	24
Number of computer science courses in Grad school	3	8	15
Number of programming languages known	1	4	7

Table 4.2 Subjects' familiarity with the C language and cohesion concepts, on the scale of 0 to 10, where 0: never used/heard, 5: about average, and 10: expert

Subject Background Variable	Low	Mean	High
Familiarity with C language	5	7.6	9
Familiarity with cohesion concepts	3	6.1	8

4.2 Experimental Programs

Four C programs obtained over the internet were used for the experiment. These programs were selected from a set of 26 programs used by Goradia in experiments conducted for his Ph.D. thesis [Goradia93]. Their modest size, simplicity, and understandability were the key factors for their selection. Table 4.3 summarizes some size related characteristics of these programs.

Table 4.3 Programs used in the Experiment 1 and their size measures

Program Code	Program Name	No. of Lines	No. of Functions	Avg. lines/ function
P-1	Expression evaluation	83	5	16.6
P-2	Tax form	161	6	26.8
P-3	Accounting	245	6	40.8
P-4	Bank promotion	172	4	43.0

4.3 Experiment Material

The following materials were used to conduct the experiment: (i) material to the administrator of the experiment, (ii) material to the subjects of the experiment, and (iii) *CMT*, the software tool that implements our measure for cohesion.

The material for the administrator of the experiment contained the following items: (i) description of the experiment, (ii) instructions on how to conduct the experiment, (iii) informed consent forms used to obtain the consent of the subjects, (iv) a copy of the text book chapter on module cohesion used for the lecture on module cohesion, (v) experiment packets to be given to the subjects, (vi) copies of a quiz to assess the knowledge of the

subjects, and (vii) slips containing program and subject codes used for assignment of programs to subjects in a random manner.

The packet given to each subject of the experiment contained the following materials: (i) description of the experiment, (ii) instructions to the subjects, (iii) chapter on module cohesion from a text book [Page-Jones88], (iv) source code listing of an experimental program, (v) a sheet to collect the subject's responses, (vi) a sheet to collect the subject's comments about the experiment, and (vii) a questionnaire to collect the subject's educational background.

CMT, a software tool that implements our measure for cohesion, was used to analyze each function of every experimental program and assign it a cohesion level.

A copy of all the material used for the experiment can be found in Appendix C.

4.4 Experiment Procedure

The experiment was conducted in four stages. The activities of each stage are briefly discussed below,

Stage 1:

The administrator of the experiment talked to the students of a graduate level course in software engineering about the experiment and distributed the informed consent forms. The students were encouraged to participate as subjects for the experiment, but were never forced. Fifteen students volunteered to be the subjects and signed the informed consent form; one student refrained.

Stage 2:

The subjects were given a lecture on the concepts of module cohesion, based on Stevens et al.'s work. Experiment packets, containing material mentioned earlier, were distributed to the subjects.

The experiment packets were prepared in the following manner. There were a total of sixteen packets, four for every program used in the experiment (see Table

4.3). Each packet was labeled with a program code and a subject code. The program codes were P1, P2, P3, or P4 (see Table 4.3). The subject codes were formed as follows: S*ij* indicating the *j*th subject assigned to the program P_i. A subject looked at only one program. The experiment employed *between-subjects* design, an experimental design method where subjects are randomly assigned to independent groups [Fenton91]. In our experiment, subjects assigned to each program constitute a group. A maximum of 4 subjects was assigned to any program.

Randomness in assignment of subjects to programs was achieved by having the subjects pick the program code they were to analyze by a random draw. The subjects were assigned code based on their program code (by coding their experiment packet). To ensure anonymity of the subjects, we did not associate subject names with their codes. The names of the subjects were gathered only on informed consent forms during Stage 1 of the experiment. Also, the informed consent forms had no indication of any kind of codes used in the experiment.

The subjects took the experiment packets home and studied the program assigned to them. They were given one week from the day of Stage 2 to complete their task. They were asked to complete and return the following items: (i) data sheet for assigning cohesion level to functions of the subject program, (ii) remarks form containing the subjects' comments on their task, and (iii) background questionnaire containing information about the subjects' educational background.

Stage 3:

In the third stage, a subject studied the program assigned to him/her and assigned cohesion level to each function based on the original definition of module cohesion by Stevens et al. The subjects recorded this information on the data sheet provided to them. They also noted their comments about their task on the remarks form and completed a questionnaire about their educational background.

Stage 4:

In the fourth stage, the subjects were given a quiz on module cohesion. This quiz was to be used to detect and eliminate data by those participants who did not demonstrate an understanding of the basic concepts of module cohesion. The intent of the quiz was not to judge or evaluate the individuals participating in the experiment, but rather to check the validity of the data obtained from the experiment. The subjects returned the quiz in the packet along with the material completed in Stage 3.

4.5 Subjects' Responses

The cohesion levels assigned by the experimental subjects to the functions of Programs P-1 through P-4 are presented in Tables 4.4 through 4.7. The columns marked *Tool* contain the cohesion levels assigned by *CMT* and the columns labeled *LOC* contain the size of a function as measured in lines of code.

Table 4.4 Cohesion assignments for the Expression Evaluation program (P-1)

Function Name	LOC	Tool	S-11	S-12	S-13	S-14
compute	23	functional	logical	logical	communicational	logical
operand_value	4	functional	functional	functional	functional	functional
get_token	7	sequential	sequential	sequential	functional	procedural
evaluate	22	functional	functional	logical	procedural	functional
main	11	functional	communicational	sequential	functional	procedural

Table 4.5 Cohesion assignments for the Tax Form program (P-2)

Function Name	LOC	Tool	S-21	S-22	S-23
initialize	19	coincidental	temporal	temporal	temporal
schedule_A	15	functional	functional	functional	functional
figure_tax	33	functional	functional	logical	functional
compute_tax	20	functional	sequential	functional	sequential
valid_data	17	functional	temporal	logical	functional
main	26	coincidental	sequential	functional	sequential

Table 4.6 Cohesion assignments for the Accounting program (P-3)

Function Name	LOC	Tool	S-31	S-32	S-33	S-34
initialize	15	sequential	functional	temporal	coincidental	temporal
change_monthly	22	sequential	functional	functional	temporal	sequential
process_transaction	15	communicational	communicational	logical	sequential	communicational
process_end_of_month	22	sequential	functional	temporal	functional	temporal
process_report	29	not defined	sequential	logical	temporal	procedural
main	41	sequential	communicational	logical	coincidental	procedural

Table 4.7 Cohesion assignments for the Bank Promotion program (P-4)

Function Name	LOC	Tool	S-41	S-42	S-43	S-44
assess_cashflow	31	coincidental	communicational	logical	communicational	communicational
assess_account_status	19	functional	functional	functional	functional	functional
recommended_account	22	functional	functional	functional	functional	functional
main	57	not defined	procedural	sequential	functional	communicational

Figure 4.1 shows the percent distribution of each cohesion level for all subjects and for the tool.

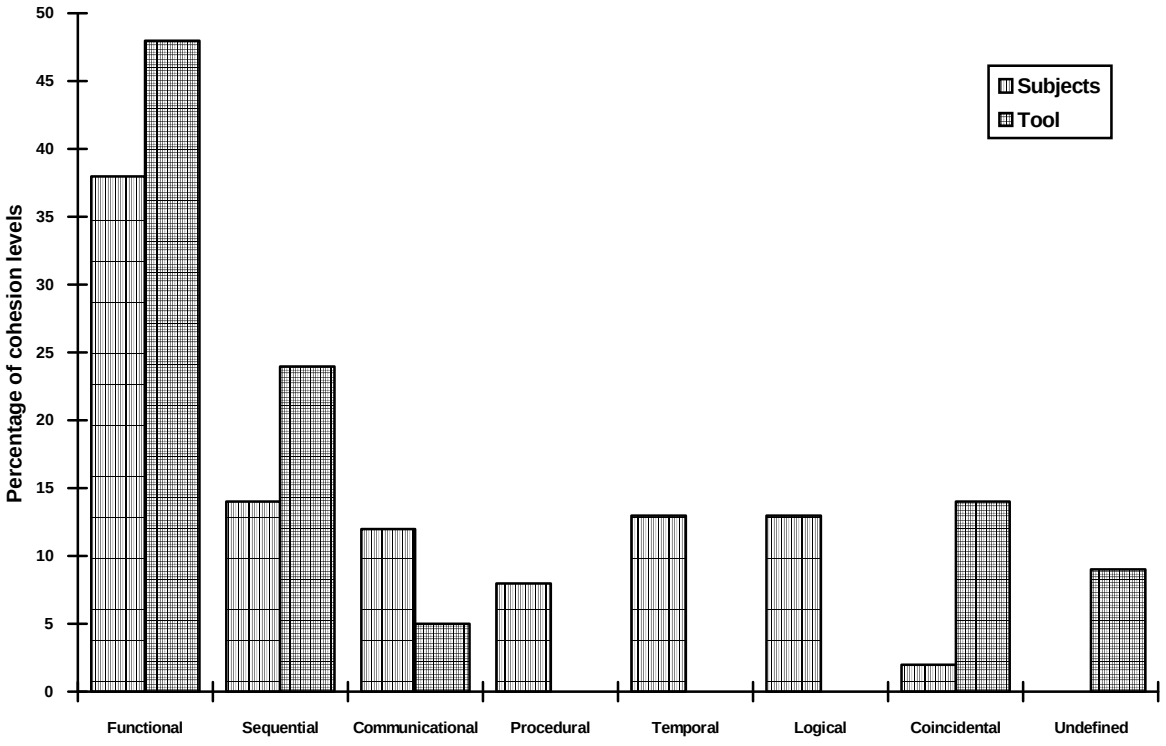

Figure 4.1 Cumulative percentage distribution of cohesion levels for subjects and the tool

4.6 Data Analysis

This experiment investigates the following questions: (i) Do the subjects agree amongst themselves on the cohesion of a function in a program? (ii) Do the subjects agree with the tool on the assignment of cohesion level to functions in a program? These two questions can be stated in terms of the following two experimental hypotheses:

Hypothesis 1: Stevens et al.'s definition of module cohesion is objective.

> *Null hypothesis* (H_0): Subjects are incapable of using the categories of Stevens et al., and therefore their assignments of cohesion levels will be randomly distributed.
>
> *Alternative hypothesis* (H_1): Subjects will display some above-chance consistency amongst themselves in using the categories of Stevens et al.

Hypothesis 2: Our measure is consistent with that of Stevens et al.

> *Null hypothesis* (H_0): The tool developed to objectify the Stevens et al. definition of module cohesion will display chance-level agreements with subjects' ratings.
>
> *Alternative hypothesis* (H_1): The tool developed to objectify the Stevens et al. definition of module cohesion will be at above-chance levels in accounting for the variance displayed by subjects' cohesion level assignments.

The above hypotheses were tested using both nominal and interval statistical tests. In using a nominal statistical test, we viewed the seven levels of cohesion as simple categories, not as scalar values. The *binomial* test was used as the nominal statistical test for analyzing simple category agreements between subjects and tool. After the nominal test, we applied analysis of variance statistical tests with the assumption that the seven levels of cohesion are ordered levels of cohesiveness, as claimed by Stevens et al. The reliability of subjects was analyzed using analysis of variance. Pearson's product-moment correlation test was applied to analyze the agreement between the tool and subjects.

4.6.1 Analysis of data using Binomial test

We have analyzed the experiment data using *binomial* test. For an experiment to qualify as a *binomial* experiment, it must have the following four properties [Mosteller67]:

1) There must be a fixed number of trials.
2) Each trial must result in a "success" or "failure", i.e., it is a binomial trial.
3) All trials must have identical probabilities of success.
4) The trials must be independent of each other.

Experiment 1 satisfies these properties, as justified below:

1) For each program used in the experiment, the number of functions when viewed as number of trials is fixed.
2) Each test for agreement on cohesion level assigned to a function results in a success/match or a failure/no-match.
3) All trials have identical probability of success, i.e., the probability of successful match on cohesion level of one function does not affect the probability of successful match on cohesion level of another function.
4) Assignment of cohesion level to one function is independent of the assignment of cohesion level to another function. Thus, the trials are independent of each other.

Tables 4.8 through 4.11 provide, for each experimental program, the percentage of agreement between each pair of subjects, and also percentage of agreement between each subject and the tool.

Table 4.8 Percentage of agreement amongst subjects and between each subject and the tool for the Expression Evaluation program

Program: Expression Evaluation (P-1)					
	S-11	**S-12**	**S-13**	**S-14**	**Tool**
S-11	-	60%	20%	60%	60%
S-12		-	20%	40%	40%
S-13			-	20%	40%
S-14				-	40%

Table 4.9 Percentage of agreement amongst subjects and between each subject and the tool for the Tax Form program

Program: Tax Form (P-2)				
	S-21	**S-22**	**S-23**	**Tool**
S-21	-	33%	83%	33%
S-22		-	33%	33%
S-23			-	50%

Table 4.10 Percentage of agreement amongst subjects and between each subject and the tool for the Accounting program

Program: Accounting (P-3)					
	S-31	**S-32**	**S-33**	**S-34**	**Tool**
S-31	-	17%	17%	17%	17%
S-32		-	0%	33%	0%
S-33			-	0%	0%
S-34				-	33%

Table 4.11 Percentage of agreement amongst subjects and between each subject and the tool for the Bank Promotion program

Program: Bank Promotion (P-4)					
	S-41	**S-42**	**S-43**	**S-44**	**Tool**
S-41	-	50%	75%	75%	50%
S-42		-	50%	50%	50%
S-43			-	75%	50%
S-44				-	50%

The cumulative probability of observing *r* or more successes, *p-value*, in a binomial experiment is computed using the equation

$$\sum_{x=r}^{n} b(x;n,p), \text{ where } b(x;n,p) = \frac{n!}{x!(n-x)!}\, p^x q^{n-x}$$

where *n* is the number of binomial trials, *r* is the number of successes, *p* is the probability of success, and *q* is the probability of failure. The subjects assign one of seven levels of cohesion. Therefore, the probability of success of a binomial trial is 1/7 or 0.143 under the assumption that these levels should be equally probable.

