

12

VERIFICATION AND VALIDATION  
OF SAFETY RELATED SOFTWARE

by

Matthew G. Arno

Submitted to the Department of  
Nuclear Engineering in Partial Fulfillment of  
the Requirements for the Degree of

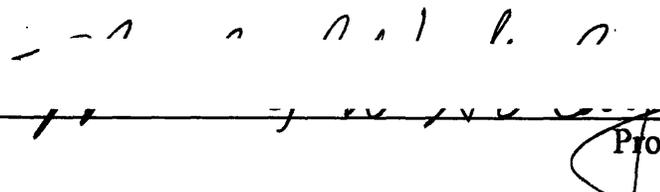
Master of Science in Nuclear Engineering  
and  
Bachelor of Science in Nuclear Engineering

at the  
Massachusetts Institute of Technology  
June 1994

© Matthew G. Arno 1994  
All rights reserved

The author hereby grants MIT permission to reproduce and to distribute publicly  
copies of this thesis document in whole or in part.

Signature of Author \_\_\_\_\_  
Department of Nuclear Engineering  
May 13, 1994

Accepted by  \_\_\_\_\_  
Prof. Michael W. Golay  
Thesis Supervisor

Accepted by \_\_\_\_\_  
Chairman  
Department Committee on Graduate Students

Science

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

JUN 30 1994

LIBRARIES

# VERIFICATION AND VALIDATION OF SAFETY RELATED SOFTWARE

## Abstract

Introducing digital control systems into nuclear power reactor safety systems requires, as a practical matter, that the software code be shown to be free of errors. If this is not the case, the uncertainty associated with the probability of an error manifesting itself prohibits the program's use in a safety system.

Complete testing ensures that a program is free of errors. An approach for this for simple software program is that of McCabe ("Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.", NBS publication 500-99, 1982). This approach analyzes the structure of a program using flowgraphs of the program execution sequence to define independent paths and create complete sets of tests. This method can be streamlined by testing multiple paths in parallel, or at the same time. Doing this requires checking to be sure that the two paths being tested are not logically interconnected. If this can be done on a large scale, larger programs can be tested much quicker than by testing each path sequentially, or separately.

Small, simple programs can be completely tested and large, complex programs cannot be completely tested. This work attempts to answer the question of where the demarcation line between testable and untestable programs is by determining how large of a program can be tested completely. As the program gets larger, so will the amount of testing necessary and reducing potential errors occurring in the testing become important. This work identifies likely locations for errors and shows the tradeoff between error reduction and reduction in number of tests needed for complete testing.

As programs grow in size, the number of independent paths gets larger fairly quickly. The majority of the savings in number of tests needed compared to testing every independent path sequentially is obtained by testing non interrelated sections in parallel and testing interrelated sections in series. Further test savings can be made by carefully coordinating the testing of interdependent section, but this increases the chances for error. The hybrid parallel-sequential testing method offers a compromise between the number of tests needed and ease of setting up the tests. The maximum size of a program that can be completely tested depends upon the number of tests that the tester is willing to conduct and upon how much probability of error or difficulty of setting up the tests that the tester is willing to tolerate.

Thesis Supervisor: Prof. Michael W. Golay, Professor of Nuclear Engineering.

## Table of Contents

|   |    |
|---|----|
| Abstract . . . . .  | 2  |
| Table of Contents . . . . .   | 3  |
| List of Figures and Tables . . . . .                                  | 4  |
| 1. Introduction. . . . .  | 5  |
| 1.1 Motivation for this Work. . . . .                                 | 5  |
| 1.2 Background. . . . .   | 6  |
| 1.3 Scope of this Work. . . . .                                       | 8  |
| 2. Literature Survey. . . . .   | 10 |
| 2.1 Poorman's Results. . . . .  | 10 |
| 2.2 Complexity Measures. . . . .                                      | 12 |
| 2.2.1 Halstead's Metric. . . . .                                      | 13 |
| 2.2.2 McCabe's Metric. . . . .  | 15 |
| 2.2.3 Metric Selection. . . . .                                       | 17 |
| 2.3 Review of Verification and Validation Literature. . . . .         | 18 |
| 3. Method of Analysis. . . . .  | 19 |
| 3.1 Flowgraphs. . . . .   | 19 |
| 3.1.1 Program Independent Flowpaths. . . . .                          | 20 |
| 3.1.2 Finding the Independent Paths. . . . .                          | 23 |
| 3.2 PLC Codes. . . . .  | 25 |
| 3.2.1 Structure of PLC Codes. . . . .                                 | 25 |
| 3.2.2 PLC Flowgraphs. . . . .   | 30 |
| 4. Program Testing . . . . .  | 33 |
| 4.1 Introduction. . . . .   | 33 |
| 4.2 Parallel Testing. . . . .   | 33 |
| 4.3 Calculating the McCabe Value. . . . .                             | 36 |
| 4.4 Minimum Number of Tests Necessary. . . . .                        | 37 |
| 4.4.1 Analysis by Section. . . . .                                    | 38 |
| 4.4.2 Parallel Subsections. . . . .                                   | 41 |
| 4.4.3 Parallel Sections. . . . .                                      | 43 |
| 4.4.4 Coordinating Overlapping Coils. . . . .                         | 43 |
| 4.5 Defining the Test Input Data and Expected Output Results. . . . . | 45 |
| 5. Summary and Conclusions. . . . .                                   | 48 |
| References. . . . .   | 48 |
| Appendix A. Flowgraphs for Program Sections. . . . .                  | 54 |

**List of Figures and Tables  
In Order of Appearance**

| <b>Type &amp; No.</b> | <b>Description</b>                                    | <b>Page</b> |
|-----------------------|---|-------------|
| Figure 2.1            | Day/Temperature program flowgraph. ....               | 17          |
| Figure 3.1            | Function Representations. ....                        | 20          |
| Figure 3.2            | A simple flowgraph illustrating dependent paths. .... | 22          |
| Table 3.1             | Possible flowpaths for Fig. 3.2. ....                 | 22          |
| Figure 3.3            | Graphical example of a PLC SUB operator. ....         | 26          |
| Figure 3.4            | A typical PLC network. ....                           | 27          |
| Figure 3.5            | Execution order for nodes and networks. ....          | 28          |
| Figure 3.6            | Flowgraph for Fig. 3.4. ....                          | 31          |
| Figure 4.1            | Flowgraph of day/temperature program. ....            | 34          |
| Table 4.1             | McCabe values for each section. ....                  | 40          |
| Table 4.2             | Subsections in each section of the program. ....      | 41          |
| Table 4.3             | Minimum number of tests needed for each section. .... | 42          |

## **1. Introduction**

### **1.1 Motivation for this Work**

As nuclear power plants age and new plants are constructed, the control and safety systems for these plants are being updated and modernized. Most industries and applications requiring automated control, i.e. fly-by-wire aircraft and chemical processing plants, have shifted from using analog, hard-wired systems, to digital control for safety systems. Digital technology has been hampered in its acceptance for use in nuclear power safety systems by the problem of verification and validation (V&V). In order for digital systems to be accepted by the nuclear community and the regulators, V&V problems must be overcome.

Methods have been developed to describe and predict the reliability of the analog systems which are currently used in nuclear safety systems. Digital systems must be shown to be at least as reliable as the older analog systems, with no larger a margin of error in the analysis, for digital control systems to be accepted as a replacement.

High reliability digital software was originally developed for the control of large, valuable systems like fly-by-wire aircraft and telecommunications systems. These programs tend to very large and complex with many opportunities for errors. Due to the size and complexity of these programs, it is impossible to test them completely. These programs are tested, both manually and automatically, until the time scale between error detection events (inverse frequency of error) is much longer than the expected lifetime of the program.

While this method of testing may be adequate for other applications, nuclear safety systems are held to a higher standard of reliability, and it is not clear that the prediction of error frequencies derived from this process is good enough to satisfy social and safety regulatory requirements. If a program is simple and short enough, it is possible to completely test it, thereby eliminating the chance that an error exists in the code. This should satisfy the safety requirements of the Nuclear Regulatory Commission. The question remaining is how large of a program can be tested completely.

## **1.2 Background**

Software presents a different situation from hardware when it comes to predicting errors and failures. With a hardware system, when a component fails, it is fixed or replaced, but it may fail again. Until the component fails though, it works well. An expected failure frequency can be determined through testing or estimated from available data. With a computer program, the situation is different. If there is an error in a program, it exists from the moment that the program is first activated. The error simply is not noticed and removed until the particular section of the program's logic that contains the error is used. Once discovered, an error is corrected and cannot reoccur, unlike a hardware failure, where a component can fail multiple times, even when it is replaced. This preexistence and permanent elimination of errors means that software testing cannot be modeled like testing for hardware components.

A hardware component can be tested repeatedly, failing many times, until a failure frequency is determined. With software, the discovery of an error

eliminates that error but there may still be other errors in the code. At no point can testing be stopped simply because a certain number of errors have been found. As more errors are found, any remaining errors, if they exist, become increasingly hard to find. During testing, normal execution conditions that are expected will be thoroughly tested and will work well. The most probable location for a persistent error will be in a situation that would only be encountered rarely, like during an accident sequence or unusual transient, which, perversely enough, is the worst possible time for an error to become revealed. This is because under those conditions the nuclear power plant operators will be busy with the accident or transient already in progress and unlikely to notice and be prepared to handle a mistake made by the program. The moment when the operators most need the program to function correctly also becomes that time at which it is most likely for the program to malfunction.

Associated with digital software systems are several sources of errors and problems, including:

- faulty logic, conceptual errors in the program's concepts, structure and expression
- transcription errors
- human input and interaction errors
- operating system errors
- hardware failures.

