On the Complete Testing of Simple Safety-Related Software

by

Kenneth E. Poorman

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

Master of Science

at the

Massachusetts Institute of Technology

February 1994

Signature of Author _____
Department of Nuclear Engineering
January 12, 1994

Certified by _____
Professor Michael W. Golay
Thesis Supervisor

Certified by _____
Professor David D. Lanning
Thesis Supervisor

Accepted by _____
Professor Allan F. Henry
Chairman, Graduate Thesis Committee

On the Complete Testing of Simple, Safety-Related Software

by

Kenneth E. Poorman

Submitted to the Department of Nuclear Engineering in
February 1994 in partial fulfillment of the requirements for the
Degree of Master of Science in Nuclear Engineering

ABSTRACT

In order to facilitate the introduction of safety-related software applications into the nuclear power industry, a theoretical study on the complete testing of simple, safety-related software is performed based upon information obtained in a literature search and applied to a sample program supplied by the research sponsor. The simple, safety-related program uses two parallel Modicon 984 programmable controllers to monitor the secondary coolant, collapsed liquid levels in two steam generators as well as the pressurizer pressure of a pressurized water reactor nuclear power station. It requires two-of-two concurrence to activate auxiliary water pumps if the steam generator levels should fall below their setpoint limits, or to generate a reactor shut down signal should the pressurizer pressure exceeds its setpoint limit.

The literature search produced few results applicable to simple software testing, validation or verification. The search did produce some useful techniques, however, 2which could be applied to the sample code. A particularly promising approach was that of McCabe [McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," National Bureau of Standards special publication 500-99, December 1982]. The technique involves analyzing the structure of the code using cyclomatic, execution-sequence-based flowgraphs. By reducing the code to functional modules and independent flowgraphs, complete sets of tests could be defined for the software code.

The work reported here is restricted to the analysis to software testing, avoiding the topics of design specifications, hardware reliability and human interaction. It is assumed for this study that the systems of interest would be simple and therefore not pose great challenges in creating specifications. Hardware and human concerns are assumed to be on the same order of difficulty as those of similar analog systems.

The results of this study indicate that the complete testing of simple, safety-related software is feasible. While the scope of this study is limited, there is reason to believe that it can be applied equally well to other applications with similar conditions. If the structure of the software execution flowpath network is kept simple, there is no evident limit on the number of lines of code (i.e., PLC networks) which could be tested completely.

Thesis Supervisors: Michael W. Golay
David D. Lanning
Professors of Nuclear Engineering

Acknowledgments

The authors would like to acknowledge the generosity of the sponsors, ABB-Combustion Engineering, Inc. and MIT. This report is based on work done by Kenneth E. Poorman in fulfilling requirements for the Master of Science degree.

Special thanks go to Michael Novak, the primary contact-person within ABB-C/E, and others at ABB-C/E who made valuable contributions in the formulation of the problem and subsequent progress.

Table of Contents

Part 3

Part 4

Part 5

List of Figures and Tables

In Order of Appearance

Part 1
Introduction

1.1 General

Penetration of digital control software into nuclear power safety applications has been retarded by the difficulties of the verification and validation (V&V) process, which includes testing the software performance against the required software specifications [1]. As a consequence, the advancement of nuclear power technology has been unable to take advantage of modern information processing technologies to the extent that is common in other industries. The work reported here was undertaken to improve the ability to test nuclear safety-related software.

The development of high reliability software has been concentrated in areas of high commercial importance (e.g., telecommunications control systems) and high human risk potential (e.g., aeronautical and astronautical application of fly-by-wire technologies). Such applications typically involve very large, complex programs having many statements, program operational states and opportunities for developmental errors.

The techniques of V&V ultimately require extensive program testing, both manually and automatically. This is done in the hope that after sufficient testing all errors will have been found, or at least that any remaining errors will not appear at unacceptable frequencies for which the program is being used. An acceptance criterion which has been

used in aerospace applications is that if the expected time scale (i.e. inverse frequency) of appearance of an undetected error is substantially longer than the expected useful life of the program then the program would be acceptable.

The difficulty of acceptance of nuclear safety-related software in nuclear power applications has arisen from concerns over its reliability. These concerns are similar to those focused upon hardware and human reliability. The problem with each is to show that the expected frequency or conditional probability of correct performance is greater than a minimum threshold value. Standard methods for providing high-reliability software have not yet evolved, and such proofs are inherently difficult.

The purpose of this work is to provide some improvements to the current situation by determining whether nuclear safety-related software can be made simple enough that it can be tested completely [2]. If it can be tested completely, it is expected that it can satisfy the safety-related requirements imposed by the Nuclear Regulatory Commission (NRC).