Tables 4.12 through 4.15 provide, for each of the programs used in the experiment, the cumulative probabilities (*p-value*) of success of a binomial test. An entry with '**' represents an agreement with a 0.05 level of significance; an entry with a '*' represents an agreement with a 0.10 level of significance; other entries are not statistically significant. The significance level, denoted by α, of a test is the probability of rejecting the null hypothesis when it is true (Type I error in a statistical test) [Newmark92].

Table 4.12 Cumulative binomial probabilities (*p-values*) for the Expression Evaluation program

Program: Expression Evaluation (P-1)					
	S-11	**S-12**	**S-13**	**S-14**	**Tool**
S-11	-	0.023**	0.538	0.023**	0.023**
S-12		-	0.538	0.152	0.152
S-13			-	0.538	0.152
S-14				-	0.152

Table 4.13 Cumulative binomial probabilities (*p-values*) for the Tax Form program

Program: Tax Form (P-2)				
	S-21	**S-22**	**S-23**	**Tool**
S-21	-	0.207	0.000**	0.207
S-22		-	0.207	0.207
S-23			-	0.042**

Table 4.14 Cumulative binomial probabilities (*p-values*) for the Accounting program

Program: Accounting (P-3)					
	S-31	**S-32**	**S-33**	**S-34**	**Tool**
S-31	-	0.604	0.604	0.604	0.604
S-32		-	1.000	0.207	1.000
S-33			-	1.000	1.000
S-34				-	0.207

Table 4.15 Cumulative binomial probabilities (*p-values*) for the Bank Promotion program

Program: Bank Promotion (P-4)					
	S-41	**S-42**	**S-43**	**S-44**	**Tool**
S-41	-	0.101*	0.010**	0.010**	0.101*
S-42		-	0.101*	0.101*	0.101*
S-43			-	0.010**	0.101*
S-44				-	0.101*

We can observe from Tables 4.12 through 4.15 that (i) there is little agreement amongst subjects on the assignment of cohesion to various functions, and (ii) there is also little agreement between subjects and the tool. The only exception to the above observations is the case of program *P-4*. For this program, there is a strong agreement among the subjects *S-41*, *S-43*, and *S-44* with a 0.05 level of significance. As may be seen from Tables 4.7 and 4.11, these subjects generally agreed on three of the four functions.

There is no statistically significant evidence to reject the null hypothesis of *Hypothesis 1*. We also cannot completely reject the alternative hypothesis of *Hypothesis 1* as the analysis of Bank Promotion program (P-4) supports the alternative hypothesis with a minimum α of 0.1. The results of the test of *Hypothesis 1* are, therefore, inconclusive.

Testing of *Hypothesis 2* is based on the acceptance of the alternative hypothesis for *Hypothesis 1* since it is meaningless to investigate whether the subjects agree with the tool when there is not much agreement amongst subjects. Because the null hypothesis for *Hypothesis 1* could not be rejected, the data for *Hypothesis 2* were not analyzed further in this series of analyses.

4.6.2 Reliability of subjects using analysis of variance

In using binomial test, we have analyzed the data for a strict match on categories of cohesion. From the results of the binomial test, we have found that there is not much agreement on strict categorical testing. An alternative to categorical testing is to investigate the extent to which subjects agree that some functions are more cohesive than others, i.e., relative ordering of functions on cohesiveness. For example, we can take subjects' assignment of cohesion levels for a given program as representing values on a scale of cohesiveness, and check the extent to which a group of subjects reliably use such a scale. This may be analyzed using an analysis of variance examining reliability of measurements, as discussed in [Winer71, pp 283-289].

Tables 4.16 through 4.19 provide, for each experimental program, the values obtained by transforming the subjects' cohesion level assignments into numbers, where 1 represents functional and 7 represents coincidental, using the original ordering of cohesion levels.

Table 4.16 Subjects' assignment of cohesion levels for the Expression Evaluation program

Function Name	Tool	S-11	S-12	S-13	S-14
compute	1	6	6	3	6
operand_value	1	1	1	1	1
get_token	2	2	2	1	4
evaluate	1	1	6	4	1
main	1	3	2	1	4

Table 4.17 Subjects' assignment of cohesion levels for the Tax Form program

Function Name	Tool	S-21	S-22	S-23
initialize	7	5	5	5
schedule_A	1	1	1	1
figure_tax	1	1	6	1
compute_tax	1	2	1	2
valid_data	1	5	6	1
main	7	2	1	2

Table 4.18 Subjects' assignment of cohesion levels for the Accounting program

Function Name	Tool	S-31	S-32	S-33	S-34
initialize	2	1	5	7	5
change_monthly	2	1	1	5	2
process_transaction	3	3	6	2	3
process_end_of_month	2	1	5	1	5
process_report	-	2	6	5	4
main	2	3	6	7	4

Table 4.19 Subjects' assignment of cohesion levels for the Bank Promotion program

Function Name	Tool	S-41	S-42	S-43	S-44
assess_cashflow	7	3	6	3	3
assess_account_status	1	1	1	1	1
recommended_account	1	1	1	1	1
main	-	4	2	1	3

Table 4.20 through 4.23 show the results of the analysis of variance test, as described in [Winer71, pp 283-289], for each of the four programs used in the experiment. An entry for value of *F* marked with '***' represents reliability at the 0.01 level of significance; an

entry with a '**' represents reliability at the 0.05 level of significance; an entry with '*' represents 0.10 level of significance; and other entries are not significant.

Table 4.20 Analysis of variance for the Expression Evaluation program

Source of variation	SS	df	MS	F
Between functions	38.7	4	9.675	4.21**
Within functions	34.5	15	2.300	
Between subjects	6.0	3	2.000	
Residual	28.5	12	2.375	

Table 4.21 Analysis of variance for the Tax Form program

Source of variation	SS	df	MS	F
Between functions	36.0	5	7.20	2.7*
Within functions	32.0	12	2.67	
Between subjects	5.33	2	2.67	
Residual	26.67	10	2.67	

Table 4.22 Analysis of variance for the Accounting program

Source of variation	SS	df	MS	F
Between functions	21.0	5	4.20	1.03
Within functions	73.5	18	4.08	
Between subjects	32.5	3	10.83	
Residual	41.0	15	2.73	

Table 4.23 Analysis of variance for the Bank Promotion program

Source of variation	SS	df	MS	F
Between functions	21.188	3	7.063	7.21***
Within functions	11.751	12	0.98	
Between subjects	2.188	3	0.729	
Residual	9.563	9	1.063	

Based on these results, it appears that there is significant reliability for programs 1 and 4 at the 0.05 or above level of significance, and for program 2 at the 0.1 level of significance.

Table 4.24 shows the unbiased theta (θ) and the unbiased estimate of the reliability of the mean of the *k* subjects, r_k, values for each of the programs used in the experiment.

Table 4.24 Theta and estimate of the reliability of the mean of the *k* subjects for experimental programs

Program	**Program Code**	**θ**	r_k
Expression Evaluation	P-1	0.6614	0.7257
Tax Form	P-2	0.4157	0.555
Accounting	P-3	-0.0212	-0.0929
Bank Promotion	P-4	1.251	0.833

These reliability estimates of Table 4.24 are conservative, as the analysis of variance did not correct for end anchor effects. However, corrected analyses revealed essentially the same patterns of results.

4.6.3 Analysis of data using correlation test

In this section, we compute the correlation coefficient between the tool and subjects for each of the experimental programs using Pearson product-moment correlation coefficient, as recommended in Couch [Couch87].

Tables 4.25 through 4.28 provide, for each experimental program, the values obtained by transforming the cohesion level assignments of subjects and tool into numbers. The function *process_report* of *Accounting* program (P-3) was excluded from the analysis as the tool was unable to assign a cohesion level to this function. The function *main* of *Bank Promotion* program (P-4) was also excluded from the analysis for the same reason.

Table 4.25 Data for Pearsons test for the Expression Evaluation program

Function Name	Tool	Subjects' Average
compute	1	5.25
operand_value	1	1
get_token	2	2.25
evaluate	1	3
main	1	2.5

Table 4.26 Data for Pearsons test for the Tax Form program

Function Name	Tool	Subjects' Average
initialize	7	5
schedule_A	1	1
figure_tax	1	2.67
compute_tax	1	1.67
valid_data	1	4
main	7	1.67

Table 4.27 Data for Pearsons test for the Accounting program

Function Name	Tool	Subjects' Average
initialize	2	4.5
change_monthly	2	2.25
process_transaction	3	3.5
process_end_of_month	2	3
main	2	5

Table 4.28 Data for Pearsons test for the Bank Promotion program

Function Name	Tool	Subjects' Average
assess_cashflow	7	3.75
assess_account_status	1	1
recommended_account	1	1

Table 4.29 provides the Pearsons product-moment correlation coefficient, *r*, for each of the programs involved in the experiment.

Table 4.29 Pearsons product-moment correlation coefficient between each subject and tool for experimental programs

Program Name	Program Code	Pearsons *r*
Expression Evaluation	P-1	-0.198
Tax Form	P-2	0.333
Accounting	P-3	0.075
Bank Promotion	P-4	0.566

As can be see from the Table 4.29, none of the four coefficients is significant at conventional levels. We also computed the Pearsons product-moment correlation coefficient oever all programs and again found that the coefficient was not significant at conventional levels.

4.7 Power of the Experiment

The power of a statistical test is defined as the probability of rejecting H_0 when H_0 is false. This probability is equal to 1 - β, where β is the probability of a Type II error. A Type II error occurs when we fail to reject H_0 when H_0 is false. It is not possible to specify an exact value for the power of statistical tests conducted in this experiment because H_1 is an inexact hypothesis. Since it did not seem appropriate, at this preliminary, exploratory stage, to specify how much of a match our subjects and tool should exhibit, an exact value for power could not be calculated. Given the approach adopted here, this was not of grave concern. Nevertheless, some factors that affect the power of statistical tests in our experiment are outlined here:

1. The power of a statistical test can be increased by avoiding Type II errors. If Type II errors are to be avoided, then a relatively large sample size or a larger α value is required. As the sample size increases, the power of a statistical test increases. In our experiment, since the sample size is too small (at most four subjects per program), the power of the experiment is weak. As the alpha level becomes more stringent (goes from .05 to .01), the power decreases. This situation did not happen in our experiment as we have used alpha level of 0.05 or 0.10. In our experiment, the major contributor to the weak power seems to be the small sample size.
2. In our experiment, the programs used did not contain many functions (at most six functions per program) and majority of these functions displayed functional cohesion, as measured by the tool. Therefore, the stimulus materials used did not contain sufficiently

large number of functions and the functions did not exhibit a good distribution of cohesion levels. These conditions also affected the power of the experiment.

In order to have an experiment with more power, we need to assign more subjects per program or employ *within-subjects* design as opposed to *between-subjects* design, use programs with sufficiently large number of functions, and use programs where functions exhibit a good distribution of cohesion levels.

4.8 Subjects' Performance on Quiz

As noted earlier, in Stage 4, each subject was quizzed on his/her knowledge on the concept of cohesion. The quiz consisted of one true/false question regarding the definition of module cohesion, one question in which a list of cohesion levels in no particular order had to be rearranged in increasing order of cohesion, seven fill-in the blanks type questions where the definition of a cohesion level had to be related to the name of the cohesion level, and six small code fragments, commonly used in the literature on module cohesion, for which the subjects had to assign a cohesion level. The subjects' answers in the first three components of the quiz were 95% correct which shows that they understood the concepts of module cohesion. The answers of the subjects in the last component (assignment of cohesion to code fragments) had some variations. This again shows that the original definitions are subjective in nature and difficult to apply to determine precisely the cohesion of a code fragment.

4.9 Feedback from the subjects

As a part of the experiment, each subject was asked to complete a remarks form. The remarks form contained the following questions, with possible answers to each question being *easy*, *average*, *difficult*, or *very difficult*:

1) How did you find the lecture on module cohesion to understand?
2) How did you find the material on module cohesion to read and understand?
3) How did you find the definitions of cohesion levels to understand?

4) How did you find the program assigned to you to understand?

5) How did you find the task of assigning of cohesion level to functions?

The feedback from the students to the above questions are summarized in the following table. The values in the Table 4.30 indicate the number of subjects who found the task, as indicated by the question number, as either easy, average, difficult, or very difficult, respectively.

Table 4.30 Summary of feedback information from subjects

Question #	Easy	Average	Difficult	Very difficult
1.	4	7	3	1
2.	8	7		
3.	8	5	2	
4.	10	4	1	
5.	3	8	4	

Most of the subjects reported that the task of assigning cohesion level to functions was not difficult, even though the degree of variation in their responses to the experiment treatment is quite high. One reason for this variation may be that the original definitions of module cohesion by Stevens et al. are more intuitive, hence easy to understand. Since they are not objective, they are hard to apply.

The subjects were also asked to provide any comments they might have about the experiment. Some of their comments were very interesting and emphasized the need for empirical research in software engineering, the intuitive and subjective nature of the Stevens et al.'s definition of module cohesion. Some comments of the subjects are quoted below:

1. *"The definitions of levels of cohesion are intuitive. Therefore, assigning a level of cohesion to a function is also intuitive. It is hard to do so objectively."*
2. *"The first three levels of cohesion (functional, sequential, communicational) were straightforward and easy to understand. The remaining levels of cohesion were confusing."*

3. *"It is not very easy to give an accurate cohesion level assignment for the given source code."*
4. *"Making the final choice of cohesion level for each function is only as good as my best estimate."*

From the feedback data collected and the general comments of the participants of the experiment, we can infer that the intuitive nature of the original definitions of module cohesion is the primary reason for great variations observed in the subjects' responses. This experiment reemphasizes the general agreement that the original rules may be easy to understand, but they are difficult to apply and determine the cohesion of a module precisely.

4.10 Deficiencies of the Experiment

Some limitations/deficiencies of this experiment are presented in this section. As a result of this experiment, we have learned some lessons in how to design and conduct controlled experiments. Future experimental studies involving our measure for module cohesion will address the following limitations:

1) The experiment had only 15 participants, a number not high enough for the *between-subjects* experiment design used. A better alternative to between-subjects design with the same number of subjects could have been *repeated-measures* or *within-subjects* design. A *repeated-measures* or *within-subjects* design requires that all subjects analyze all programs used in the experiment [Fenton91]. This design method has the advantage of requiring fewer experimental subjects and allows for a more efficient use of subjects. The problem of fewer participants could have also been solved by simply using more than 30 subjects since that would then meet the criterion of "large enough" sample, according to the Central Limit Theorem [Newmark92].