These errors fall broadly into three categories: those of human, hardware, and software errors. Human error is one of the largest sources of system failures and consequently must be carefully considered in the design of any system. Hardware errors and failures need to be evaluated, but the methods available for

doing this are well-developed and understood. Software error is the newest of these categories and therefore less has been done to understand and guard against problems arising in this area. The work presented here concentrates on software testing and V&V.

If a computer program is short and simple enough, it can be tested completely, and should meet the requirements for safety-related systems. To do this, it is necessary to determine whether a program can be tested completely. The work of doing this generally requires one to find the limits on how large and complex a program can be and still be completely tested. As a program grows in size, the number of tests necessary in order to completely test it grows. This can become a limit on how large of a program can be tested. Problems encountered in creating a testing program include determining and specifying the test inputs and outputs for each test and arranging the tests so that the whole program, and all possible conditions, have been checked.

### **1.3 Scope of this Work**

The work reported here is concerned with how to test software completely. Human interactions with the program, the hardware the program is running on, and the program compiler effect the reliability of the system and will be briefly discussed and related to software testing. Other factors , such as environmental effects and hardware failures are important, but are not considered here. This work provides the necessary conditions to test a program completely, determine the number of tests necessary for complete testing of a program, and sets up a complete test for an example program.

Given that it is possible to test small, simple programs completely and that large, complex programs cannot be tested completely, there must be some point between the two extremes separating completely testable programs from program which cannot be completely tested. Where this point is located is of interest to software designers and testers. This work investigates the question of where the demarcation line between testable and untestable programs is by examining how large of a program can be tested completely. In addition, as the program gets larger, so will the amount of required testing and the associated number of errors occurring in the testing will become important. This work identifies likely locations for errors and shows the tradeoff between error reduction and reduction in number of tests needed for complete testing.

The literature on the subject of computer software verification and validation is discussed and summarized. This serves to narrow the focus and provides a starting point for this work. The method of analysis being used is described and explained. Next, the programming language in which the example program is written is described. In the next section, the example program is analyzed, with step-by-step explanations, leading to the results and conclusions obtained.

## 2. Literature Survey

Most of the literature on software testing deals with long, complex programs thousands of lines long. These programs control complicated systems with complex hardware-software interactions. Most of the literature agrees that this type of program cannot be completely tested. The literature concerned with the testing and verification and validation of simple safety-related software is much more limited. For simple software, there are methods described that provide a starting point for further work in testing simple software.

### 2.1 Poorman's Results

If a program has been tested completely, this "guarantees the correct operation of the code, given that the computer hardware functions properly." [5, p.104] Poorman determined that it is possible to test a program completely, given certain properties of the program. These properties involve human interactions and the nature of the control algorithm. Human interaction with the program should be held to a minimum in order to avoid problems arising in the man-machine interface. Before a program is written the control algorithm, or logic, of the program must be understood. The logic should also be as straightforward as possible in order to reduce the possibility of making a conceptual error not easily discovered. This applies equally to both digital and analog systems. With safety-related software, the logic is usually simple and straightforward. A test of a computer program consists of specification of a set of input data, and the subsequent comparison of the program's output results to those expected for this set of input data. If a conceptual error is made in writing the program, testing of the program may not reveal this type of error because the

anticipated and produced result are both incorrect. For the purposes of this work, it is assumed that the control algorithm has been developed and is correct.

Software reliability is composed of three sections: human interaction concerns, hardware concerns, and software concerns. The hardware required for digital systems is more complex than that used for analog systems, but the technology required is well-developed and the reliability analysis methods and maintenance and testing procedures used for both are the same. Both types of hardware can be made equally reliable.

With human interactions, as a worst case scenario, the interface between humans and the software can be maintained the same as it was with the preceding analog systems. If the interface is unchanged, the human error frequency, whatever it is, should remain unchanged also. Therefore the human error frequency should be no greater than previously, and with improvement in the human-machine interface, the error frequency from this class of causes should be reduced.

Once it has been ascertained that a code is error-free, by means of complete testing, any future problems encountered with the system will be attributable to either the hardware or human error, and therefore the system reliability can be modeled with methods that are already in place and well-developed. The goal of complete software testing is to eliminate the logic and expression of a program as sources of failure so that the reliability of the entire system depends on other sources of failure, sources which can be modeled.

Poorman used McCabe's [2] cyclomatic complexity theory to evaluate Programmable Logic Circuits (PLC) codes using relay ladder logic. This type of logic was developed in order to mimic the logic used in older analog control systems when expressed by modern digital systems. During an execution of the code, the program will follow a certain "path" through the code determined by the values of the input variable. A flowpath of a program is a graphical representation of all the possible paths that the execution could take when executing the code. By constructing a program state flowpath of the code, which is easily done, given the nature of ladder logic, the number of independent state paths through the code can be determined, which places an upper bound on the number of tests that are necessary to fully test the code. This results in his conclusion that complete testing of small programs is possible and feasible if the number of independent paths is small enough. Poorman presents a methodology for testing programs, but he only demonstrated it using one logical block in a subsection of a code. For larger pieces of code, his procedure is not the most efficient possible. In other words, it is possible to test a computer program completely in fewer tests than would be required if his method were implemented for an entire program. This work will show how to take Poorman's methodology and streamline it to work better with larger programs.

## **2.2 Complexity Measures**

There are various to ways to quantify the nature or complexity of a program. Complexity measure can be divided into two categories, linguistic and cyclomatic. Linguistic measures utilize the properties of the written code, such as number of statements, number of words, and number of input and output data values. Cyclomatic measures attempt to quantify the logical complexity of the

flowgraph of the program, not the text of the program. The two categories of measures each apply better to different types of programs. There are many different measures of each type and some which are hybrids of both types, but these two categories are generally recognized as the standards that other measures are compared to. [7]

### 2.2.1 Halstead's Metric

Halstead's metric [6] is a linguistic metric that was developed by testing many different programs and empirically fitting error data to the results. This metric has been used as the basis for many subsequent linguistic metrics. By using this metric, one will know roughly how many errors to expect to exist in a program once it has been created. This metric counts the number of distinct operators in the code (i.e. if..then, while..do statements), designated as  $n_1$ , and the number of variables and data structures, or operands, used, designated as  $n_2$ . The total number of times that each is used is also counted and given by  $N_1$  and  $N_2$  respectively.

With these values, Halstead defines a program length as being measured by the variable "H" where:

$$H = n_1 \log_2(n_1) + n_2 \log_2(n_2) \quad (2.1)$$

and by the "Halstead length", "N":

$$N = N_1 + N_2. \quad (2.2)$$

Using a combination of these measures, Halstead determined that the total number of errors that can be expected in a program is given by the relationship:

$$\text{Number of Errors} = N \log_2(n_1+n_2)/3000. \quad (2.3)$$

For example, a program with 75 operands which are accessed 1300 times and having 150 distinct operators which are used 1200 times would be expected to have  $(1200+1300)\log_2(75+150) = 6.5$  errors according to this metric.

This metric can be useful for estimating how many errors one would typically expect to find in a code before any programming actually starts with a general idea of what the code has to do and what methods will be employed in the program. The disadvantage to this metric is that programmers can learn to anticipate and correct for the sort of errors that this metric anticipates, and consequently requiring a new empirical relationship. In addition, the number of errors anticipated is not the number of errors that there are in the code. There may be more or fewer. A program of testing cannot be stopped when the expected number of errors have been found, because there may be more errors, nor can it be continued until the same number has been found, because all of the errors may have been found. This metric is most useful in comparing codes to each other. While it gives a tester an idea of a rough number of errors that may be found, it can not be used to determine how to test the code or when to stop.

### 2.2.2 McCabe's Metric

McCabe [2] developed a "complexity metric" which evaluates the logical structure of a program and returns a value that indicates the "complexity" of the code. His metric determines the number of different flowpaths which exist for a program, in other words, the number of different ways that the code can be executed. Actually, the McCabe metric does not evaluate the actual complexity of a program, according to the usual meaning of the term complex. Usually "complex" is used to refer to the logical difficulty or intricacy of the program. A simple program with easily identifiable logic can have a high McCabe metric value. An example is a *case* statement. If a program displays different information depending on the day of the week, there would be seven different possible "cases." The McCabe metric for this section of the code would be equal to seven although a case statement is a simple logical statement, easily checked for correctness, since only one of the paths can be followed during a given iteration of the program.

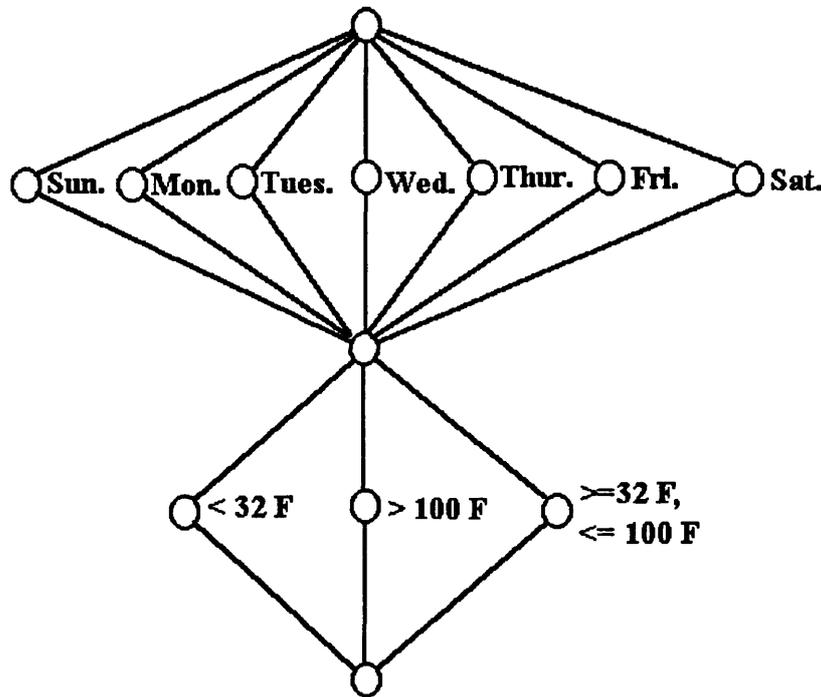
Based on his experience with various programs, McCabe picked a metric value of ten as the demarcation line between what he considered a "simple" program and a "complex" program. This number is not an absolute demarcation because whether a program is actually complex or not will depend as much on the logic employed as on the number of branching points there are in the logic flowgraph. Ten makes a useful rule of thumb though.

The McCabe measure creates of flowgraph of the program, thus identifying the number of locations where the code "branches" and counting all of the branches. For example, a mathematical calculation within a program

would not add anything to the number of paths through the program. In such a case, the program comes to the calculation, performs it, and proceeds to the next command. In that sequence, there is no other option other than performing the calculation that could be exercised. An if-then statement adds one to the number of paths because the program enters the statement at one point and can go one of two different ways upon leaving it (if.., or then..), adding one to the number of executable flowpaths. The result of the tabulation of executable branches thereby is the number of independent executable paths that exist in the code. The benefit of using this metric is that no matter what the logic of the code is, the program can be fully tested in the resulting number of tests. If every independent path is tested, the code will have been fully tested, since any other path through the code will be a linear combination of independent paths. It may be possible to test the code completely in fewer tests, but it will not require more. There are ways to test multiple independent paths at once. In order to test the program it is necessary to find the independent paths and determine the input data and output results corresponding to each independent path. This topic is treated in the next section. Essentially, the McCabe metric sets an upper bound upon the number of tests that will be necessary for complete program testing.

For example, consider a program that displays different information depending on the day of the week and the temperature(below freezing, between freezing and 100°F, and above 100°F), but where the two classes of input data are not interconnected. The flowgraph for this program is shown below in Fig. 2.1. According to McCabe's metric this program would have a metric value of nine, consisting of seven paths for the day of the week and three paths for the temperature ranges, and with one common path between the two sets of input data values. Since the data being displayed which depends upon the day of the

week does not depend on the temperature and vice versa, the two can be tested at the same time with only seven tests. This reduces the number of tests that have to be done by two. As a program gets larger, the number of incidences like this tends to increase, causing the McCabe value to increase much faster than the minimum number of tests necessary.



**Fig. 2.1.**  
Day/Temperature program flowgraph.

### 2.2.3 Metric Selection

If one wishes to test a program completely, the McCabe metric is much more useful because the values and information associated with it are concrete properties of the program. The Halstead metric is an empirical relation that makes a prediction of the number of errors, but it does not help in actually finding the errors. McCabe's metric provides a basis for systematically analyzing

and testing the whole code. For this reason, the McCabe metric is used, as it was in the work of Poorman. The goal of this effort is to discover and set up the necessary tests to test completely a piece of safety software.

### **2.3 Review of Verification and Validation Literature**

Most of the literature concerned with verification and validation of software deals with large programs and human and hardware error issues that are beyond the scope of this work. The literature of these topics is not applicable to the complete testing of simple, high-reliability software codes, and is not discussed here further.

### 3. Method of Analysis

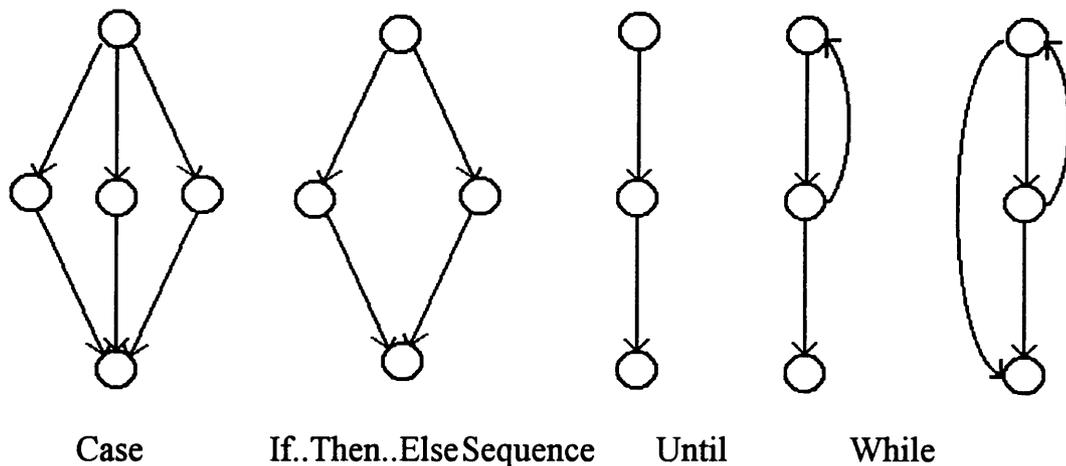
The example program is a Programmable Logic Circuit (PLC) code. In order to analyze it, the program was divided into sections. These sections were then flowgraphed. A flowgraph is a visual representation of all the possible logical paths that the program could follow during an execution. In order to analyze the program, the nature of flowgraphs and of the programming language need to be understood.

#### 3.1 Flowgraphs

A program can be represented visually by a flowgraph. Every node in the flowgraph represents an action performed by the program. Each node has one or more input paths and one or more output paths. If a node has one output path, then the node performs some sort of operation, like a mathematical calculation. A node with multiple outputs performs some sort of comparison, either mathematical, e.g. is A greater than B, or logical, "Is condition C TRUE?" The outputs from the node must represent all of the possible outcomes of whatever evaluation is performed in that node. A mathematical comparison has three possible outputs, a branch each for  $A < B$ ,  $A = B$ , and  $A > B$ . A logical evaluation has two outputs, C "TRUE", and C "FALSE". By creating one of these nodes for each statement in a program and connecting them together properly, a "flowgraph" of the whole program can be created.

McCabe recommends that the population of functions used to create a program be limited when using this approach. He recommends the use of : *sequence, if..then, while..do, until..do, and case.* The flowgraph representations

of these functions are shown in Fig. 3.1. These functions lend themselves to easy representation within flowgraphs. There is one function that should be particularly avoided. This is the *conditional goto* statement. Since this statement allows the path through the program to jump around, it becomes difficult to segment the code into modules which can be analyzed reliably because of the flow discontinuities created by use of this statement.



**Fig. 3.1.**  
Function Representations

### 3.1.1 Program Independent Flowpaths

By visual inspection of the flowgraph of a program, the number of independent paths through the program can be determined. In order to determine the value of the McCabe metric, F (the number of flowpaths), from visual inspection of the flowgraph, it is necessary to count the number of nodes (N) and the number of links between nodes (L), then Equation 3.1 applies, providing the relationship:

$$F = L - N + 1. \tag{3.1}$$

This result applies to strongly connected flowgraphs. "Strongly connected" means that a closed loop exists and every node has at least one input and one output. A closed loop exists when the program does not end execution. When it is finished with one execution, it returns to the top of the program and executes again. For a program that executes once and then is finished, it can be made strongly connected by linking the last node in the program back to the first node. If the first and last nodes of the program are not connected, but could be, then Equation 3.2 is used:

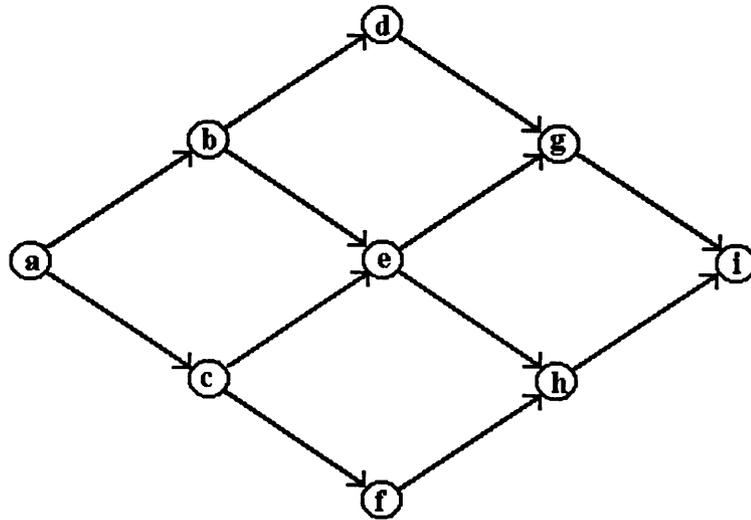
$$F = L - N + 2. \quad (3.2)$$

After creating a flowgraph of a program, the number of independent paths can be determined. After the number of independent paths has been determined, the independent paths need to be identified because there are more ways to traverse the code, or dependent paths, than there are independent paths. Consider the example flowgraph below in Fig. 3.2.

In this example, using the McCabe metric, the number of independent paths through this flowgraph is  $L - N + 2 = 12 - 9 + 2 = 5$ . There are five independent paths in this flowgraph, but a quick visual examination can determine that there are six total possible flowpaths through the segment of code. Table 3.1 gives the possible flowpaths.

Since there are six possible paths and only five independent paths, one of the paths must be a linear combination of the other five paths. The selection of which five paths are to be defined as the independent paths in this case is not

unique, Any flowgraph beyond the simplest will have multiple sets of paths each of which can be considered to constitute a complete set of independent paths.



**Fig. 3.2.**  
A simple flowgraph illustrating dependent paths  
Reproduced from [5, p. 46]

**Table 3.1**  
Possible flowpaths for Fig. 3.2.

| Path | Flowpath  |
|------|-----------|
| 1    | a+b+d+g+i |
| 2    | a+b+e+g+i |
| 3    | a+b+e+h+i |
| 4    | a+c+e+g+i |
| 5    | a+c+e+h+i |
| 6    | a+c+f+h+i |

Visual inspection of Fig. 3.2 indicates that paths 1 and 6 are independent. This leaves four paths, three of which are independent and the fourth is a linear combination of the other three. For this example any of the three are

independent, with the fourth being a combination of the other three. If paths 2, 3, & 4 are used as the independent paths, then path 5 can be expressed as path 5 = path 4 + path 3 - path 2.

$$\text{path 5} = (a+c+e+g+i) + (a+b+e+h+i) - (a+b+e+g+i) \quad (3.3)$$

The same procedure can be used with paths 2, 3, or 4 as the dependent path. This example has four sets of independent paths. It does not matter which set is defined as the independent set of paths. One set is as good as another for this purpose since they all can be used in linear combination to construct a given path through the code. This is true not only for this example, but in general.