2) The programs used in the experiment did not contain functions with even distribution of cohesion, at least as assigned by *CMT*. A different set of programs with a better distribution will increase the significance of this experiment.

3) It may be that the levels of cohesiveness discussed by Stevens et al. are easy to identify by expert programmers, but not by non-experts. Future studies varying experience level of the raters can address this issue.

4.11 Conclusions

In this experiment, we have made an attempt to validate our measure for module cohesion against the measure proposed by Stevens et al. for module cohesion. The experiment has reinforced our assertion that the Stevens et al.'s definition of cohesion is very subjective, since the responses given by the subjects to the experiment treatment have great variations. As a result we have very little data against which the cohesion assigned by our measure can be compared and evaluated.

Chapter 5
Analysis of the Cohesion of Large Programs

This chapter presents the results of analyzing, using *CMT*, some large software systems ranging in size from 650 to 19000 lines of code. The software systems analyzed were obtained from course projects and repositories in the public domain. The analysis of these large systems is presented as two separate experiments, referred to as *Experiment 2* and *Experiment 3*. *Experiment 3* can be regarded as using *production* paradigm in which subjects produce stimulus materials (programs) and their productions are evaluated to see if they fit an expected level of cohesion.

Both the experiments are similar in that they analyze multiple implementations of each software systems. They do, however, differ on how the multiple implementations are arrived at. In *Experiment 2*, the multiple implementations of a program are essentially successive releases of the same software system, whereas in *Experiment 3*, multiple implementations constitute programs for the same problem implemented by different programmers.

There is yet another difference in the programs used in the two experiments. The programs for *Experiment 2* have been taken from repositories on the Internet. They are actually used by people around the world and, as may be inferred from their multiple releases, have been actively "maintained" by their original developers or other volunteer programmers. These programs are large in size and may be considered to be "real-world" programs. The programs of *Experiment 3*, on the other hand, have been developed as part of a course taught at the University of Southwestern Louisiana. They are smaller in size as compared to the programs in *Experiment 2*, but have the benefit that the specification of every function in the program is known.

The data from the two experiments may be used primarily for exploring traits and trends in the cohesion of software systems so as (*a*) to formulate hypotheses for future

investigations and (*b*) to improve the cohesion measure itself. A summary of the experimental subjects, the procedure, and the data collected in the two experiments is presented in Sections 5.1 and 5.2. Observations from the data and other discussions of the two experiments are presented in Section 5.3.

5.1 Experiment 2: Analysis of Real-World Software

In *Experiment 2*, several versions of two public domain software systems, *SC* and *UEMACS*, were analyzed. The system *SC* is a spreadsheet software and *UEMACS* (*MicroEmacs*) is a text editor software. These systems were obtained on Internet from the ftp site *ftp.uu.net*. The purpose of this experiment is to observe the distribution of cohesion levels over various versions of a software system. We have analyzed three versions of the *SC* system, namely *SC-4.1*, *SC-5.1*, and *SC-6.8*. Their size related characteristics are presented in Table 5.1. The original authors of *SC* are James Gosling and Mark Weiser.

Table 5.1 Sizes of the three versions of spreadsheet *SC*

Program Name	Author	Lines of Code	Number of functions
SC-4.1	Robert Bond	4469	88
SC-5.1	Robert Bond	6255	109
SC-6.8	Jeff Burht	10121	198

Figure 5.1 shows, for the three versions of *SC*, the percentage of functions assigned each level of cohesion by *CMT*. The data are analyzed in Section 5.3.

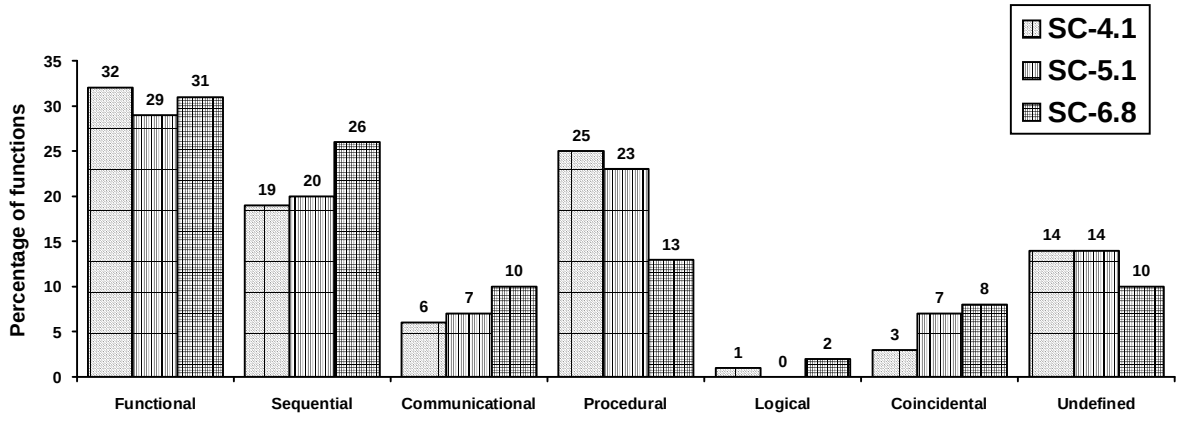

Figure 5.1 Percentage of functions demonstrating various cohesions in the three versions of spreadsheet *SC*

We analyzed four versions of *UEMACS* system, namely *UEMACS-3.6*, *UEMACS-3.7*, *UEMACS-3.8b*, and *UEMACS-3.9*. The size related characteristics of *UEMACS* software are presented in the Table 5.2. The *UEMACS* was originally written by Dave Conroy. The four versions analyzed were written by Daniel Lawrence.

Table 5.2 Sizes of the three versions of text editor *UEMACS*

Program Name	Author	Lines of Code	Number of functions
UEMACS-3.6	Daniel Lawrence	9771	191
UEMACS-3.7	"	12592	268
UEMACS-3.8b	"	16023	315
UEMACS-3.9	"	18884	359

Figure 5.2 shows the percentage of functions assigned each level of cohesion for the four versions of *UEMACS*. The data is analyzed in Section 5.3.

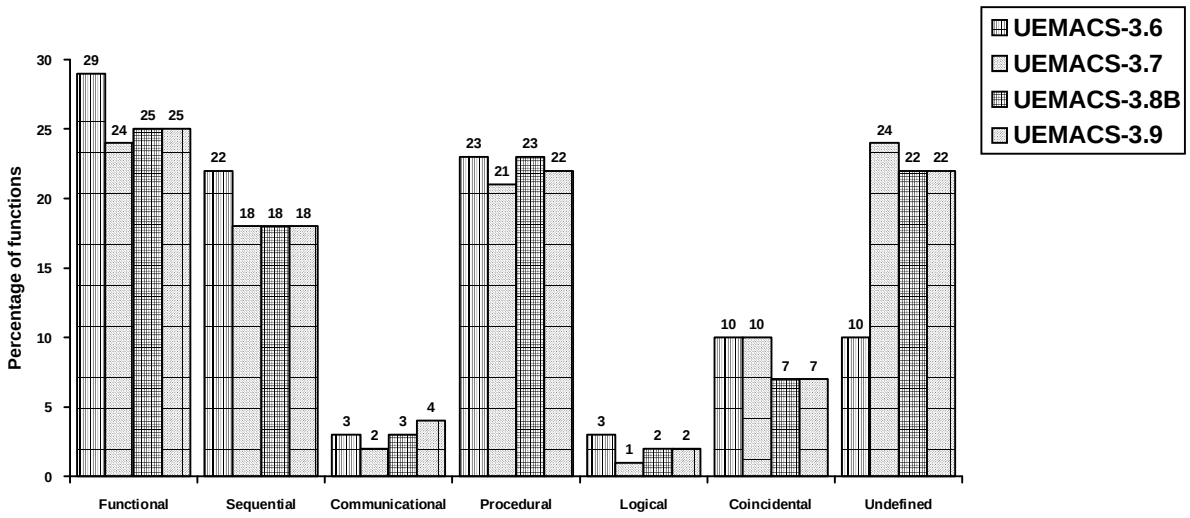

Figure 5.2 Percentage of functions demonstrating various cohesions in the four versions of text editor *UEMACS*

5.2 Experiment 3: Analysis of Course Projects

In *Experiment 3*, several different implementations of three different programming problems were analyzed. These programs were developed in a senior level software

engineering course offered during the Spring 92, Spring 93, and Spring 94 semesters at the University of Southwestern Louisiana (USL). The three programs developed are called *lex.scheme*, *calc*, and *kwic*, respectively. The purpose of this experiment is to investigate how well the cohesion of a module inferred from its specification matches that assigned to its implementations by *CMT*.

An interesting and important property of the programs analyzed in this experiment is that every implementation of each of the three systems contained a common set of functions or modules, which we call *interface functions*. The functional and interface specification of these *interface functions* were provided by the course instructor. Each of the programs implemented these specifications. Compliance with these specifications was strictly enforced by way of severe penalty for any deviation. Even though the functional and interface specification of the *interface functions* were fixed, the choices of data structures, algorithms, and other "hidden" functions were left to the programmer. Therefore, there are significant differences between the multiple implementations of each function of each program.

Table 5.3 summarizes the characteristics of the three systems in terms of the number of available implementations for each system and the average size of implementations for each system.

Table 5.3 Characteristics of the *lex.scheme*, *calc*, and *kwic* systems

Semester	System Name	No. of implementations	Avg. lines of code
Spring 92	*lex.scheme*	14	891
Spring 93	*calc*	20	938
Spring 94	*kwic*	24	667

Sections 5.2.1 through 5.2.3 describe the interface functions for the *lex.scheme*, *calc*, and *kwic* systems, the average size (lines of code) of the implementations of the interface functions for the three systems, and the results of the analysis of these systems by *CMT*. Section 5.3.4 summarizes the results of the *Experiment 3*.

5.2.1 Analysis of the *lex.scheme* system

The program *lex.scheme*, a lexical analyzer for Scheme in C, was developed during Spring 92. This program reads a sequence of characters from the standard input and generates a sequence of tokens to the standard output. There were a total of twenty students in the course. Each of the students developed an implementation of *lex.scheme* system.

The design of the *lex.scheme* system provided by the instructor had eleven interface functions. Of the twenty implementations, fourteen were selected for consideration in this experiment. The remaining six implementations were rejected because they contained missing and/or inconsistent interface functions.

Table 5.4 provides the names of the interface functions, the average size of each interface function, and the distribution of cohesion levels assigned by *CMT*. The numerical value in a given row and column, with the exception of the column labeled "Avg. LOC", of the Table 5.4 indicates the number of implementations of the corresponding interface function that were found to have the level of cohesion that labels the column. The column labeled "Stevens et al." gives the cohesion of interface functions based on Stevens et al.'s definition.

Table 5.4 Analysis of *lex.scheme*: average size of interface functions and the number of implementations of interface functions exhibiting various cohesions.
Shaded boxes denote cases discussed in Section 5.3.

Function Name	Avg. LOC	Stevens et al.	Func	Seq	Comm	Proc	Logi	Coin	Undef
compare_token	24	functional	11	2				1	
get_char	14	functional	6	7				1	
get_token	72	functional	5	6		1		2	
is_eof_char_stream	9	functional	13	1					
is_eof_token	8	functional	13					1	
main	22	functional	3						11
open_char_stream	31	functional	12	2					
open_token_stream	46	functional	2	9				3	
print_token	42	functional	8	1				1	4
unget_char	11	functional	6	1				1	6
unget_error	7	functional	3						11

5.2.2 Analysis of the *calc* system

The students in the Spring 93 semester developed the program *calc*, a simple calculator in C. The *calc* system displays the prompt *"calc> "*, at which the following commands can be given: (i) **set** <*variable_name*> **to** <*expression*> (ii) **print** <*variable_name*> (iii) **quit**.

The design of the *calc* system presented by the instructor had ten interface functions. Of a total of twenty two implementations of this system, twenty implementations were included in this experiment. The remaining two implementations contained missing and/or inconsistent interface functions and therefore were not included in the experiment. Table 5.5 provides the names of the interface functions, the average size of each interface function, and cohesion levels assigned by *CMT*.

Table 5.5 Analysis of *calc*: average size of interface functions and the number of implementations of interface functions exhibiting various cohesions. Shaded boxes denote cases discussed in Section 5.3.

Function Name	Avg. LOC	Stevens et al.	Func	Seq	Comm	Proc	Logi	Coin	Undef
assign_value	29	sequential	8	8	1				3
create_symtab	10	functional	19	1					
evaluate_a_line	59	logical	1	6		13			
get_token	65	functional	6	7		1		6	
get_token_string	5	functional	20						
get_token_type	5	functional	20						
get_value	18	functional	9	9	2				
initialize_calculator	12	temporal	20						
main	24	functional	3						17
match_token_type	8	functional	18					2	

5.2.3 Analysis of the *kwic* system

During Spring 94 the program *kwic* was developed. This program reads a set of lines from a file and outputs KWIC index of the file to the standard output. The KWIC index of a file *S* is the alphabetically ordered sequence of all the circular shifts of all lines in *S*. A line is *circularly shifted* by repeatedly removing its first word and appending this word to the end of

the remaining line. There were 24 students in the course. Each student developed an independent implementation of *kwic* system.

The design of the *kwic* system presented by the instructor had fourteen interface functions. All the twenty four implementations of *kwic* system were included in this experiment. Table 5.6 provides the names of the interface functions, the average size of each interface function, and the results of the analysis by *CMT*.

Table 5.6 Analysis of *kwic*: average size of interface functions and the number of implementations of interface functions exhibiting various cohesions. Shaded boxes denote cases discussed in Section 5.3.

Function Name	Avg. LOC	Stevens et al.	Func	Seq	Comm	Proc	Logi	Coin	Undef
alphabetize	22	functional	10	13				1	
append_to_line	28	sequential	3	18	1		1	1	
circular_shift	30	sequential	10	12			1	1	
circulate_and_add_line	33	sequential		22		1		1	
dump_line	13	functional	14	1					9
dump_line_storage	22	functional	14	3				1	6
empty_line_storage	11	functional	23					1	
get_line	18	functional	12	10			2		
line_cmp	23	functional	6	8	7		1	2	
line_to_string	23	functional	12	10				2	
main	28	functional	11						13
num_of_lines	11	functional	19	3				2	
num_of_words	14	functional	16	3			1	4	
string_to_line	31	functional	15	6				3	

Figure 5.3 summarizes the data of Table 5.4, Table 5.5, and Table 5.6. For each of the programs, *lex.scheme*, *calc*, and *kwic*, the graph shows the percentage of implementations exhibiting various levels of cohesion taking into account all the interface functions of a given program. Tables B.5 through B.7 of Appendix B provide information on average number of processing elements for implementations of the interface functions for each of three systems studied in *Experiment 3*.

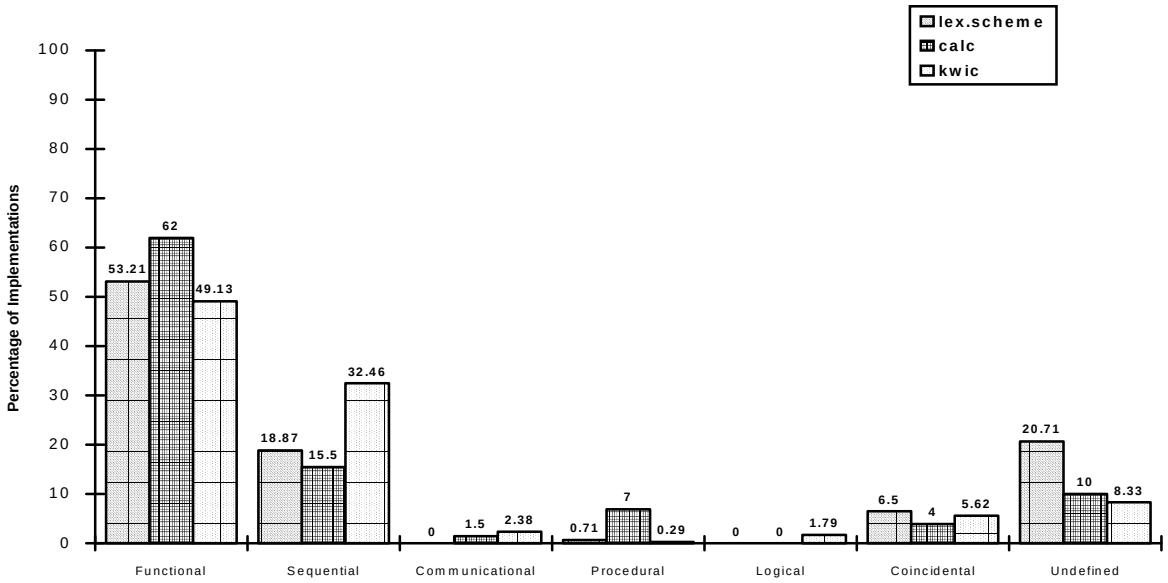

Figure 5.3 Percentage of implementations in each cohesion category for *lex.scheme*, *calc*, and *kwic* systems

From the graph of Figure 5.3, we can observe that a high percentage of implementations of all functions indeed exhibit reasonably good levels of cohesion. In general, functional cohesion, sequential cohesion, and communicational cohesion are considered acceptable levels of cohesion. There is a small fraction of implementations with coincidental cohesion. The reasons for this could be one or more of the following: (i) poor design/implementation of the given specifications, (ii) possible side effects or poor information hiding in the implementation, and (iii) implementation limitations of *CMT* in canonicalizing identifiers representing function names and dealing with reference parameters (see discussion in Section 5.3). The data presented in Figure 5.3 is discussed further in Section 5.3.

5.3 Analysis of the data collected in the Experiments

In this section, we will analyze the data collected in *Experiments 2* and *3*. This is done by making some observations of the data and providing rationale for these observations.

Observation 1: The cohesion for 76%, 85%, and 54% implementations of function *main* of the three programs *lex.scheme, calc*, and *kwic*, respectively, is *undefined*. This ratio is much higher than for other functions, with the exception of the function *unget_error* of program *lex.scheme*.

When is cohesion *undefined*? What is the reason for such high *undefined* cohesion, especially for the function *main*?

CMT assigns an *undefined* cohesion to a function if it cannot recognize any variables that constitute as output variables from that function. A variable is treated as an output variable if it is a reference formal parameter or a global variable and there is at least one definition of that variable within the function. The identifier corresponding to the name of a function is also considered as an output variable if the function returns an explicit value using a *return* statement. Since the assignment of cohesion level to a function is mainly based on the interactions among output variables, the cohesion of a function with no output variables is considered *undefined*.

It is possible to provide some situations which will lead to the assignment of cohesion level of a function as *undefined*. The *main* function in a C program is a good candidate for the assignment of *undefined* cohesion level if the program does not declare any global variables. A function that merely uses the values of formal parameters and/or global variables without actually changing them is also a good candidate for *undefined* cohesion level. This happens in the case of functions that print the values of variables but do not change them. In addition, functions that have no formal parameters and do not modify any global variables, and functions with empty code sections are good candidates for the assignment of *undefined* level of cohesion.

Observation 2: The total number of functions with *undefined* cohesion in all the programs, see Figures 5.1, 5.2, and 5.3, range from 8.33% to 24%. The latter three versions of the *UEMACS* show 22% to 24% functions with *undefined* cohesion, whereas the functions with *undefined* cohesion in all the revisions of *SC* range from 10% - 14%.

Why is the number of functions with *undefined* cohesion so high for *UEMACS*?

In case of *UEMACS-3.7*, *UEMACS-3.8b*, and *UEMACS-3.9*, the percentage of functions in the *undefined* category is significantly high, approximately 24%. Upon studying the code of the corresponding functions, we found the following reason for this high number: In all three versions, there are forty functions with the peculiarity that each has just one statement, a call to another function. Each of these forty functions calls the same function with some differences in the parameters. Since all the formal parameters of the forty functions are passed by value, these variables do not constitute as output variables. Therefore, *CMT* cannot determine cohesion for these functions.

Observation 3: Having functions with *coincidental* cohesion is not considered to be good software engineering practice. Still 3% to 10% of functions in *Experiment 2* are *coincidental*.

Is this large number alarming? Do the functions identified to have *coincidental* cohesion really have *coincidental* cohesion?

From the analysis of our implementation of *CMT*, we can provide primarily two reasons why our tool assigns *coincidental* cohesion to a function incorrectly:

1) Limitations of algorithm for canonicalizing variables: At present, *CMT* canonicalizes only local variables. Global variables and formal parameters are not canonicalized. However, we had not taken appropriate steps not to canonicalize definitions made to the name of a function where a function returns explicit values using *return* statements. In such cases, there will be several output variables without much relationship among them, which leads to the assignment of coincidental cohesion.
2) An actual parameter corresponding to a formal reference parameter is considered as being defined at the call site whether or not it is actually modified in the called function. If this actual parameter is global in scope in the calling function or a formal parameter in the calling function, then it will be treated as an output variable in the calling function. Treatment of reference parameters in this manner may result in multiple output variables for a fucntion without meaningful dependencies between these output variables and/or

between these output variables and other output variables that are really modified. The relationships between such output variables tend to be coincidental.

Observation 4: About 65% of implementations of the function *evaluate_a_line* of the program *calc*, see Table 5.5, are assigned *procedural* cohesion. This is the only function in all the programs of *Experiment 3* with such high numbers of implementations with *procedural* cohesion.

The function *evaluate_a_line* is the only function of the *lex.scheme*, *calc*, and *kwic* systems exhibiting high number of implementations with procedural cohesion. Examination of the design specification of the *calc* system showed that this function performs a set of activities (e.g., set, print, quit) that appeared to be logically cohesive. However, the implementations of this function seem to be performing these activities within a common procedural unit like a "while" loop. Therefore, the tool assigns procedural cohesion to the implementations of the function *evaluate_a_line*.

Observation 5: About 75% implementations of the function *append_to_line* and 92% implementations of the function *circulate_and_add_line* are assigned *sequential* cohesion. Though quite a few of the implementations are assigned *sequential* cohesion, the ratios for these particular functions are high.

Examination of the specification of the function *append_to_line* showed that this function takes pointers to two strings and returns the result of concatenating these strings. The specification also indicates that the function may destroy the contents of one of these strings as a possible side-effect. From this specification, we may infer that most of the implementations did in fact change the contents of one of the strings and the changed string depended on the other string for concatenation. The changed string must have been returned using a *return* statement. Therefore, the output variables are those representing the function name and the changed string. The output variable representing the function name is then dependent on the changed string variable. This is clearly a case of *sequential* cohesion.

Twenty two implementations, out of a total of twenty four, of *circulate_and_add_line* function were assigned *sequential* cohesion by *CMT*. Stevens et al.'s definition would clearly assign sequential cohesion to this function since there are two activities and the output of one activity (circulate) forms the input to the other (add). Here, the assignment of sequential cohesion by *CMT* to this function clearly agrees with that of the function's specification, and the intuition behind the Stevens et al.'s definition of sequential cohesion.

Observation 6: About 71% to 81% of implementations in *Experiment 3* are assigned functional and sequential cohesion and only 7% to 13% implementations are assigned other levels, except *undefined*.

Given that these programs were developed based on a "sound" design, does the distribution convey anything about the cohesion measure?

Assuming that the design of the three systems is good and that the implementations do not disagree with the design, one may infer from the distribution levels of *Experiment 3* that an assignment of functional and sequential levels of cohesion, based on our measure, indicates a module with a good design.

Chapter 6
Related Work

In this chapter, we investigate recent attempts at defining an objective measure for module cohesion. Our focus will be on approaches to computing module cohesion for function-based software systems where the term *module* refers to a procedure or a function in a Pascal-like language. Patel [Patel92], Rising and Calliss [Rising92], and Embley and Woodfield [Embley87] have proposed measures and/or guidelines for computing cohesion of abstract data type based software systems where a module refers to a package-like construct in Ada. These latter approaches are not discussed here as the presented research deals only with function-based software systems.

6.1 Slice Based Cohesion Measures

In this section, we present some work on measures of cohesion based on program slicing and Weiser's slice based metrics.

Program *slicing* is a method of program reduction or decomposition originally proposed by Weiser [Weiser81, Weiser84]. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form of program, called a *slice*, that reflects the same subset of behavior. The desired subset of behavior is specified using a *slicing criterion*. A slicing criterion of a program P is a tuple $<i,V>$ where i denotes a specific statement number in P and V is a subset of variables in P. A slice S, of program P on slicing criterion $<i,V>$, is an executable program that is a subset of P and whose execution behavior is the same as what would be observed in the module P. Figure 6.1 shows the slice obtained for the module *sum_and_product1* of Figure 2.3 on slicing criterion <16,*sum*>.

Slices have since been used during software development in a number of ways. These include debugging [Weiser82, Weiser86], software maintenance [Gallagher89], program integration [Horwitz88], and software metrics [Longworth85, Thuss88, Ott89, Ott91]. These

applications use a restricted form of slicing in the sense that a slice is taken with respect to a variable that is *defined* at or *used* at a program point *p*, whereas Weiser's definition of slicing allows a program to be sliced with respect to a program point *p* and an arbitrary set of variables.

```
1     procedure sum_and_product1 (m: integer; var sum: integer);
2     var i: integer;
3     begin
4       sum := 0;
6       i := 1;
7       while i <= m do begin
8         sum := sum + i;
10        i := i + 1;
11      end;
12    end;
```

Figure 6.1. Slice of *sum_and_product1* with slicing criterion <16,*sum*>

6.1.1 Ott and Thuss

Ott and Thuss attempt to show that a connection exists between the intersection of slices generated from the output variables of a module and the cohesion of a module [Ott89, Thuss88]. They regrouped Stevens et al.'s seven levels of cohesion into four groups. The four levels of cohesion they chose to use and their equivalents in Stevens et al.'s definitions are:

Low Cohesion	-	Coincidental and Temporal
Control Cohesion	-	Logical and Procedural
Data Cohesion	-	Communicational
High Cohesion	-	Sequential and Functional

A variable is defined to be an output variable if it retains its value after the module has completed execution. Slices are obtained for the output variables at the end statement of a module. These slices are sometimes called *end-slices* [Lakhotia91a]. *Slice profiles*, developed by Longworth [Longworth85], were used as a convenient representation for revealing slice patterns within a module. A slice profile of a module is a table which includes one column for each output variable of the module. Only variable-referent executable

statements (*VRES*) are shown in a slice profile. A "|" in a cell indicates that the corresponding statement is included in the slice of the program with respect to the output variable listed for that column. Table 6.1 shows the slice profile for the procedure *sum_and_product1* of Figure 2.3.

Table 6.1 Slice profile for procedure *sum_and_product1*

Line	Sum	Prod	Statement
1			procedure sum_and_product1(m,n: integer; var sum,prod: integer);
2			var i,j: integer;
3			begin
4	|		i := 1;
5	|		sum := 0;
6	|		while i <= m do begin
7	|		sum := sum + i;
8	|		i := i + 1;
9			end;
10		|	j := 1;
11		|	prod := 1;
12		|	while j <= n do begin
13		|	prod := prod * j;
14		|	j := j + 1;
15			end;
16			end;

The level of cohesion of a module is then estimated by inspecting the intersection of the slices obtained for the output variables of the module. These levels of cohesion are defined as follows:

1) A module falls into *low cohesion* group if the intersection of slices of the output variables is empty. The procedure *sum_and_product1* of Figure 2.3 is an example of low cohesion module.
2) A module exhibits *control cohesion* if the intersection of slices of the output variables primarily contains control statements and definitions for the control variables. The module *sum_and_product2* of Figure 2.4 has control cohesion since the intersection of the slices generated from the output variables contains only a loop statement and statements that define loop variables.

3) A module tends to exhibit *data cohesion* if the intersection of slices of the output variables contains non-control variable data definitions. The procedure *sum_and_product3* of Figure 2.5 is an example of module with data cohesion. This module has three output variables *x*, *sum*, and *prod*. The module first defines the values of the array *X*. The computation of *sum* and *prod* depends on the array *X*. The intersection of the slices generated from the output variables contains the definition of array *X*.
4) A module exhibits *high cohesion* if the slice of one output variable is entirely contained in the slice of another output variable. The module *sum_and_average* of Figure 2.9 has two output variables *sum* and *average.* The slice with respect to the variable *average* totally contains the slice on variable *sum*. Therefore, the module has high cohesion.