In this example, all the paths could be quickly determined by inspection. A set of independent paths was constructed easily with no difficulty. Larger structures will not yield to simple visual inspection to determine a set of independent paths. A methodology is necessary to construct a set of independent paths for larger flowgraphs.

### **3.1.2 Finding the Independent Paths**

Since any set of independent paths will be adequate for defining the structure of a program, it is only necessary to find one such set. McCabe developed a methodical procedure for finding a set of independent paths. The first step is to choose a "basepath" through the flowgraph. McCabe suggests picking the path passing through the largest number of nodes in order to facilitate the next step in the process. The choice of basepath determines the nature of the

remainder of the independent paths, eliminating most of the possible alternative path sets from selection.

The next step consists of finding the remaining independent paths. To find the next independent path, find the first node on the basepath which has a branch leading away from the basepath and follow it, returning to the basepath as quickly as possible. For the next independent path, follow the first alternate path until its first branching and then rejoin an already established path as soon as possible. Do this until all the branchings in the alternate path have been followed or the flowpath rejoins the basepath. To continue, follow the basepath to its second branching and follow the same procedure. By using this recursive procedure until all branchings off the basepath have been fully explored, a complete set of independent paths will be created.

Using the flowgraph in Fig. 3.1 as an example, path  $a+b+e+g+i$  is chosen as the basepath since it passes through the largest number of nodes with branches. The next path is found by branching away at the first node, toward node "c", and then rejoining the basepath, creating path  $a+c+e+g+i$ . Branching at the first opportunity from the first alternate path and then rejoining it yields path  $a+c+f+h+i$ . This fully develops the first alternate path, so the basepath is followed to the next branching at node "b", yielding path  $a+b+d+g+i$ . The next path determined is  $a+b+e+h+i$ , finishing off the branches off the basepath. This procedure yields five paths, paths 2, 4, 6, 1, and 3 from Table 3.1, the number expected from the McCabe metric.

This procedure provides a means of systematically finding the independent paths for larger programs, even if the nesting of paths becomes

complex and confusing. Once the independent flowpaths are identified, each path must be tested to verify that it performs correctly. The necessary inputs and the expected outputs can be determined from examination of the flowgraph of the program.

## **3.2 PLC Codes**

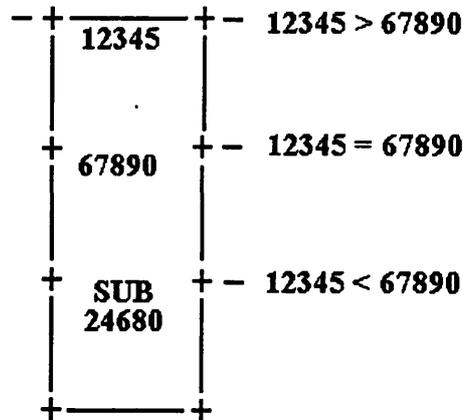
Programmable Logic Circuits (PLC's) were developed for use in high-reliability software systems. PLC's simulate relay-ladder logic. The purpose of this was to gain many of the benefits of pure relay-ladder logic. This and other properties of PLC logic make it attractive for the purpose of complete testing of software.

Relay ladder logic executes every program command during execution, making it difficult to nest a command such that it will not be executed properly. In addition, this property eliminates execution loops and therefore the possibility of becoming stuck in an infinite loop. Also, PLC codes in use by the nuclear industry are limited in how often the code may access input values (once per cycle) in order to prevent the values from changing while the program is executing. This makes the output of the code at the end of a cycle deterministic as soon as the cycle is started. This feature allows a given input data set to be associated with a corresponding output data set.

### **3.2.1 Structure of PLC Codes**

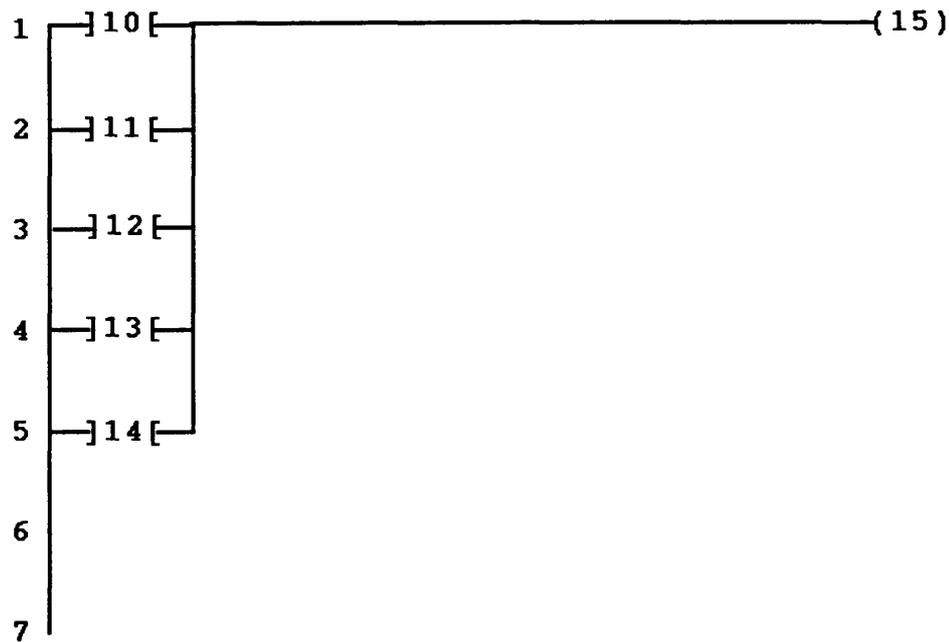
PLC codes are organized into "nodes" making up a "network", with a set of networks making up a "segment." Each node corresponds to a memory

location. Some simple commands require only one node, while more complex commands require several nodes. Each node is placed on a network which consists of an array of 7 rows and 11 columns. A node can be placed at the intersection of any row and column. A typical node is shown in Fig. 3.3. and a typical network in Fig. 3.4.



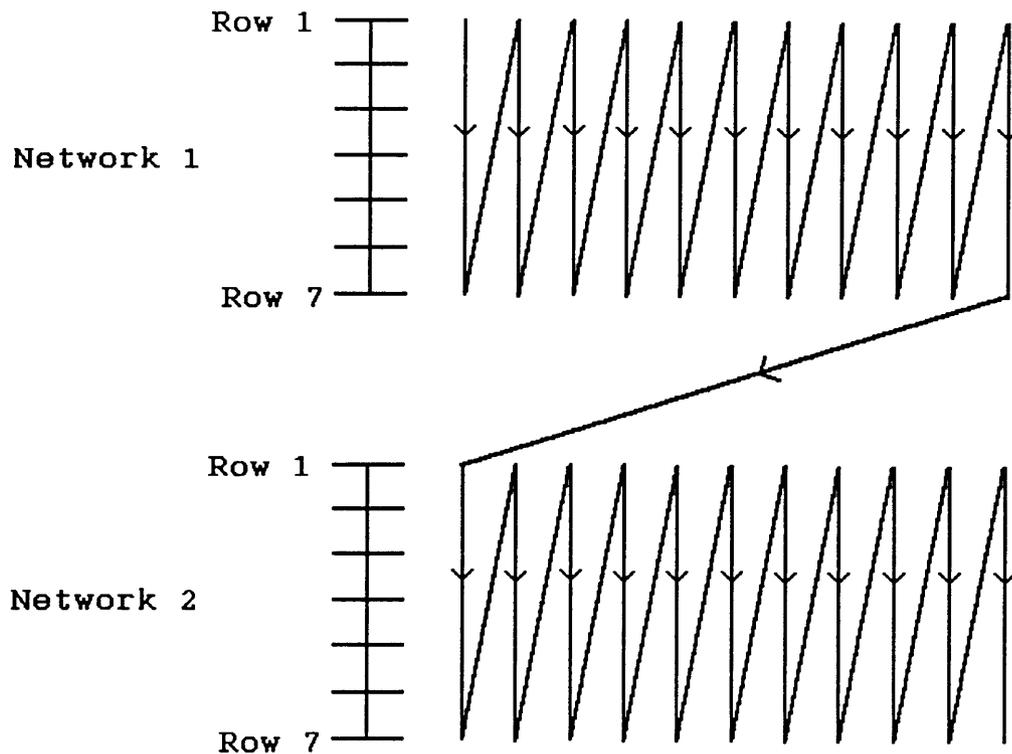
**Fig. 3.3**  
Graphical example of a PLC SUB operator.

The subtraction node in Fig. 3.3 receives power input at the upper left-hand corner. If the lead is powered the node subtracts register 12345 from register 67890 and stores the results in register 24680. In addition, one of three output leads along the right side of the node is powered, depending on the result of the calculation as shown.



**Fig. 3.4.**  
A typical PLC network. [1]

Each network is executed node-by-node top-to-bottom, left-to-right until every node has been executed in the network. After the bottom right node has been executed, execution moves to the top left node in the next network in the code, as shown in Fig. 3.5. Execution proceeds through the code executing every node and every network every time through the code. The exception to this is the *skip* command. This command is analogous to a goto statement in the Basic language. The use of this command is not recommended. It should only be used carefully.



**Fig. 3.5.**  
Execution order for nodes and networks  
Reproduced from [5, p. 58]

The PLC language has a limited set of operators. There are several types of sequential operators that manipulate data values. Of non-sequential operators, those which cause a branching in the flowgraph of the code, there are four types: counters, timers, comparitors, and Boolean operators. Counters count up to or down from a specified variable value, returning a different result depending upon whether or not the counter has reached zero for downcounters and the variable value for upcounters. Timers are similar, measuring elapsed time rather than number of counts. The comparitor is a subtraction node that is used to compare two variable values rather than as a mathematical operation. It has three possible results:  $A < B$ ,  $A = B$ , and  $A > B$ . The Boolean operator checks if a value is TRUE or not and adjusts its output

accordingly. The Boolean operator may return a TRUE value if the variable value checked is alternatively TRUE or FALSE, depending on the operator.

Data are manipulated and moved from network to network by means of coils and registers. Coils are designated by a three digit number and are either in an "on" state (TRUE) or an "off" state (FALSE). The state of a given coil may only be modified at one point in the code, but it may be evaluated in a Boolean operation any number of times both before and after its state is modified. The 11th column of each network is reserved for modifying coil states. A coil in the 11th column is turned on if the lead coming to it is powered, and off if the lead is not powered. Registers are designated by five digit numbers and may store either numerical values or other information bit-by-bit. The size of a register depends upon the processor used to execute the PLC code, but is either 8-bit or 16-bit. The values contained in a register may be modified and read as many times as desired during the execution of a code. Registers storing numerical values are modified using BLKM, ADD, SUB, MUL, and DIV nodes. An individual bit can be changed in a register by using an MBIT node. A BLKM node copies one register to another register. ADD adds two numerical register values together and stores the result in a third. SUB, MUL, and DIV are similar nodes for their respective mathematical calculations.

For the purposes of determining the independent paths through a code and complete testing, the operators that cause the code to branch are the interesting cases. In the example program used in this work, a branch may occur at a SUB, SENS, UCTR, DCTR, or TIMR module or at a coil evaluation. A SENS node checks the value of one bit of a register. UCTR, and DCTR are up and down counters respectively. UCTR counts up from zero to a specified value. DCTR

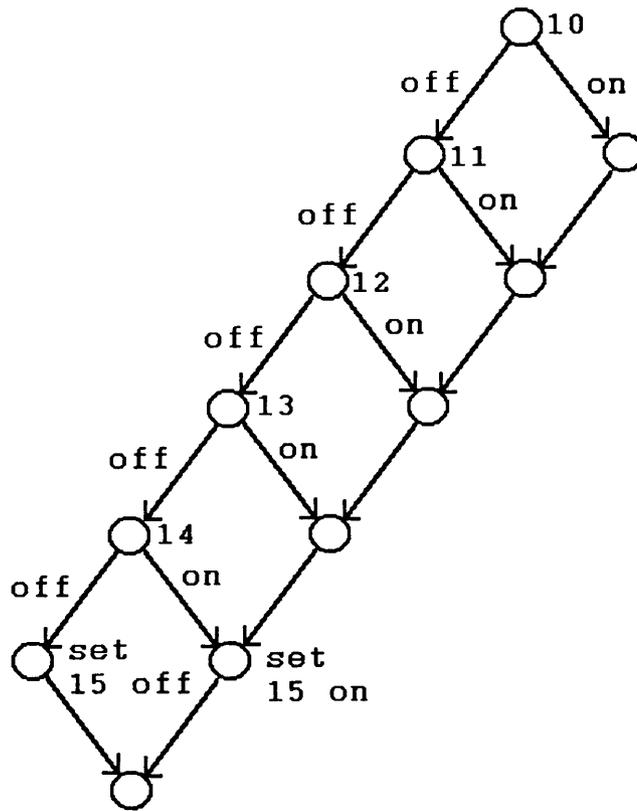
counts down from the specified value to zero. A TIMR node measures elapsed time until a specified amount of time has passed. The actions performed by the other operators still need to be known in order to connect the flowgraph together properly and to create an input data set needed to test a path, but they do not add to the number of paths.

A complete description of PLC programming can be found in the Modicon Programming Manuals. [3,4]

### **3.2.2 PLC Flowgraphs**

Due to the graphical nature of PLC codes, they are easier to flowgraph than programs written in other languages, such as FORTRAN or C. The network shown in Fig. 3.4. shows the logical "or" operation involving use of five coils to set the state of another coil, creating the flowgraph shown in Fig. 3.6. If any of these five coils are in the "on" state, then the coil being modified will be turned "on."

Every network in a program can be graphed this way, of course most will be more complicated than this example. Once each network has been flowgraphed, the flowgraphs can be used as a basis for constructing a flowgraph for a given subsection of the code. All the subsections of the program can then be combined into a single flowgraph if the program is not too large or complex.



**Fig. 3.6**  
Flowgraph for Fig. 3.4.

The limiting factor in how large a program can be flowgraphed is the ability to construct a flowgraph where the state of every given coil is only evaluated once. Since a coil remains set in the same position for a given iteration of the program, it should only be evaluated once in the flowgraph no matter how many times it is evaluated in the program. The first time that the coil is evaluated determines the flow direction in which the logic proceeds and that flow direction cannot be changed subsequently. Thus, at subsequent coil state evaluations, only one flow path outcome is allowed, and no possibility for branching actually exists. Constructing a flowgraph such that each coil state is only evaluated once is called reducing the flowgraph. For proper analysis of the program, only reduced flowgraphs should be used in the analysis.

If the program is too large to be combined into one reduced flowgraph, there are ways to analyze the code section by section, as is discussed in the next chapter.

## 4. Program Testing

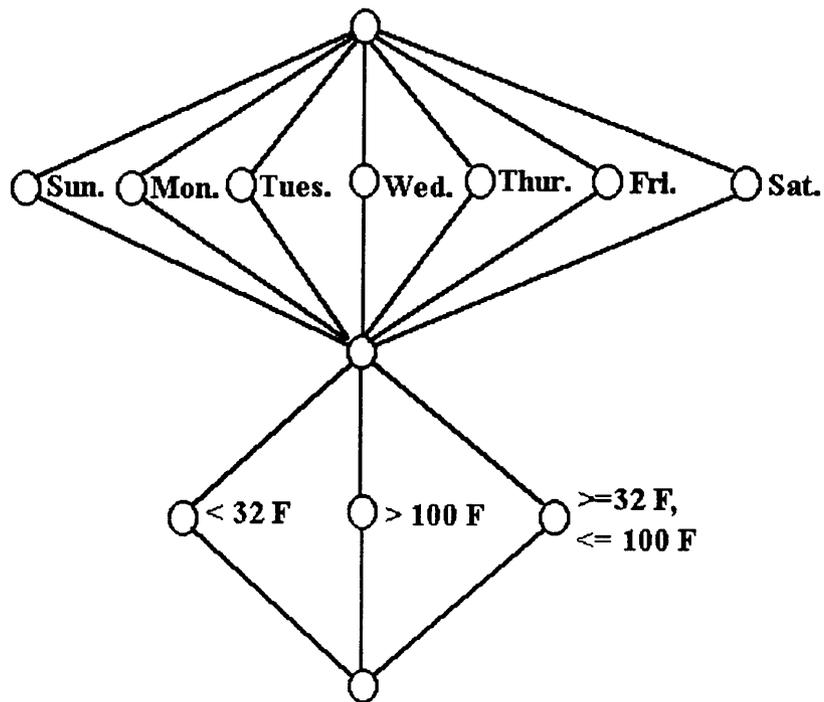
### 4.1 Introduction

The investigation described in this work was performed using an example PLC ladder logic program supplied by the ABB-Combustion Engineering Co. The program is used in a Diverse Nuclear Power Reactor Protection System. It receives input signals from the pressurizer, steam generators, motor generators, and the control room and it produces plant component trip signals and relays signals which are sent back to the control room.

### 4.2 Parallel Testing

Consider the program example from Chapter 2 involving information displayed depending on the day of the week and the temperature. Represented in a flowgraph, such a program is displayed as in Fig. 4.1.

As can be seen in Fig. 4.1, this program branches out in a *case* statement depending upon the day of the week, it displays information, and then all the branches come back together at one node. The program then branches out in another *case* statement depending upon the temperature. All these branches then come back together at the bottom of the flowgraph. The program would then either stop execution or return to the beginning.



**Fig. 4.1**  
Flowgraph of day/temperature program.

If each independent path is tested separately, testing this program would require nine tests. The basic premise of testing only independent paths relies on the fact that it is not necessary to test every single combination of paths that could be taken. It is sufficient to make sure that having reached a given node, it executes properly and connects to the right succeeding node. If this is all that is necessary, then it should be possible to test two independent paths at the same time as long as there is no interrelation between the two path segments being tested.

Testing each independent path separately in the above program would first require the establishment and testing of a basepath. The path through the Sunday node and the below-freezing node will work. Next, the Monday through

Saturday paths will be tested, each time also passing through the below-freezing node. After the Saturday path has been tested, the next test would be the combination of the Sunday path and the above-100°F path. Lastly, the Sunday path and the freezing-to-100°F path combination would be tested.

An alternate method of testing is to test a new path in the upper section and a new path in the lower section at the same time. First, test the same basepath, then test the Monday/above-100°F path, then the Tuesday/freezing-to-100°F path. Next, finish testing the Wednesday to Saturday paths in the same way as above. This method of testing needs only seven tests to test the same program, netting a savings of two tests. The premise in this method of testing is the same as in testing only the independent paths. It does not matter how the program arrives at a given node as long as it executes that node's operations correctly. This premise has not been violated by testing more than one independent path at the same time.

Testing every possible combination in this program requires 21 tests. Testing the independent paths separately requires 9. Testing some of them simultaneously lowers the number of tests needed to 7. This simultaneous testing will be referred to as parallel testing. After a flowgraph of a program has been constructed, sections that can be tested in parallel will become apparent. Each time all of the branches in the program's reduced flowgraph come together into one node before proceeding to the next portion of the code indicates a location where the lower portion of the program does not depend on the upper portion. This method only works where the complete reduced flowgraph of the entire program can be constructed.