Ott and Thuss's approach to determining module cohesion has the following limitations:

1) The definitions of control cohesion and data cohesion are subjective as they are not precisely defined. Ott and Thuss do not explain how the cohesion of a module is determined if the intersection of the slices of the output variables contains both control and non-control statements as well as data definitions, i.e., they do not consider the possibility that the processing elements of a module may be associated by more than one level of cohesion.
2) Assignment of cohesion to a module using their approach is not consistent with that of Stevens et al. Ott and Thuss classify the module *sum_and_product4* of Figure 2.6 as having control cohesion whereas Stevens et al. assign communicational cohesion (data cohesion in terms of Ott and Thuss) to the same module.
3) Ott and Thuss's classification of levels of cohesion does not preserve the linear order defined by Stevens et al. They classify temporal cohesion to be lower than logical cohesion whereas Stevens et al. classify logical cohesion to be lower than temporal cohesion.

4) The subset of language constructs considered in Ott and Thuss's approach does not include procedure calls which are crucial to the communication between the modules of a program.

6.1.2 Cohesion measures based on Weiser's slice based metrics

Weiser suggested the use of slicing as a basis for program metrics and proposed several slice-based metrics that provide information about the structuring of a program [Weiser81]. These metrics are called *Coverage, Overlap, Clustering, Parallelism,* and *Tightness*. These metrics are not proposed originally as cohesion metrics by Weiser. However, some of these metrics were later found to be indicators of module cohesion.

6.1.2.1 Longworth, Ott and Thuss

Weiser's slice based metrics were first applied to measuring cohesion by Longworth [Longworth85]. He found that *coverage*, *overlap*, and *tightness* appeared to be related to cohesion. Longworth applied these metrics to slices obtained using Weiser's definition of slicing. In most cases, these metrics were found to be indicators of cohesion with some exceptions. Longworth discovered the sensitivity of these metrics to the size and position of the processing elements in a module. In some situations, this caused relatively low metric values in modules having data or high cohesion.

Ott and Thuss [Ott91] attributed the sensitivity of these metrics to the absence of *used by* relationships of the sliced variable in the corresponding slice. Slices as defined by Weiser capture only the *uses* relationship of a sliced variable, i.e., they contain only those statements which may have an effect on the value of the sliced variable at the statement prescribed by the slicing criterion. These slices do not capture the *used by* relationship of a sliced variable, i.e., they do not contain the statements that may be affected by the value of the sliced variable. Ott and Thuss defined a new form of slicing to take into account both uses and used by relationships of a sliced variable. These slices are called *metric slices.* A metric slice for

the variable *v* in module *M*, denoted by *MSLICE(M,v)*, is defined in terms of *relevant slice* (or *backward slice*) for the variable *v*, denoted by *RSLICE(M,v)*, and *dependent slice* (or *forward slice*) for the variable *v*, denoted by *DSLICE(M,v)*. *RSLICEs* represent *uses* relationships and *DSLICEs* represent *dependent* or *used by* relationships. The *metric slice* for the variable *v* in module *M* is defined as

$$MSLICE(M,v) = RSLICE(M,v) \cup \quad DSLICE(M,v)$$

Ott and Thuss claim that metric slices can eliminate the sensitivity problems due to the size and positioning of the processing elements. They have also expanded the list of metrics to contain the following: *Coverage*, *Overlap*, *Tightness*, *Parallelism*, *MinCoverage*, and *MaxCoverage*. The definitions of *Coverage*, *Overlap*, *Tightness* and *Parallelism* are based on Weiser's original work [Weiser81] with some modifications made by Longworth [Longworth85]. The definitions of *MinCoverage* and *MaxCoverage* are due to Ott and Thuss [Ott91].

Table 6.2 provides the definitions of *Coverage*, *Overlap*, *Tightness*, *Parallelism*, *MinCoverage* and *MaxCoverage*. Let V_M be the set of all variables used by module *M* and let *V* be a subset of V_M containing only those variables considered in the metric computations. The variables $v_i \in V$ are enumerated and the symbol SL_i corresponds to the slice obtained for v_i. Let SL_{int} be the intersection of all the SL_i, that is

$$SL_{\text{int}} = \bigcap_{i=1}^{|V|} SL_i .$$

Coverage (C_M) is a comparison of the length of the slices to the length of the module. *Overlap* (O_M) is a measure of the average number of overlapping statements in all the slices considered. *Tightness* (T_M) is a measure of the number of statements included in every slice. *Parallelism* (P_M) measures the number of slices with few statements in common. *MinCoverage* (MIN_M) measures the length of the shortest slice as a ratio to the module length. *MaxCoverage* (MAX_M) measures the length of the longest slice as a ratio to the module length. All the metrics, except for P_M, range from 0 to 1. For all metrics, except

for P_M, higher metric values would indicate higher levels of cohesion. P_M ranges from 0 on up, with the assumption being that 0 indicates the highest level of cohesion.

Table 6.2 Definitions of slice based cohesion metrics

Metric	Definition
Coverage (C_M)	$C_M = \frac{1}{\|V\|}\sum_{i=1}^{\|V\|}\frac{\|SL_i\|}{\|M\|}$
Overlap (O_M)	$O_M = \frac{1}{\|V\|}\sum_{i=1}^{\|V\|}\frac{\|SL_{\text{int}}\|}{\|SL_i\|}$
Tightness (T_M)	$T_M = \frac{\|SL_{\text{int}}\|}{\|M\|}$
Parallelism (P_M)	$P_M = \|\{SL_i \text{ such that } \|SL_i \cap SL_j\| < \tau \text{ for all } j <> i\}\|$
MinCoverage (MIN_M)	$MIN_M = \frac{1}{\|M\|}\min_i \|SL_i\|$
MaxCoverage (MAX_M)	$MAX_M = \frac{1}{\|M\|}\max_i \|SL_i\|$

Ott and Thuss have illustrated, using an example program, how slice profiles along with slice based metrics may be used during maintenance process to indicate inappropriate modularization or portions of the code that may require more effort to be understood [Ott92a, Ott92c]. The slices obtained in these studies are *intermodule slices*. An intermodule slice is similar to an intramodule slice except that the slice is computed through any procedure invocations in the module being sliced [Horwitz90].

It is important to note that these slice based metrics are not true measures of cohesion, although they appear to be useful as indicators of cohesion. Values of these metrics that are between 0 and 1 only provide an indication of the level of cohesion if not an exact level as defined by Stevens et al. These metrics are useful in the absence of any more precise

techniques that can provide true measures of module cohesion while still maintaining the original intent of cohesion as defined by Stevens et al.

6.1.2.2 Bieman and Ott

Ott and Bieman studied the effects of software changes on module cohesion using a variation of metric slice called *metric data slice* or *data slice* [Ott92b]. A metric data slice contains, not statements, but data tokens, i.e., variable and constant definitions and references. They could not make any sound conclusions except that the effects of changes on module cohesion seemed to match their intuition concerning the expected effects on particular cohesion attributes.

Bieman and Ott [Bieman94] continued this work on cohesion using metric data slices. However, they concentrated on developing quantitative measures of functional cohesion, the most desirable of the seven cohesion categories. In order to measure the functional cohesion, they have defined the following terms: *data slice, slice abstraction, glue tokens*, and *super-glue tokens*. Using these terms, they have provided three measures of functional cohesion. These measures are called *strong functional cohesion* (*SFC*), *weak functional cohesion* (*WFC*), and *adhesiveness*. The rest of this section provides the details of these measures.

A *data slice* is defined as a sequence of data tokens (variable and constant definitions and references) contained within the slice of the variable being considered. A *slice abstraction* of a procedure *P*, *SA*(*P*), is a set of data slices of the output variables of the procedure. A *glue* token is a data token that lies on more than one data slice. A *super-glue* token is a data token that lies on all data slices in *SA*(*P*). The set of all glue tokens in procedure *P* is denoted by *G*(*SA*(*P*)). Similarly, the set of all super-glue tokens is denoted by *SG*(*SA*(*P*)). All super-glue tokens are also glue tokens, i.e., $SG(SA(P)) \subseteq G(SA(P))$. If $|SA(P)| \leq 2$, then $SG(SA(P)) = G(SA(P))$.

The three measure of functional cohesion *SFC*, *WFC*, and *adhesiveness* are now defined in terms of the above pieces of information. Let *tokens*(*p*) denote the set of data tokens in procedure *p*. Each appearance of a data token in a program is counted as a different token, and each token can be in more than one data slice.

The *strong functional cohesion* (*SFC*) is defined as the ratio of super-glue tokens to the total number of data tokens in a procedure *p*:

$$SFC(p) = \frac{|SG(SA(p))|}{|tokens(p)|}.$$

The *weak functional cohesion*(*WFC*) is defined as the ratio of glue tokens to the total number of tokens in a procedure *p*:

$$WFC(p) = \frac{|G(SA(p))|}{|tokens(p)|}.$$

The *adhesiveness* of a token is the relative number of slices that it glues together. The adhesiveness, α, of a token *t* in procedure *p* is defined as follows:

$$\alpha(t, p) = \frac{\#\ of\ slices\ in\ p\ containing\ t}{|SA(p)|} \quad if\ t \in G(SA(p))$$

$$\alpha(t, p) = 0 \quad if\ t \notin G(SA(p))$$

The overall adhesiveness, *A*, for procedure *p* is computed as the ratio of the amount of adhesiveness to the total possible adhesiveness:

$$A(p) = \frac{\sum_{t \in G(SA(p))} \#\ of\ slices\ containing\ t}{|tokens(p)|.|SA(p)|}$$

The value of these cohesion measures (*strong functional cohesion*, *weak functional cohesion*, and *adhesiveness*) range from zero to one. Bieman and Ott show analytically the following relationship between the three functional cohesion measures: $SFC(p) \leq A(p) \leq WFC(p)$. In their work, they concluded that a module with only coincidental cohesion, as per Stevens et al., will measure near zero for all the three of their measures. However, they did

not know how their measures would evaluate modules that fall into the other cohesion classes.

6.2 Emerson's Approach

In an effort to measure cohesion, Emerson defined a discriminant metric for module cohesion [Emerson84]. He grouped Stevens et al.'s seven levels of cohesion into three groups: Type I = {functional, sequential, communicational}, Type II = {procedural, temporal}, and Type III = {logical, coincidental}. In his approach, a program is represented as a reduced flow graph. A reduced flow graph F of a program M is constructed from the program's flow graph F' by deleting from F' any vertex corresponding to an executable statement of M which does not contain a reference to a variable. For every vertex x deleted from F', arcs terminated at x in F' now terminate at the successor vertex to x in F. For every variable i referenced in the reduced flow graph F of a program, a reference set R_i is defined. R_i is the set of vertices of F which refer to the *i-th* variable, assuming that the variables have been enumerated 1,2,...v. The cohesion of a reference set R_i, denoted by $k(R_i,F)$, is defined as follows:

$$k(R_i, F) = \frac{|R_i| \dim R_i}{|VF - \{T\}| \dim F}$$

where *dim* R_i is the number of elements in a maximal linearly independent set of complete paths which pass through the vertex set R_i, VF is the number of vertices of F, T is the terminal vertex, and *dim* F is the number of linearly independent paths in F.

Finally, the cohesion of the module M, denoted by $k(F)$, is defined to be the average cohesion of the reference sets as shown below:

$$k(F) = \frac{\sum_{i=1}^{v} k(R_i, F)}{v}$$

Intuitively, given that there are *m* maximally independent paths and *n* variables in a program, Emerson's approach associates a high value of cohesion if each path references all *n* variables. As the number of paths that do not reference one or more variables increases, the cohesion of the program decreases.

Emerson claims, without much empirical evidence, that his measure discriminates between the three groups of cohesion. He also provides boundary values between the three groups of cohesion.

The choice of flow graph as means to model the program and to determine the module cohesion seems to reduce the credibility of his measure. Program flow graphs only represent the sequence of execution in a program and not the control or data dependence between statements. Therefore, measuring cohesion, a concept that requires knowledge about the control and/or data dependencies in a program, based solely on the flow graph representation of a program, will lead to incorrect cohesion. Also, determining the cohesion of a module using his approach is not easy because of the ambiguous nature of the range values and the difficulties involved in getting meaningful values for these ranges for a given language.

6.3 Other work on cohesion measures

There are several other studies that examined cohesion indicators rather than attempting to measure cohesion directly [Troy81, Card86, Hutchens85, Selby91].

Troy and Zweben [Troy81], in their study of measuring the quality of structured designs, had investigated the following design features as measures of cohesion:

1. The number of effects listed in the design document;
2. The number of effects other than I/O errors;
3. The maximum fan-in to any one box in the structure chart;
4. The average fan-in in the structure chart; and
5. The number of possible return values.

Troy and Zweben did not find evidence of a clear relationships between these design features and the quality of the software. In their work, quality is measured by the number of source code modifications. They did not attempt to show a relationship between these design measures and cohesion. The results of their investigation may mean that cohesion is not related to number of source code modifications or that these measures are not indicative of cohesion.

Card, Church, and Agresti [Card86] conducted an empirical study of software design practices in a FORTRAN-based scientific computing environment. The practices examined affect module size, module cohesion, data coupling, descendant span, unreferenced variables, and software reuse. Measures characteristic of these practices were extracted from FORTRAN modules developed for five flight dynamics software projects monitored by Software Engineering Laboratory (SEL). The relationship of these measures to cost and fault rate was then analyzed. In their study of module cohesion as a design practice, each FORTRAN module was rated as performing as one or more of the following functions: input/output, logic/control, and/or algorithmic processing. Those modules described as having only one function were classified as high cohesion; those having two functions, medium cohesion; and those having three or more functions, low cohesion. They found that fifty percent of high-cohesion modules were fault free, whereas only eighteen percent of low-cohesion modules were fault free. They did not find any significant relationship between module cohesion and development cost. However, they concluded that developing high-cohesion modules is a good practice.