Safety software programs tend to do many unconnected operations with multiple inputs and outputs. For instance, the program being analyzed in this work has a section involving the pressurizer and other sections dealing with the steam generators. The sections do not depend on each other. In other words, the calculations and logic being executed in the steam generator sections do not depend on the logic and operations in the pressurizer section. This feature of the program means that it is likely that whole sections of the program can be tested in parallel, yielding a large reduction in the number of necessary tests.

Other types of programs are not as amenable to parallel testing. Safety software monitors a system and generates trip signals, alarms, and activates components that deal only with that system as needed. A given program might control several systems, as this one controls the pressurizer, the steam generators, and the motor generator. Programs that control only one system or have complex control logic will benefit less from parallel testing.

### **4.3 Calculating the Value of the McCabe Complexity Metric.**

The first step in completely testing the program is to calculate the program's McCabe complexity metric value. The nature of PLC logic, especially the graphical representation used in a listing of the code, makes this a simple procedure. Every mathematical compare (SUB) operation increases the McCabe metric value by two. The SENS, UCTR, DCTR, and TIMR operations each increase it by one.

A coil state evaluation is a special case. If a coil state is evaluated before it is changed in a given iteration of the program, the McCabe metric value

increases by one. In addition, each coil only contributes to the McCabe metric value once. Multiple evaluations of the state of the same coil still only increase the metric value once. If a coil state is evaluated after it has been set earlier in the program, the evaluation does not increase the value of the metric at all because the flow direction which the program logic can take at that point has already been determined earlier at the point the coil state was set in the program. Therefore the code can not branch differently at the later evaluation.

In order to determine the McCabe metric value, one must take the number of additional tests that will be required from each operation that had branches and add one to the value in order to account for the base path.

#### **4.4 Minimum Number of Necessary Tests**

In order to investigate the difficulty of completely testing a complex program an industrial example program was examined. The program being analyzed has a McCabe metric value of 171. This value does not represent the complexity of the program accurately. As is noted earlier, the McCabe metric tends to increase in value much quicker than the actual complexity of the code. One would like to be able to test a program without running 171 tests. If a smaller number of tests can be used without sacrificing completeness, this would be desirable, especially if the difference in number of tests is large.

##### **4.4.1 Analysis By Section**

With a code as large as this example, it is necessary to break the program into smaller pieces for analysis. Once these smaller pieces have been analyzed,

they can then be recombined to form the analysis for the whole program. This necessity of breaking the program into pieces vindicates McCabe's selection of a metric value separating simple from complex programs. Breaking the program into pieces reduces the McCabe metric value for the amount of code that is being analyzed at once. The code is eventually broken up into pieces that are the size of McCabe's cutoff value between complex and simple programs.

By inspection, the program is easily divided into sections, sometimes known as modules, that each have individual functions. These functional boundaries also mark logical boundaries where the logic of the preceding set of networks has no effect upon the logic of the succeeding set of networks. The program structure is outlined in Table 4.1. The purpose of the first section of the program is to load constants and to initialize the program (Init.). The next section checks the trip setpoint settings (Trip check). After that there is one section each for the pressurizer (PZR), motor generator (MG), steam generator 1 (SG1), and steam generator 2 (SG2). Next is a section that generates the trip signals (Trip Gen.). The last two sections are used for creating output signals (Output 1, Output 2). If all nine of these sections were totally unrelated to each other, it would be possible to test the whole program in however many tests it takes to test the largest section.

The nine sections are not totally unrelated, but the amount of interrelationship is small. By analysis of the program, it is possible to determine how and where the sections interrelate. Most of the interrelation comes from the output data sections using results obtained in the previous sections. The system contains one coil that is used in every section. This is the coil that puts the program into an automatic test mode. This autotest coil is controlled by the

operators and is not altered by the program. This coil is an exception to the rule that each section is an independent module in the code. It is used in many networks spread throughout the program. It is dealt with here in a different manner in order to allow the rest of the program to be analyzed in the normal manner.

The program analysis was conducted by dividing the program into sections that were small enough to be analyzed reliably. In order to create the reduced flowgraph of each section, the analysis was started at the smallest unit, the network, and then increased in size. First, the logic of each network in the program was flowgraphed. This flowgraphing was then used as the basis for forming the reduced flowgraph for a section. If a section is too large to create a reduced flowgraph, it should be subdivided again. This did not pose a problem with the code analyzed, but there is a possibility that such a problem might occur. Although the source code for the example program is proprietary, the flowgraphs created during the analysis are listed in Appendix A.

This type of analysis depends on having a reduced flowgraph of the code. Without the reduced flowgraph, determining the nature of the logic of the program becomes much more difficult. The section forms the basic element for analysis of the program because this is the largest amount of code that can be conveniently formed into a reduced flowgraph.

Once each section has been identified and the reduced flowgraphs for each section have been constructed, the McCabe metric value for each section is determined. Table 4.1 gives the results of this analysis.

**Table 4.1**  
McCabe metric values for each section of the example program.

| Program Section | McCabe metric value |
|-----------------|---------------------|
| Initialization  | 42+1                |
| Trip Check      | 30+1                |
| Pressurizer     | 20+1                |
| M.G.            | 11+1                |
| S.G. 1          | 23+1                |
| S.G. 2          | 19+1                |
| Trip Gen.       | 11+1                |
| Output 1        | 22+1                |
| Output 2        | 53+1                |
| <b>Total</b>    | <b>231+1</b>        |
| Entire Program  | 170+1               |

The number of tests necessary in each program section is given as a number plus one in order to represent the basepath as one of the paths. This path would only be tested once, so when the total value for the metric is evaluated, it is only counted once. The total shown in Table 4.1 indicates how many test would be necessary in order to test the program if each program section were tested sequentially. As can be seen, testing each section sequentially increases the number of tests needed. A significant benefit comes from testing sections simultaneously, or in parallel.

#### 4.4.2 Parallel Subsections

Within a section, there may be subsections that can be tested in parallel. When the reduced flowgraph for such a section is created, the flowgraph will have locations where all the branches come back together into one node. The logic below this node will not depend on the logic above it. Either subsection could be executed initially with no change in the program's logic or output. Each

section in the example program possesses several such locations. Each section is divided into subsections that are separate in the same manner that the whole program is divided into sections. Table 4.2 shows how many separate subsections appear in each section.

**Table 4.2**  
Subsections in each section of the example program.

| Program Section | number of parallel subsections |
|-----------------|--------------------------------|
| Init.           | 13                             |
| Trip Check      | 7                              |
| PZR             | 4                              |
| MG              | 2                              |
| SG1             | 4                              |
| SG2             | 4                              |
| Trip Gen.       | 6                              |
| Output 1        | 4                              |
| Output 2        | 25                             |

The limiting factor in determining the minimum number of tests needed in order to completely test one section of the program is the number of tests needed to test the largest, or most complex, subsection. If each subsection is truly separate, the smaller subsections can be tested at the same time in fewer tests. Table 4.3 lists the number of tests necessary to test each section, based on the largest subsection in each section.

The number of tests needed for a subsection is essentially equal to the McCabe metric value for the subsection. McCabe chose 10 as the value distinguishing simple from complex programs. [2] The analysis of the example program reduces the size of each component module that is being tested to having a maximum metric value of 11. This was not done intentionally, but the

result shows a justification for McCabe's choice of 10 as the transitional metric value.

**Table 4.3**  
Minimum number of tests needed for each program section.

| Section    | # of tests |
|------------|------------|
| Init.      | 7+1        |
| Trip Check | 3+1        |
| PZR        | 4+1        |
| MG         | 5+1        |
| SG1        | 4+1        |
| SG2        | 4+1        |
| Trip Gen.  | 3+1        |
| Output 1   | 7+1        |
| Output 2   | 10+1       |
| Total      | 47+1       |

Now, if each section is tested sequentially, 48 tests will be needed in order to test the entire program. This is a marked improvement from the 171 tests estimate to be needed if every independent path in the program were tested one at a time. Even with this marked decrease in the number of tests needed, the required number can be reduced even further. By testing several sections at a time just as several subsections are tested at one time, the number of tests needed can be reduced even more.

#### 4.4.3 Parallel Sections.

The next step in reducing the number of tests is to test sections simultaneously that are not interconnected. Trip checks, PZR, SG1, SG2, and Output 1 are totally unrelated, so they can be tested in parallel. The MG section

depends upon the PZR section and the Trip Gen. section depends upon the SG sections. Thus, these two sections can be tested in parallel after the previous four sections have been tested. The program can now be tested in four stages. The first stage tests Init.; the second, Trip checks, PZR, SG1, SG2, and Output1; the third, MG and Trip Gen.; the fourth, Output2. Each stage takes as many tests as the longest section contained in it. The program can now be tested in 30 tests.

Up to this point, the only coil that has required special attention is the autotest coil. In order to reduce the number of tests further, it is necessary to pay careful attention to the coils and memory registers that are used in different sections of the code in order to coordinate how each section expects the coil to be set for a given test.

#### **4.4.4 Coordinating Overlapping Coils**

Reducing the number of tests requires that coils evaluated or set in a particular state in one section of the program are in the proper state for a test in another section of the code. The PZR section sets coils that are evaluated in the MG section. It is necessary to make sure that the PZR section sets the coils the right way in order to conduct the tests needed for the PZR section. This task is made easier by the fact that most of the sections require fewer tests to be completely tested than the number of tests needed for the largest section. The largest section, Output 2, also evaluates the greatest number of coils that were set or evaluated earlier in the program also.

In order to coordinate overlapping coils, listings are made of the coils and memory registers used by each section. The sections are then compared in order

to see which coils are used in multiple sections. When the input and output sets are created for each test, the status of the overlapping coils must be carefully checked. The arrangement of the order of the tests for a given section will need to be changed to coordinate coil positions between all the sections.

The program has 50 coils that are used in multiple sections of the code. This represents under half of the total number of coils used. Only two memory registers are used in multiple locations. Usually, coil states are set based on operations involving the memory registers, such as a mathematical compare or SENS operations. The coil is then evaluated in a different location in the program rather than accessing the memory register again.

Coordination of knowledge of each coil's status was relatively easy for most of the coils. Only about five of the coils provided trouble in checking and ensuring that they were set correctly for each section's testing. These five coils were harder to coordinate because of the logic which controlled their state and also the nature of the logic in which they were being evaluated later in the program. Most of the overlapping coils were set with simple logic and then evaluated later in a simple logical expression. At the later evaluation, many of the coil were used to generate signals to be sent to the control room to indicate that status of the coil's state, which is a simple binary logical statement. When all of the coils had been checked and verified that they were coordinated correctly, 17 tests were needed in order to test the entire program and still have the overlapping coils set correctly. This number is contrasted to the minimum of 11 tests that are necessary in order to test the largest subsection of the program. Only 6 additional tests are needed to coordinate the state of the overlapping coils between all the sections of the program.

#### **4.5 Defining the Test Input Data and Expected Output Results**

The input data and output results sets for each test are determined by examination of the reduced flowgraphs for each subsection. Due to the small size of each subsection, it is easy to reliably determine the relationship between input and output. Once a complete set of independent paths has been created, the input necessary for each path to be followed can be readily determined by inspection of the flowgraph. In the same manner, the output of a given path can be determined. It does not matter in what order the paths are tested as long as they are all tested.

Referring to the day/temperature program example again, the first test case input would be the day variable equals Sunday and the Temperature variable is less than 32°F. The test results would be to check and see that the program displayed the information that it is supposed to for Sundays with the temperature less than 32°F. The next test case is day equals Monday and temperature greater than 100°F with the test results commensurate with results desired for the given input.

When program sections that overlap with each other are tested in parallel, care must be taken in the treatment of the coils which are shared in common. By rearranging the order of the tests in each section, most conflicts can be avoided. For example, take the autotest coil in the example program. If the second test for the Motor Generator section needs the autotest coil to be on, but the second test for the Pressurizer section needs it to be off, the second test for the Motor Generator could be moved to a different position in the sequence of tests where

the Pressurizer also needs the autotest coil to be on. If the conflict cannot be avoided within the confines of however many tests are already needed, another test must be added to give room for the conflict to be resolved. With the example program, the necessary setting for the autotest coil in each test caused several tests to need to be added to the total number needed so that it could be set correctly for each test and still coordinate all the other coils, some of which depended on the autotest coil being set a certain way before they could be set a certain way. It is ironic that the portion of the program added for the purpose of automatic testing significantly increased the complexity of the program, making testing harder, and more error prone.

In order to coordinate between sections with overlapping coils, it is necessary to keep track of all coils that overlap and how those coils depend on the setting of other coils and values contained in registers. Rearranging the order of the tests requires knowing the input data set needed for a test. If the test is self-consistent and all the sections in the test can be tested at once, it is a valid test. The challenge is in coming up with a minimum number of tests that are all self-consistent and still make up a complete testing regimen. In the example program, the autotest coil and another coil, the coil which blocks the trip signals (out-stop), were the hardest to coordinate between the sections because the out-stop coil depended on the setting of the autotest coil. The coordination of these two coils between all the sections that they showed up in accounted for most of the increase in the number of tests necessary above the minimum possible.

The need to coordinate the overlapping coils greatly increases the potential for making errors in the creation of the necessary input data sets and the

expected output results sets for each test. If the sections having overlapping coils are tested sequentially, this source of error can be avoided.

## 5. Summary and Conclusions

Once a program has been completely tested, it is thus shown to be free from defects. Complete testing "guarantees the correct operation of the code, given that the computer hardware functions properly." [5] Any problems that occur in the use of the program from that point on must be traceable to causes outside of the program. For example, the hardware that program is running on may fail. The human user may make a mistake. The sensors and inputs may malfunction.

By completely testing the software, the reliability of the digital control system will be reduced to being only a function of the reliability of the hardware and human interaction. Hardware reliability is well-understood and modeled. Human interaction reliability with the software will be no worse that with analog systems. If the hardware reliability of the digital system is made better than the hardware reliability of the older analog systems, the digital systems can expected to be better than the analog systems.

Complete testing of digital safety software is possible given that the program can be analyzed in pieces. By dividing the program into smaller pieces, the program can be analyzed piece by piece and the analyzed pieces can be used to analyze the whole program. Dividing the program into sections and then subsections narrows the portion of code of interest down to a size that can be readily analyzed. Once the subsections are flowgraphed and independent paths are mapped, the subsections can be used to create a set of input and outputs data sets needed to test the section they are contained in.

There are several ways to completely test a program. A set of independent flowpaths through the entire program can be created and tested. Doing this requires not only knowing how many independent paths exist, but also the ability to construct a reduced flowgraph of the whole program in order to determine the paths and the necessary input data and output results sets. Even a program of the modest size of the sample program is too complex to permit creation of a reduced flowgraph. In addition, keeping track of 171 different tests presents opportunities for errors and missed tests, even if all 171 tests are defined successfully.

An easier method of testing is that of testing the program section by section and subsection by subsection, sequentially. This method requires even more tests, than the previous method, but each test is easier to define because the section of the program of interest in each test is much smaller. While the number of tests for the example program in this work increases to 232, this is more than compensated by the simplification of each test. In addition, the tester does not need to keep track of all 232 tests at once. He only need to keep track of the tests needed for the section that he is working on at the moment. The previous method may not even be possible if the control logic is very complex, while this method will work reliably even with more complex logic.

Testing program sections sequentially and also testing the program subsections within each section in parallel reduces the number of tests needed in the example program to 48. By definition, all the subsections in a given section are not interdependent of each other. They do not share any inputs or outputs. Testing the program this way adds little to the difficulty of constructing each test compared to the previous method but drastically reduces the number of tests needed, in this case by a factor of almost five.

The number of tests needed can be reduced even further without adding greatly to the difficulty of constructing each test. By testing in parallel sections that are not interdependent and by testing sequentially sections that are interdependent, the number of tests can be reduced to 30 for the sample program. This step in reducing the number of tests needed extends the procedure used in the previous method to more of the program at once. The increase in difficulty and therefore error probability is minimal compared to the previous method because the two are essentially the same, this one is just applied to more of the code at once.

Reducing the number of tests any more requires that sections that are interdependent, that have overlap in the coils and registers they use, be tested simultaneously. This can be done by means of more careful planning of the tests because the different sections have to be tested in a certain order with respect to each other in order to ensure that coils used in common in different program sections are set correctly to test the sections. Doing this carefully reduces the number of tests necessary to 17. The minimum number of tests possible is 11 because this is the size of the largest subsection with a fully reduced flowgraph. This step in test reduction almost halves the number of tests necessary, but it greatly increases the difficulty of defining each test and of creating the input data and output result sets. A decision concerning whether to reduce the number of tests this far by means of this method will have to be made with every program in order to decide whether the reduction in the number of tests offsets the increased probability of making a mistake in creating and executing the tests.

As programs become larger, the McCabe metric value increases quickly. A significant savings in the number of tests needed can be obtained by testing sections of the program that are not interdependent in parallel, or simultaneously. Further test savings can be made by carefully coordinating the testing of interdependent sections, but as the program becomes larger, doing this will become harder, thereby increasing the chances for error. The major portion of the savings in number of tests needed as compared to the number of tests required in testing every independent path sequentially is obtained by testing non interrelated sections in parallel and testing interrelated sections in series. This testing method offers a compromise between the number of tests needed and ease of setting up the individual tests.

The maximum size of a program that can be completely tested depends upon the number of tests that the tester is willing to conduct and upon how much probability of error or difficulty of setting up the tests that the tester is willing to tolerate. By conducting complete testing with a hybrid combination of parallel and sequential testing, a tester can avoid the most likely sources of error while still reaping most of the potential gains possible with parallel testing. Depending upon how long it takes the tester to run a single test, it may be quicker in the long run to use the hybrid method rather than to reduce the number of tests as far as possible and consequently to spend more time setting up the necessary input data sets.

As the number of tests is reduced, the input data sets will be more complicated and error-prone because of the need to coordinate program sections with overlapping coils. In order to avoid having errors in the input data sets and test specifications, it will be necessary to spend more time checking to see that

prescribed tests are self-consistent and complete. The example program has only one overlapping coil that must be carefully coordinated using the hybrid method. The nature of this coil is such that it is impossible to test the program without giving it special attention anyway.

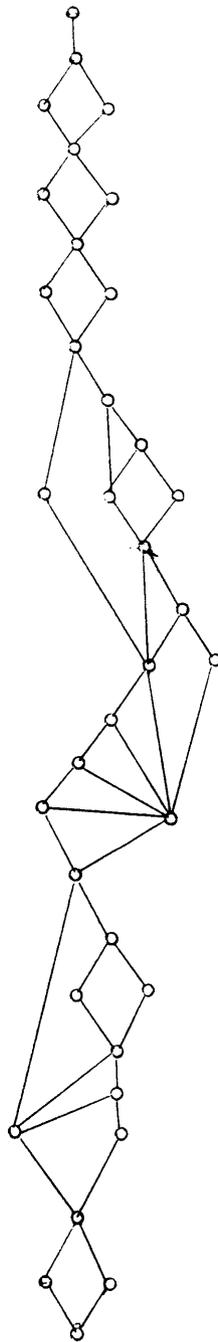
More research should be done with other programs written in other languages and of different lengths to determine how the number of tests needed with each method scales with the program length and logical complexity. Other programming languages may not be as amenable as the PLC language to constructing flowgraphs of a program's logic. Software programs written for safety-related applications should be developed with the intent to completely test the program. Developing a program with intent to completely test it should simplify the testing process and enable longer and more complex programs to be completely tested, extending the usefulness of this technique.