Selby and Basili [Selby91] used data bindings [Hutchens85], a measure based on data interactions, as the basis for calculating cohesion and coupling and analyzing system structure. Routines (main program, procedure, or function) are placed into clusters based on the data bindings. The coupling of a cluster with other clusters is determined. A ratio of the cluster coupling to the internal strength of a cluster is computed. Their experiment indicated that clusters with high coupling/strength ratio had the most errors and the highest error

correlation efforts. Selby and Basili, however, did not attempt to show a relationship between their measure and cohesion.

6.4 Comparison with Related Works

This section provides a comparison of our approach to other approaches discussed in related work using the sample programs listed in Chapter 2. Assignment of cohesion levels to these modules by Stevens et. al's, Lakhotia and Nandigam's, Ott and Thuss's (four levels of cohesion), and Emerson's approaches is provided in Table 6.3. Table 6.4 presents the assignment of cohesion levels to the same modules using Ott and Thuss's (based on metric slices), Ott and Bieman's (based on metric data slices), and Bieman and Ott's (based on functional cohesion measures) approaches. A (*) in the following tables indicates that the cohesion assigned to that module differs from that assigned by Stevens et al.

Table 6.3 Comparison of various cohesion measures - Part I

Approach → Module ↓	**Stevens et al.**	**Lakhotia and Nandigam**	**Ott and Thuss (1988-89)**	**Emerson**
sum_and_product1	coincidental	coincidental	low	Type II (*)
sum_and_product2	procedural	procedural	control	Type II
sum_and_product3	communicational	communicational	data	Type II (*)
sum_and_product4	communicational	communicational	control (*)	Type II (*)
sum_or_product1	logical	logical	control	Type III
sum_or_product2	logical	logical	low (*)	Type III
sum_and_average	sequential	sequential	high	Type II (*)
compute_sum	functional	functional	high	Type I

Table 6.4 Comparison of various cohesion measures - Part II

Approach → Module ↓	**Ott and Thuss (based on metric slices;) 1991-92**					**Ott and Bieman (based on metric data slices) 1992**					**Bieman & Ott (functional cohesion) 1994**		
	C	*O*	*T*	*Min*	*Max*	*C*	*O*	*T*	*Min*	*Max*	*SFC*	*WFC*	*A*
sum_and_product1	0.5	0.0	0.0	0.5	0.5	0.5	0.0	0.0	0.5	0.5	0.0	0.0	0.0
sum_and_product2	0.71	0.6	0.43	0.71	0.71	0.71	0.6	0.43	0.71	0.71	0.43	0.43	0.43
sum_and_product3	0.57	0.51	0.29	0.49	0.64	0.6	0.51	0.3	0.49	0.65	0.3	0.58	0.49
sum_and_product4	0.71	0.6	0.43	0.71	0.71	0.71	0.58	0.42	0.71	0.71	0.42	0.42	0.42
sum_or_product1	0.55	0.16	0.09	0.55	0.55	0.56	0.22	0.13	0.56	0.56	0.13	0.13	0.13
sum_or_product2	0.5	0.0	0.0	0.5	0.5	0.53	0.11	0.06	0.53	0.53	0.06	0.06	0.06
sum_and_average	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
compute_sum	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

It can be seen from the Table 6.3 that the assignment of cohesion to modules using our approach is consistent with that of Stevens et al. The data in Table 6.4 is difficult to interpret and determining the level of cohesion of a module based on these numbers is not objective, except when the value of the metric is either 1.0, indicating very high cohesion, or 0.0, indicating very low cohesion.

Chapter 7
Research Contributions and Future Work

7.1 Research Contributions

This dissertation makes the following contributions to the broad area of software engineering and to the field of software metrics in particular:

1) An objective measure for determining the cohesion of a module has been proposed. It has been shown that the proposed measure attempts to preserve the intent of the original definitions of the same by Stevens et al.
2) The proposed measure for cohesion is algorithmically computable. To show that this is indeed the case, a software tool, named *CMT* (for *Cohesion Measurement Tool*), has been developed. To demonstrate that *CMT* is not a tool that can process only toy-like programs and can, in fact, handle large programs, we have tested the tool on industrial-strength software systems obtained from public domain sources.
3) Realizing the importance of empirical validation in software engineering and software metrics, this research has attempted to validate the proposed measure using a controlled experiment. The initial results of the data analysis of the experiment data are very encouraging. The experience gained from the design and analysis of this experiment will have profound impact on continuing and improving the similar kind of empirical validation involving our measure for module cohesion.
4) The proposed measure for module cohesion is language-independent and therefore makes it applicable to most procedural languages.

7.2 Future Work

There are several promising directions for continuing the work accomplished in this research. The possible directions for extending this work broadly fall into five areas:

1) Better measures - the proposed rules for various levels of cohesion can be analyzed formally and improved.
2) Language tools - development of metric tools for measuring the cohesion of programs written in languages such as Pascal, FORTRAN, etc.
3) Program decomposition - techniques and tools to decompose a module which is not cohesive into two or more sub-modules with higher cohesion than the parent module. The challenging task here is to generate sub-modules that are executable.
4) Empirical studies - design of controlled experiments to investigate the effects of module cohesion on such external product attributes as modifiability, maintainability, reliability, etc.
5) Explanation capabilities - facilities for the explanation of why a certain cohesion level was assigned to a module. This may encourage the programmer or maintainer of the module to rewrite the module to improve the cohesion of the module.

References

[Aho86] Aho, A. V., Sethi, R. and Ullman, J. D., *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Baker90] Baker, A. L., Bieman J. M., Fenton, N., Gustafson, D. A., Melton, A. and Whitty, R., "A Philosophy for Software Measurement," *Journal of Systems Software*, Vol. 12, 1990, pp 277-281.

[Bieman94] Bieman, J. M. and Ott, L. M., "Measuring Functional Cohesion," *IEEE Transactions on Software Engineering*, Vol. 20, No. 8, August 1994, pp 644-657.

[Brooks80] Brooks, R., "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," *Communications of ACM*, Vol. 23, No. 4, April 1980, pp 207-213.

[Card86] Card, D. N., Church, V. E. and Agresti, W. W., "An empirical study of software design practices," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 264-271.

[Conte86] Conte, S. D., Dunsmore, H. E. and Shen, V. Y., *A Software Engineering Metrics and Models*, Benjamin / Cummings, 1986.

[Couch87] Couch, J. V., *Fundamentals of Statistics for the Behavioral Sciences*, West Publishing Company, 1987.

[Embley87] Embley, D. W. and Woodfield, S. N., "Cohesion and Coupling for Abstract Data Types," *Proc. 6th Phoenix Conf. on Computers and Communications*, Phoenix, Arizona, Feb. 1987, pp 229-234.

[Emerson84] Emerson, T. J., "A Discriminant Metric for Module Cohesion," *7th International Conference on Software Engineering*, March 1984, pp 294-303.

[Fenton91] Fenton, N. E., *Software Metrics - A Rigorous Approach*, Chapman & Hall, 1991.

[Ferrante87] Ferrante, J., Ottenstein, K. J. and Warren, J. D., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp 319-349.

[Gallagher89] Gallagher, K. B., *Using Program Slicing in Software Maintenance*, PhD thesis, University of Maryland, Baltimore, Maryland, December 1989.

[Goradia93] Goradia, T. S., *Dynamic Impact Analysis: Analyzing Error Propagation in Program Executions*, Ph.D. Thesis, Dept. of Computer Science, New York University, November 1993.

[Hecht77] Hecht, M. S., *Flow Analysis of Computer Programs*, North-Holland, Inc., 1977.

[Horwitz88] Horwitz, S., Prins, J. and Reps, T., "Integrating Non-Interfering Versions of Programs," *Conf. Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1988.

[Horwitz90] Horwitz, S., Reps, T. and Binkley D., "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, January 1990, pp 26-60.

[Hutchens85] Hutchens, D. H. and Basili, V. R., "System Structure Analysis: Clustering with Data Bindings," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 8, August 1985, pp. 749-757.

[Karstu94] Karstu, S., *An Examination of the Behavior of Slice Base Cohesion Measures*, Master's Thesis, Michigan Technological University, Department of Computer Science, August 1994.

[Lakhotia93] Lakhotia, A, "Ruled-based approach to computing module cohesion," *15th International Conference on Software Engineering*, May 1993.

[Lakhotia91a] Lakhotia, A, *Insights into relationships between end-slices*, CACS TR-91-5-3, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, La., September 1991.

[Lakhotia91b] Lakhotia, A. and Nandigam, J., *Computing Module Cohesion*, CACS TR-91-5-4, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, La., November 1991.

[Lengauer79] Lengauer, T. and Tarjan, R. E., "A fast algorithm for finding dominators in a flow graph," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, July 1979, pp 121-141.

[Lewis91] Lewis, T. G., *CASE: Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1991.

[Longworth85] Longworth, H. D., *Slice Base Program Metrics*, Mater's Thesis, Michigan Technological University, 1985.

[Mosteller67] Mosteller, F., Rourke, R. E. K. and Thomas, G. B., *Probability and Statistics*, Addison-Wesley Publishing Company, 1967.

[Myers75] Myers, G. J., *Reliable Software through Composite Design*, Petrocelli / Charter, 1975.

[Myers78] Myers, G. J., *Composite / Structured Design*, Van Nostrand Reinhold Company, 1978.

[Newmark92] Newmark, J., *Statistics and Probability in Modern Life*, 5th edition, Sanders College Publishing, A Harcourt Brace Jovanovich College Publisher, 1992.

[Ott89] Ott, L. M. and Thuss, J. J., "The Relationship between Slices and Module Cohesion," *Proc. 11th International Conference on Software Engineering*, May 1989, pp 198-204.

[Ott91] Ott, L. M., *Slice Based Metrics for Estimating Cohesion*, Technical Report CS-TR 91-04, Department of Computer Science, Michigan Technological University, November 1991.

[Ott92a] Ott, L. M., *Using Slice Profiles and Metrics during Software Maintenance*, Technical Report CS-TR 92-02, Department of Computer Science, Michigan Technological University, January 1992.

[Ott92b] Ott, L. M. and Bieman, J. M., *Effects of Software Changes on Module Cohesion*, Technical Report CS-TR 92-06, Department of Computer Science, Michigan Technological University, March 1992.

[Ott92c] Ott, L. M. and Thuss J. J., *Using Slice Profiles and Metrics As Tools in the Production of Reliable Software*, Technical Report CS-TR 92-08, Department of Computer Science, Michigan Technological University, April 1992.

[Page-Jones88] Page-Jones, M., *The Practical Guide to Structured Systems Design*, 2nd Edition, Yourdon Press Computing Series, 1988.

[Patel92] Patel S., Chu, W. and Baxter, R., *A Measure for Composite Module Cohesion*, Lockheed Software Technology Center, Lockheed Palo Alto Research Laboratories, Orgn. 96-10, Bldg. 254E, 3251 Hanover Street, Palo Alto, CA, 1992.

[Pressman92] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, 3ed, McGraw Hill, 1992.

[Reasoning85] *REFINE User's Guide*, Reasoning Systems, Palo Alto, CA, 1985.

[Reasoning89] *DIALECT User's Guide*, Reasoning Systems, Palo Alto, CA, 1989.

[Reasoning91] *INTERVISTA User's Guide*, Reasoning Systems, Palo Alto, CA, 1991.

[Reasoning92a] *REFINE/C User's Guide*, Reasoning Systems, Palo Alto, CA, 1992.

[Reasoning92b] *REFINE/C Programmer's Guide*, Reasoning Systems, Palo Alto, CA, 1992.

[Reasoning92c] *YOYO control-flow-graph system*,version 2, Reasoning Systems, Palo Alto, CA, 1992.

[Reasoning92d] *REFINE/C extension for generating control-flow graphs*, Reasoning Systems, Palo Alto, CA, 1992.

[Rising92] Rising, L and Calliss, F. W., "Problems with determining package cohesion and coupling," *Software - Practice and Experience*, Vol. 22(7), July 1992, pp 553-571.

[Selby91] Selby, R. W., and Basili, V. R., "Analyzing Error-Prone System Structure," *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, February 1991, pp. 141-152.

[Sommerville89] Sommerville, I., *Software Engineering*, 3 ed, Addison Wesley, 1989.

[Stevens74] Stevens, W. P., Myers, G. J. and Constantine, L. L., "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, May 1974.

[Sundaresan94] Sundaresan, G., *Constructing Control Dependence Graphs*, Course Project, The University of Southwestern Louisiana, Lafayette, La, 1994.

[Thuss88] Thuss, J. J., *An Investigation into Slice Based Cohesion Metrics*, Master's Thesis, Michigan Technological University, 1988.

[Troy81] Troy, D. A., and Zweben, S. H., "Measuring the Quality of Structured Designs," *The Journal of Systems and Software*, Vol. 2, pp 113-120, 1981.

[Weiser81] Weiser, M., "Program Slicing," *Proc. 5th International Conference on Software Engineering*, March 1981, pp 439-449.

[Weiser82] Weiser, M., "Programmers use slicing when debugging," *Communications of ACM*, Vol. 25, July 1982, pp 446-452.

[Weiser84] Weiser, M., "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp 352-357.

[Weiser86] Weiser, M. and Lyle, J., "Experiments on Slicing-Based Debugging Aids," In Soloway, E. and Iyengar, S. (Editors), *Empirical Studies of Programmers*, Ablex Publishers, Norwood, NJ, 1986.

[Winer71] Winer, B. J., *Statistical Principles in Experimental Design*, 2ed, McGraw-Hill Book Company, 1971.

[Yourdon78] Yourdon, E. and Constantine, L. L., *Structured Design*, Yourdon Press, 1978.

[Yourdon79] Yourdon, E. and Constantine, L. L., *Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Inc., 1979.

[Zuse91] Zuse, H., *Software Complexity - Measures and Methods*, Walter De Gruyter, 1991.

Appendix A
Cohesion Measurement Tool (CMT)

We have developed a tool, named *CMT* (*Cohesion Measurement Tool*), that determines the cohesion of functions in a C program. The CMT is a robust tool that has been tested on several large software systems. This chapter describes the architecture of *CMT*, its subcomponents, and an example session with *CMT* to demonstrate its use.

A.1 Architecture of the *CMT*

The *CMT* has been implemented on Software Refinery using the REFINE programming language [Reasoning85]. It consists of the following components: *Data-flow Analyzer*, *Variable Canonicalizer*, *Control-dependence Analyzer*, *VDG Constructor*, and *Cohesion Analyzer*. Additionally, it also uses the REFINE/C *interactive workbench* and the REFINE/C extension for *Control-flow graphs* provided by Reasoning Systems, Inc. [Reasoning92a, Reasoning92b, Reasoning92c, Reasoning92d]. Figure A.1 gives the data flow architecture of *CMT*.

The REFINE language provides an integrated treatment of set theory, logic, transformation rules, pattern matching, and procedure. It provides a powerful object base and primitives for manipulating objects in the object base. This object base is used to store REFINE programs and other software-related objects from application domain. The Software Refinery environment provides development tools such as the command interface, the context mechanism, the on-line help system, browser/editor for the object base, interactive user interfaces for REFINE applications, customizable lexical analyzers and parsers, and the debugging system [Reasoning89, Reasoning91].

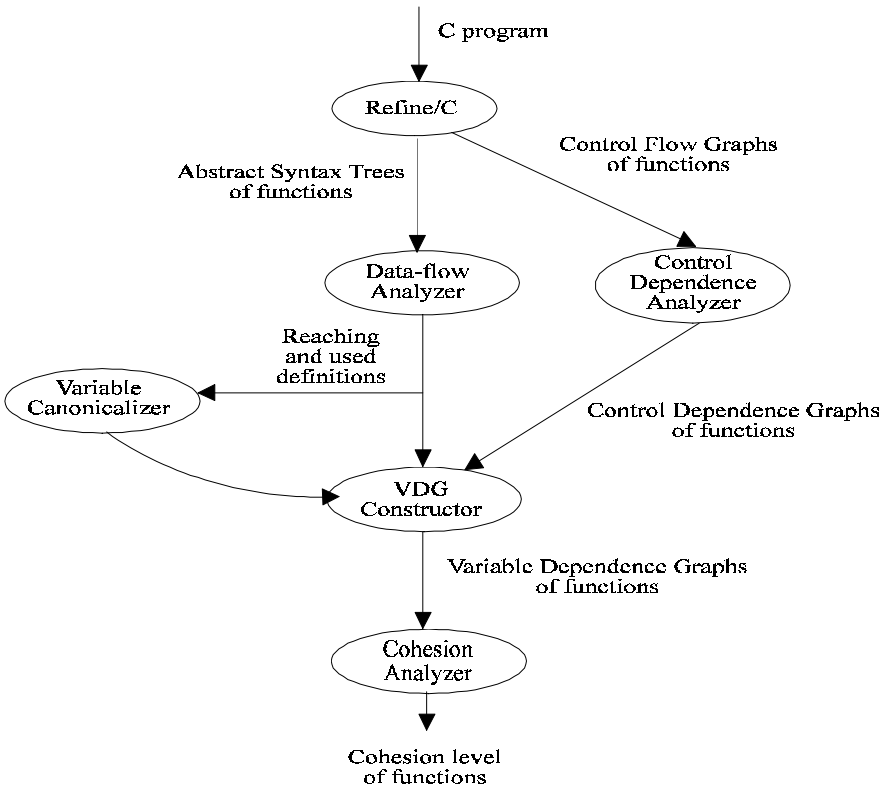

Figure A.1 The Data Flow Architecture of *CMT*

A.2 Components of the *CMT*

This section describes the following components of *CMT*: REFINE/C interactive workbench, REFINE/C control-flow graph generator, control dependence analyzer, data-flow analyzer, variable canonicalizer, *VDG* constructor, cohesion evaluator, and user interface.

A.2.1 REFINE/C Interactive Workbench

REFINE/C is an extensible, interactive workbench used to understand, analyze, evaluate, and redocument existing C programs. REFINE/C performs the following actions when analyzing a C program:

1) reads the C source code and represents the program as an abstract syntax tree (AST) in REFINE's object base,
2) analyzes the AST and stores analysis results in a symbol table format.

REFINE/C also provides an Application Programmer's Interface (API) so that one can use its reverse engineering features to build customized analysis tools. With the help of API, *CMT* uses REFINE/C to achieve the first of the actions listed above. The second action performed by REFINE/C is replaced with CMT's own analysis components.

Figure A.2 shows how REFINE/C represents the following C function definition in the form of an abstract syntax tree (AST):

```
int fact(n)
int n;
{
      if (n < 2)
        return 1;
      else
        return n * fact(n-1);
}
```

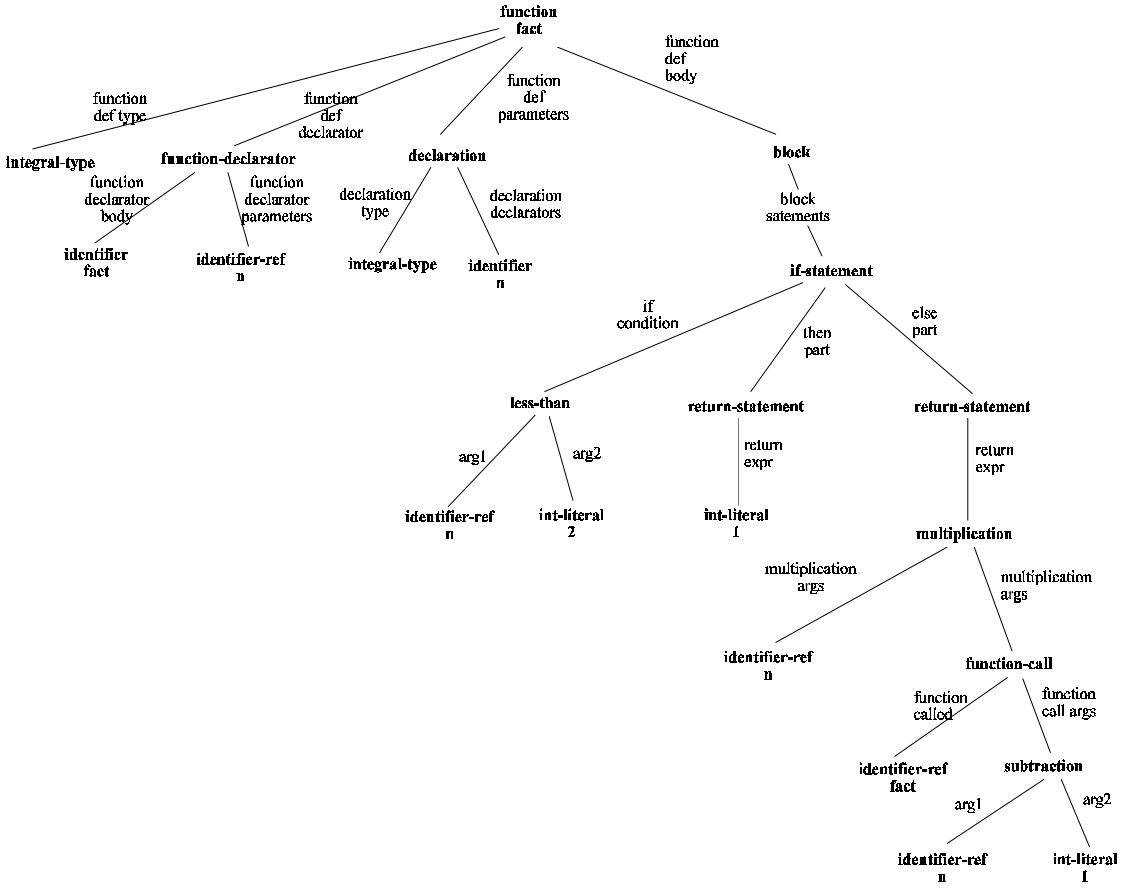

Figure A.2 Function *fact* and its Abstract Syntax Tree

A.2.2 REFINE/C Cfg Generator

REFINE/C *control-flow-graph generator* is an extension of REFINE/C that generates intraprocedural control-flow graphs for C functions [Reasoning92d]. REFINE/C *cfg generator* works on top of CFG system provided by Reasoning Systems [Reasoning92c]. The CFG system is a language-neutral system for representing and analyzing control-flow information about programs. The CFG system represents a control-flow graph as a network of REFINE objects. There are four main object classes used in representing control-flow graphs: CONTROL-FLOW-GRAPH, CFG-NODE, CFG-EDGE, and CFG-CONDITION. These are used to represent, respectively, the following: (1) an entire control-flow graph, (2) a node (basic block) in a control-flow graph, (3) an edge between two nodes in a control-flow graph, and (4) the condition under which a CFG-EDGE is taken to exit a CFG-NODE.

Figure A.3 shows the control-flow graph generated by REFINE/C Cfg Generator for the following C function that computes the sum of integers 1 through *n*:

```
1     void compute_sum(int *s, int n)
2     {
3           int i;
4
5           *s = 0;
6           for (i = 0; i < n; i++)
7             *s = *s + i;
8     }
```

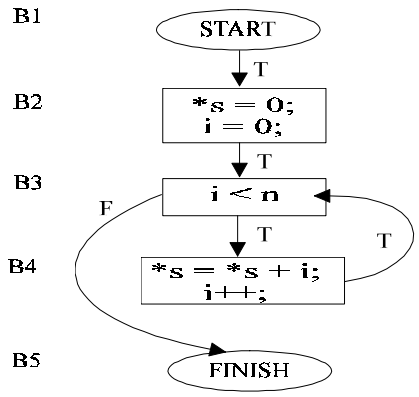

Figure A.3 Function *compute_sum* and its control flow graph

A.2.3 Control Dependence Analyzer

The *control dependence analyzer* constructs control dependence graphs of functions using as input the control-flow graphs generated by the REFINE/C Cfg Generator. The control dependence analyzer, developed by Sundaresan [Sundaresan94], implements the algorithms presented by Lengauer et al. [Lengauer79] and Ferrante et al. [Ferrante87] to construct post dominator trees and control dependence graphs. The algorithm by Lengauer et al. [Lengauer79] requires modifying the control-flow graph generated by the REFINE/C Cfg Generator by adding a special node called *entry node* and two edges. One edge is added from the *entry* node to the *start* node and is labeled *true*. The second edge is added from the *entry* node to the *finish* node and is labeled *false*. A control dependence graph is then constructed based on the modified control flow graph. Figure A.4 shows the control-flow graph of Figure A.3 with the modifications and the corresponding control dependence graph.

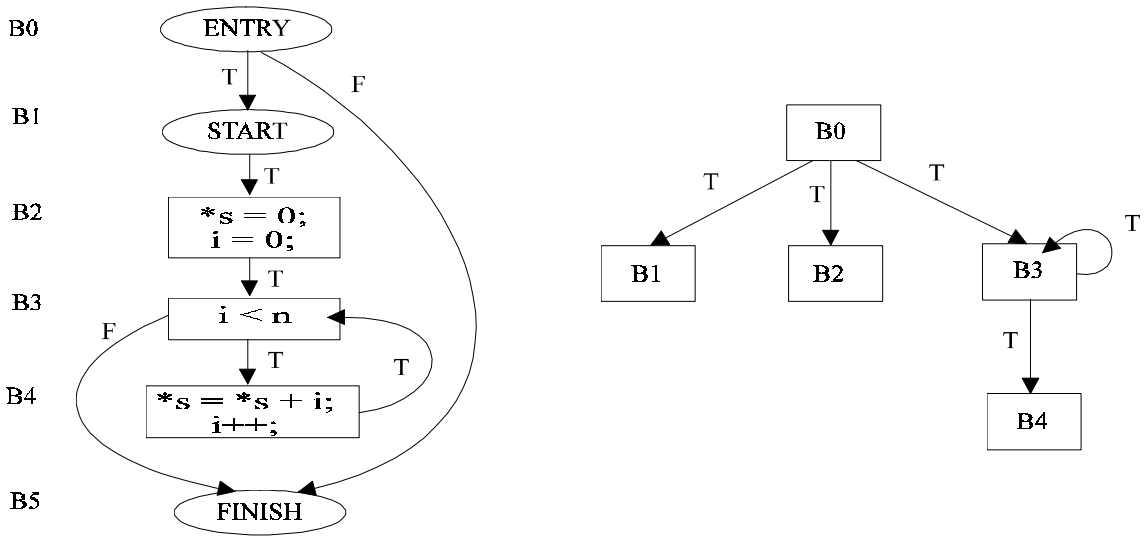

Figure A.4 A control flow graph and the corresponding control dependence graph

A.2.4 Data-flow Analyzer

The data-flow analyzer of CMT performs the following functions: (i) identifies the definitions created within each basic block of a control-flow graph, (ii) implements the

iterative algorithm by Aho et al. for computing reaching definitions [Aho86], (iii) identifies definitions that are used in the definition of a variable, and (iii) identifies definitions that are transitively used in the definition of a variable.

As an example, Table A.1 shows, for the control-flow graph in Figure A.4, the definitions created within each block (*DEFS*), the definitions that reach the top of each block (*RD-IN*), and the definitions that reach the bottom of each block (*RD-OUT*). The data-flow analyzer keeps track of more information about the definitions than could be presented in the example without affecting clarity. This additional information includes the identification of the function in which the definition was created, the expression which contains the definition, whether the definition is local or global to the function, the line number of the program statement that contains the definition, etc. In Table A.1, each tuple, representing a definition, contains three fields: (i) variable defined, (ii) defining expression, and (iii) number of source-level program statement that defines the variable. A "?" denotes that the corresponding field is irrelevant to that definition.

Table A.1. A subset of information generated by the data-flow analyzer of *CMT*

Basic Block Id	RD-IN	DEFS	RD-OUT
B0	{}	{(s,?,1),(n,?,1)}	{(s,?,1),(n,?,1)}
B1	{(s,?,1),(n,?,1)}	{}	{(s,?,1),(n,?,1)}
B2	{(s,?,1),(n,?,1)}	{(s,*s=0,5),(i,i=0,6)}	{(s,*s=0,5),(i,i=0,6), (n,?,1)}
B3	{(n,?,1),(s,*s=0,5), (s,*s=*s+i,7), (i,i=0,6),(i,i++,6)}	{}	{(n,?,1),(s,*s=0,5), (s,*s=*s+i,7), (i,i=0,6),(i,i++,6)}
B4	{(n,?,1),(s,*s=0,5), (s,*s=*s+i,7), (i,i=0,6),(i,i++,6)}	{(s,*s=*s+i,7), (i,i++,6)}	{(s,*s=*s+i,7), (n,?,1),(i,i++,6)}
B5	{(n,?,1),(s,?,1), (s,*s=0,5), (s,*s=*s+i,7), (i,i=0,6),(i,i++,6)}	{}	{(n,?,1),(s,?,1), (s,*s=0,5), (s,*s=*s+i,7), (i,i=0,6),(i,i++,6)}

A.2.5 Variable Canonicalizer

The *variable canonicalizer* is responsible for canonicalizing variables in a program. A variable is canonicalized if all definitions of the variable are related, i.e., they are defined to achieve a single purpose. A module in which every variable is canonicalized is said to be a canonicalized module.