## References

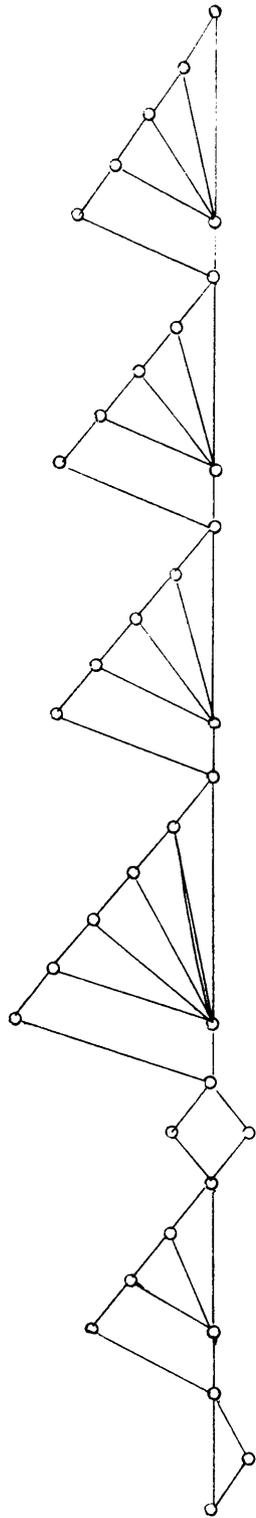
1. ABB-Combustion Engineering, Inc., Sample PLC Code, provided by ABB- C/E, Windsor, CT, 1993.
2. T.J. McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," National Bureau of Standards special publication 500-99, December 1982.
3. Modicon, Inc., Modicon Modsoft Programmer User Manual, GM-MSFT-001 Rev. B, N. Andover, MA, September 1991.
4. Modicon, Inc., Modicon 984 Programmable Controller Systems Manual, GM-0984-SYS Rev. B, N. Andover, MA, May 1991.
5. K.E. Poorman, On the Complete Testing of Simple, Safety-Related Software, Report No. MIT-ANP-TR-019, Massachusetts Institute of Technology, February, 1994.
6. M.H. Halstead, Elements of Software Science, Elsevier, North-Holland, 1977.
7. H. Zuse, Software Complexity: Measures and Methods, Walter de Gruyter, Berlin, 1991

## Appendix A

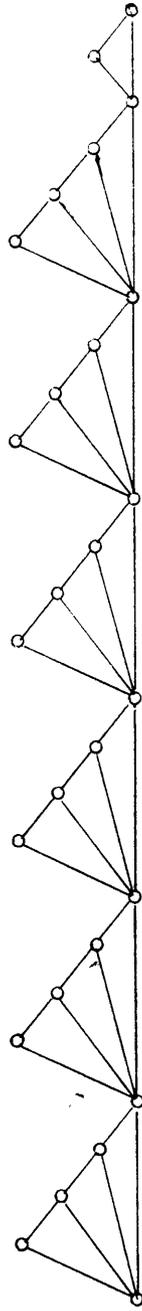
This appendix contains the flowgraphs that were generated to analyze the example program. Since the actual text of the program is proprietary, no references to actual operations performed in the program are included on the flowgraphs. The flowgraph is traversed by starting at the top and working down. The bottom node on one page is the same node as the top node on the next page.



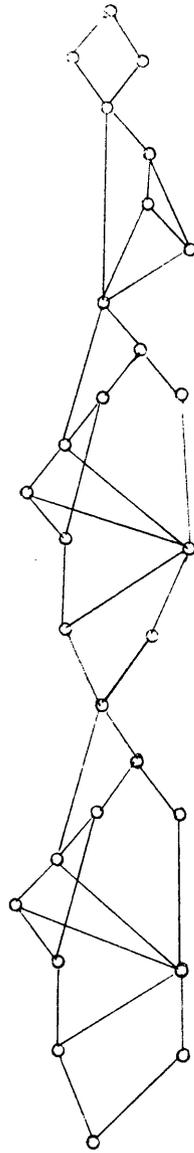
Continued on next page



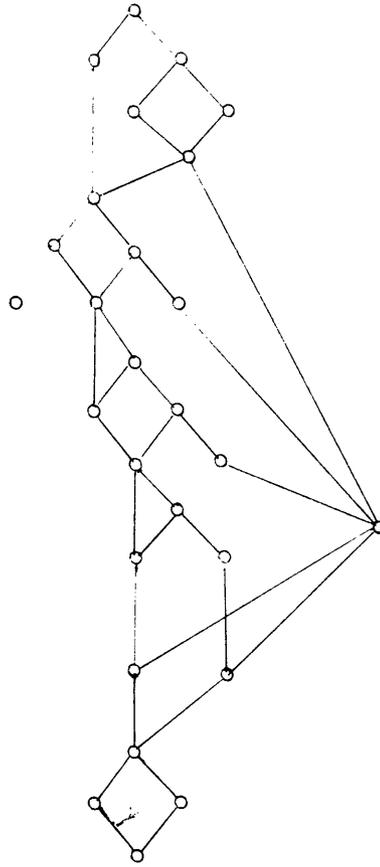
Continued on next page



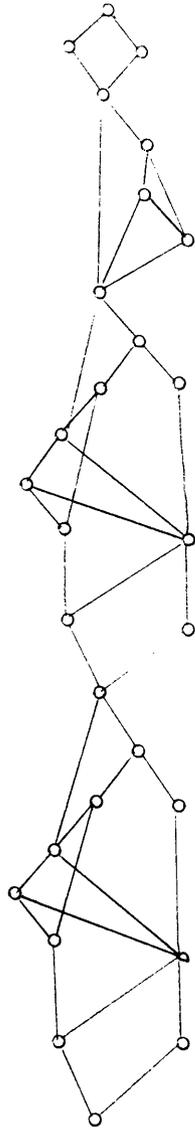
Continued on next page



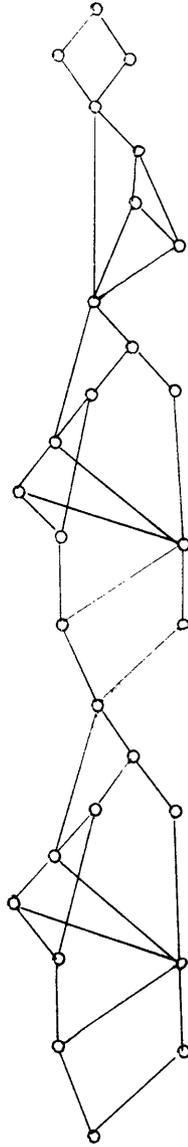
Continued on next page



Continued on next page

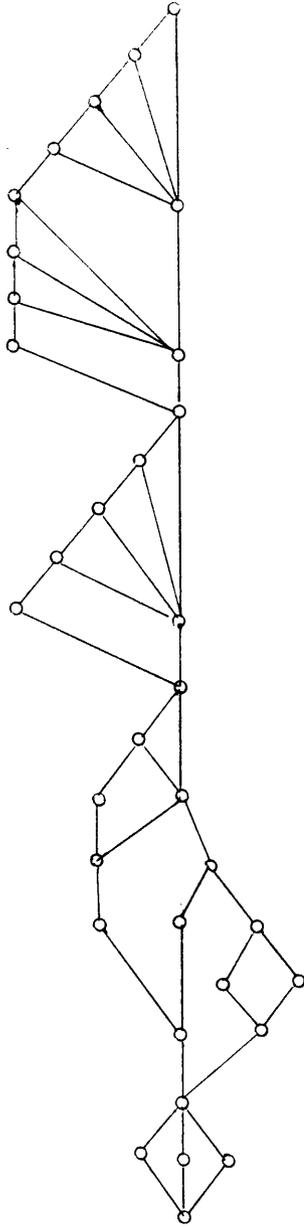


Continued on next page

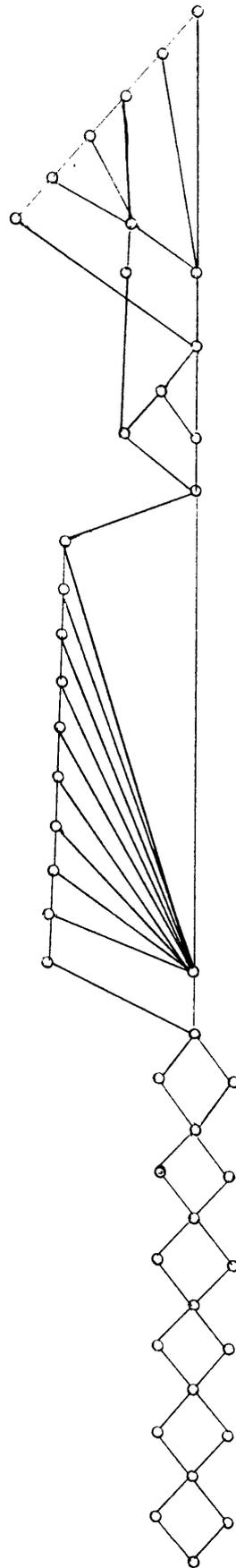


Continued on next page

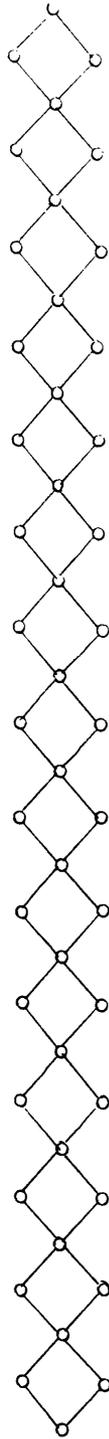




Continued on next page



Continued on next page



Continued on next page