The variable canonicalizer performs this task by grouping definitions of a variable within a module into one or more sets such that all definitions belonging to a set are reachable at one or more uses of that variable. A variable is canonicalized if there exists only one such set for that variable. A variable is not canonicalized if there are multiple sets of definitions for that variable. Once this partitioning of definitions of variables is done, a variable can be easily canonicalized by replacing, each occurrence of the variable from a set with a unique name. This replacement will not affect the functionality of the module. The replacement of variables with unique names is not done by changing the source code, but by maintaining the information of the set to which a particular definition belongs to. The variable canonicalizer implements the algorithms presented in Figure 4.22 for canonicalizing variables. For example, consider the C function in Figure A.5:

```
1     compute_sum_and_prod(int m,int n,int *sum,int *prod)
2     {
3           int i;
4
5           *sum = 0;
6           for (i = 1; i <= m; i++)
7             *sum = *sum + i;
8           *prod = 1;
9           for (i = 1; i <= n; i++)
10            *prod = *prod * i;
11    }
```

Figure A.5 An example C function that computes sum and product of numbers

In the above module, the instances of variable *i* involved in the computation of sum are not related to the instances of variable *i* involved in the computation of product. We can

safely replace every instance of the variable *i* involved in the computation of product, for example, with a unique variable without affecting the functionality of the module. *CMT* does not necessarily do the renaming of variables, but keeps information to distinguish the uses of variable *i* involved in the computation of sum from the uses of variable *i* involved in the computation of product.

A.2.6 VDG Constructor

The *VDG Constructor* uses control-flow, data-flow, and variable canonicalization information to represent data and control dependencies between the variables of a module in the form of a variable dependence graph (*VDG*). In a VDG, the nodes represent the variables and edges represent data and/or control dependencies between the variables. The VDG Constructor also performs interprocedural analysis by propagating any dependencies that exist between the formal parameters of a called function as data dependencies between the corresponding actual parameters at the call site.

Figure A.6 shows the VDG of function *compute_sum_and_prod* of Figure A.5. The effect of variable canonicalization can be seen in Figure A.6 in the case of variable *i* which was used in computing both sum and product. The variable canonicalizer determines that the variable *i* in the computation of sum is different from the variable *i* in the computation of product. This information is used by the VDG Constructor to create two separate nodes for the variable *i*.

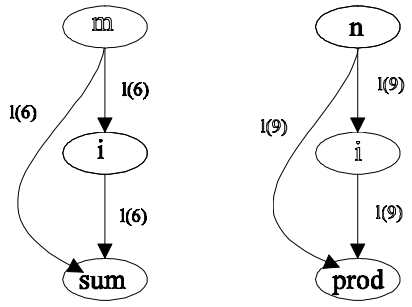

Figure A.6 Variable dependence graph for function *compute_sum_and_prod*

A.2.7 Cohesion Analyzer

The *cohesion evaluator* applies our rules of cohesion to the variable dependence graph of a module and determines the cohesion of that module. The two REFINE functions shown below encode cohesion rules for sequential and communicational cohesion levels. The function *sequential-cohesion?* takes two REFINE objects of type *vdg-node* and returns *true* if there is a data or control dependence edge from one node to another node; it returns *false* otherwise.

Rule for Sequential Cohesion: $x \rightarrow y \vee\ y \rightarrow x$ (see Table 3.1)

```
function sequential-cohesion? (node-x: vdg-node,
                               node-y: vdg-node) : boolean =
   (ex (x) (x in vdg-node-out-edges(node-x) &
            vdg-edge-to(x) = node-y))
   or-else
   (ex (x) (x in vdg-node-out-edges(node-y) &
            vdg-edge-to(x) = node-x))
```

The function *communicational-cohesion?* also takes two REFINE objects of type vdg-node and returns *true* if both these nodes have a data dependence on a same third node or a third node has data dependence on both of these nodes.

Rule for Communicational Cohesion:

$$\exists z(z \xrightarrow{D} x \wedge z \xrightarrow{D} y) \vee (x \xrightarrow{D} z \wedge y \xrightarrow{D} z) \quad \text{(see Table 3.1)}$$

```
function communicational-cohesion? (node-x: vdg-node,
                                    node-y: vdg-node) : boolean =
   (ex (x,y) (x in vdg-node-out-edges(node-x) &
              data-edge(x) &
              y in vdg-node-out-edges(node-y) &
              data-edge(y) &
              vdg-edge-to(x) = vdg-edge-to(y)))
   or-else
   (ex (x,y) (x in vdg-node-in-edges(node-x) &
              data-edge(x) &
              y in vdg-node-in-edges(node-y) &
              data-edge(y) &
              vdg-edge-from(x) = vdg-edge-from(y)))
```

A.2.8 CMT's User Interface

In *CMT*, the top-level description of a C program is defined as an instance of the object class *program*, typically in ".prog" file. The ".prog" file is similar to a *make* file that C programmers use to specify which C source files are compiled and linked to form an executable. The ".prog" file specifies a program name, a directory, and a set of C source files in that directory. REFINE/C provides two functions, called *parse-program-file* and *parse-program*, to build a program object and analyze the individual C source files specified in the program definition. An example of a program definition in ".prog" file is shown below:

```
program prog1
     directory "~jn/TestCases/"
     files "main.c"; "sum.c"
```

The *CMT*'s consists of a collection of functions that can be invoked to get such information as control-flow graph of a function, control dependence graph of a function, reaching definitions for each block of a control-flow graph of a function, variable dependence graph of a function, and variable dependence graphs of all functions in a program. The tool also provides utilities to dump variable dependence graph and cohesion level information of each function in a C program to external files.

A.3 An Example Session with CMT

In this section, we will analyze a C program with *CMT* to determine the cohesion functions defined in a program. A typical session with CMT starts with the invocation of REFINE. The customized Emacs makes editing REFINE programs more convenient and allows invoking REFINE compiler using editor commands. The REFINE command interface uses ".>" as its prompt. The example program to be analyzed by CMT is shown below:

```
#include <stdio.h>

void compute_sum(int *s, int n);
void compute_prod(int *p, int n);

int sum, prod;

main()
{
      int m,n;
      m = n = 5;
      compute_sum(&sum,m*n);
      compute_prod(&prod,n);
      printf("%d %d \n",sum,prod);
}

void compute_sum(int *s, int n)
{
      int i;
      *s = 0;
      for (i=1; i<=n; i++)
         *s = *s + i;
}

void compute_prod(int *p, int n)
{
      int i;
      *p = 1;
      for (i=1; i<=n; i++)
         *p = *p * i;
}
```

The following sequence of commands show how to start REFINE, load REFINE/C, load the control flow graph system, load REFINE/C CFG extension, and load *CMT* system.

```
%login
% start REFINE
   .> (load-system "c" "1-0")
   .> (load "<reasoning dir>/yoyo/control-flow-graph/v2/load.lisp")
   .> (load "<reasoning dir>/c/1-0/cfg-extension/ccat-package")
   .> (load "<reasoning dir>/c/1-0/cfg-extension/cfgs")
   .> (load "~jn/Cohesion/load.lisp")
```

The following sequence of commands to REFINE command interface illustrates the steps needed to process a C program and view the resulting variable dependence graph and cohesion level of functions in the program. In this session, we assume that the source code of our example C program is in a file named *prog7.c* in the directory *~jn/TestCases* and the program definition for this program is in a file named *prog7.prog* in the same directory. The REFINE function *test-prog* accepts a program definition file as input, parses the program,

analyzes all the functions in the individual files of the program, and returns a *program* object. The information resulting from analysis performed by different components of CMT is collected and maintained as a number of attributes defined on the program object.

```
.> (test-prog "~jn/TestCases/prog7")

.> (show-vdg  last-pgm  'cls::|compute_sum| )

     Vdg:                compute_sum
     Vdg Nodes:          s, i, n
     Vdg Edges:          (i, s, loop, 21)
                         (n, s, loop, 21)
                         (n, i, loop, 21)
     Output Variables:   s
     Module Cohesion:    Functional

.> (show-vdg last-pgm 'cls::|compute_prod| )

     Vdg:                compute_prod
     Vdg Nodes:          p, i, n
     Vdg Edges:          (i, p, loop, 29)
                         (n, p, loop, 29)
                         (n, i, loop, 29)
     Output Variables:   p
     Module Cohesion:    Functional

.> (show-vdg last-pgm 'cls::|main| )

     Vdg:                main
     Vdg Nodes:          sum, prod, m, n
     Vdg Edges:          (n, prod, data)
                         (n, sum, data)
                         (n, prod, data)
     Output Variables:   sum, prod
     Module Cohesion:    Communicational

.> (dump-all-vdg last-pgm "~jn/TestCases/prog7.vdgs")

.> :exit
```

The cohesion of the function *compute_sum* is functional because this module has only one output variable, namely *s*. The cohesion of the function *compute_prod* is also functional

because there is only one output variable, namely *p*. The cohesion of the function *main* is communicational because it has two output variables, *sum* and *prod*, and both are data dependent on a common variable *n*. Further, a loop dependence between the formal parameters *s* and *n* of the function *compute_sum* is used to establish a data dependence between the corresponding actual parameters *sum*, *m*, and *n*, at the call site in the calling function *main*. Similar dependence has been established at the call site for the function *compute_prod*. In fact, in our example program, all the dependencies between the variables of the VDG of *main* are established through the interprocedural analysis and propagation of the dependencies found between the formal parameters of functions *compute_sum* and *compute_prod*.

Appendix B

B.1 Processing Element Information for Programs in Experiment 1

Tables B.1 through B.4 provide information about the number of processing elements in each of the functions contained in programs P-1 through P-4 of Experiment 1, the highest cohesion between each processing element pair, and the cohesion of the function/module.

Table B.1 Processing element information for the Expression Evaluation Program (P-1)

Program Code	Function Name	No. of Processing Elements (PE)	No. of PE Pairs	Highest Cohesion between PE pairs	Module Cohesion
P-1	compute	1	0	{}	Functional
	operand_value	1	0	{}	Functional
	get_token	2	1	{Sequential}	Sequential
	evaluate	1	0	{}	Functional
	main	1	0	{}	Functional

Table B.2 Processing element information for the Tax Form Program (P-2)

Program Code	Function Name	No. of Processing Elements (PE)	No. of PE Pairs	Highest Cohesion between PE pairs	Module Cohesion
P-2	initialize	4	6	{all pairs Coincidental}	Coincidental
	schedule_A	1	0	{}	Functional
	figure_tax	1	0	{}	Functional
	compute_tax	1	0	{}	Functional
	valid_data	1	0	{}	Functional
	main	11	55	{all pairs Coincidental}	Coincidental

Table B.3 Processing element information for the Accounting Program (P-3)

Program Code	Function Name	No. of Processing Elements (PE)	No. of PE Pairs	Highest Cohesion between PE pairs	Module Cohesion
P-3	initialize	5	10	{9 pairs Coincidental, 1 pair Sequential}	Sequential
	change_monthly	2	1	{Sequential}	Sequential
	process_transaction	2	1	{Communicational}	Communicational
	process_end_of_month	5	10	{8 pairs Coincidental, 2 pairs Sequential}	Sequential
	process_report	0	0	{}	Undefined
	main	4	6	{3 pairs Coincidental, 3 pairs Sequential}	Sequential

Table B.4 Processing element information for the Bank Promotion Program (P-4)

Program Code	Function Name	No. of Processing Elements (PE)	No. of PE Pairs	Highest Cohesion between PE pairs	Module Cohesion
P-4	assess_cashflow	2	1	{Coincidental}	Coincidental
	assess_account_status	1	0	{}	Functional
	recommended_account	1	0	{}	Functional
	main	0	0	{}	Undefined

B.2 Processing Element Information for Programs in Experiment 3

Tables B.5 through B.7 provide, for various implementations of *lex.scheme*, *calc*, and *kwic* systems used in *Experiment 3*, information about the average number of processing elements when functions displayed the corresponding cohesion level.

Table B.5 Average number of processing elements for interface functions in *lex.scheme*

Function Name	Func	Seq	Comm	Proc	Logi	Coin	Undef
compare_token	1	3				2	
get_char	1	2.14				2	
get_token	1	2.16		2		2	
is_eof_char_stream	1	2					
is_eof_token	1					2	
main	1						0
open_char_stream	1	2					
open_token_stream	1	2				2	
print_token	1	2				2	0
unget_char	1	2				2	0
unget_error	1						0

Table B.6 Average number of processing elements for interface functions in *calc*

Function Name	Func	Seq	Comm	Proc	Logi	Coin	Undef
assign_value	1	2	2				0
create_symtab	1	2					
evaluate_a_line	1	2		2			
get_token	1	2		2		2	
get_token_string	1						
get_token_type	1						
get_value	1	2.4	3				
initialize_calculator	1						
main	1						0
match_token_type	1					2	

Table B.7 Average number of processing elements for interface functions in *kwic*

Function Name	Func	Seq	Comm	Proc	Logi	Coin	Undef
alphabetize	1	2.1				2	
append_to_line	1	2.3	3		5	2	
circular_shift	1	2			3	3	
circulate_and_add_line		2.8		5		3	
dump_line	1	2					0
dump_line_storage	1	2				2	0
empty_line_storage	1					2	
get_line	1	2.2			5		
line_cmp	1	3	2.9		2	2.5	
line_to_string	1	2.3				2	
main	1						0
num_of_lines	1	2				2	
num_of_words	1	2			3	2	
string_to_line	1	2.5				2	