1.2 Background

Ultimately, it will be necessary to show that the use of safety-related software can be expected to improve safety [1]. If this can be done with low uncertainty then use of such software will become imperative. However, a partial proof -- one consisting of an

expectation of highly reliable behavior, but subject to substantial uncertainties -- will likely be unsatisfactory. Incorrect software performance can arise from many sources, including the following:

- incorrect conceptual program formulation;
- programming errors, which can include:
  - transcription errors,
  - program conceptual errors (e.g., incorrectly formulated logic statements)
  - numerical method errors;
- operating system errors, or those arising from incorrect use of the operating system;
- input data and control parameter errors.

Ways of searching for errors all rely upon comparison of program foundations, features or results to independent standards. These comparisons can fail because of faulty standards, faulty comparisons or incomplete comparisons. The different types of error searches include the following:

- use of independent theoretical derivations of solutions, with independent computation of their solutions using the program being tested;
- independent checking line-by-line of program encoding for transcription and program conceptual errors;
- comparison of program results for simple cases to those obtained from independent sources, including analytic solutions, those from peer programs, and those from more accurate programs;
- independent review of numerical methods used in the program being tested, checking for solution sensitivities to numerical parameter variations (e.g., time step size changes) and

comparison of program computational results to those obtained using alternative numerical formulations;

- independent reviews of the operating system and its uses in controlling the program (because of the large experience-based characteristic of the typical operating system, however, the former is presumably a relatively rare source of error);
- independent review of input and control data to ensure its correctness, and comparison of sample results to reference result sets;
- verifying that specified input data leads to expected output results over the anticipated operational range of program use.

Several means of developing reliable software are available. These include:

- employing, where possible, simple theoretical problem treatments having few alternative logic states, which lend themselves to reliable checking of the algorithm's conceptual formulation, software encoding, and solutions;
- use of highly reliable programmers;
- use of image-oriented programming languages which have internal consistency and completeness tests and employ natural language inputs, input echoes and outputs.

In addition, the program can be structured to enhance reliability. Some ways of doing this include the following:

- minimizing program complexity
- modularizing the program, or portions of it, through the use of well-tested macros or subroutines.

1.3 Method of Investigation

There are primarily two tasks undertaken for the work reported here [1]. The first task is a literature search of both nuclear and non-nuclear industries concerning development, verification, validation and application of simple, high-reliability software. This includes an examination, with the aim of identifying the reasons for the successes and failures which have occurred, of the experiences within the international nuclear power industry and their attempts to introduce highly-reliable, but simple, nuclear safety-related software.

The second task is to examine an example of simple nuclear safety-related software in order to understand the strengths of current approaches for enhancing reliability, and to identify some means of improving both expected reliability and the ability to demonstrate high reliability.

1.4 Scope of this Work

The primary sources contributing to the system unreliability are review before defining the scope of the work presented in this report:

1. the hardware;
2. the environment (e.g., electromagnetic (EM) pulses);
3. the system program;
4. the compiler;

5. the software specifications; and

6. the testing procedure.

The work presented here is concerned only with simple software and does not include a reliability study of other potential sources of control system errors, including the necessary hardware components, environmental effects, conceptual errors, or design errors -- all of which are commonly found in more complex software codes. The software being examined is considered therefore to be fully and correctly specified, including all features and feature interactions.

Simply put, the work shows that simple software can be completely tested to satisfy all of its specifications given the very limited scope allowed for a program. If there exist unstated control system specifications, the control software code will fail to deal with them.

Part 2
Literature Search

2.1 General

The literature search of this work found that there is not much information available that specifically deals with simple, safety-related software testing, verification or validation. Rather, the majority of relevant software information deals with long, complex software programs that consist of thousands of lines of code, complicated control specifications and complex hardware or software interactions. Most of the articles indicate that software codes can not be completely tested. While this conclusion is common to much of the literature, there are various reasons for it, either stated or implied. For example, some of the arguments given for software untestability are based on the premise that software codes are too large or complicated to permit one to be certain that all possible conditions (i.e., software states) have been tested; some arguments criticize the hardware components that execute the software codes; and finally, some arguments suggest a lack of confidence in the operators' ability to use the software properly, whether or not the software is functioning correctly. One other software concern often cited relates to the user-friendliness of the code, or specifically, the complications that arise when creating user-friendly software. This is because it is usually necessary to add many features to the software codes in order to make them user-friendly that are not necessary for proper execution of the code itself, and any added features can only complicate the

code structure. While the literature uses all of these reasons to label software as being untestable, it is apparent that most of the reasons cited are not software related but are directed at the hardware, human, or control aspects of the software code.

In order to explain why these various concerns are not relevant to this particular study, a closer examination of some of the articles is necessary. An example of each case cited above is presented subsequently in more detail and a counter-argument is proposed in order to isolate that concern from this study.

2.2 Software Concerns vs. Other Concerns

Since this study concentrates on software concerns, it is necessary to eliminate other concerns from the scope of the study. Such concerns include those about the correctness of system control algorithms, the associated hardware systems, and the human operators [3, 4].

2.2.1 Concerns about Control Algorithms

In simple control cases, the control algorithms are typically well understood before any effort to encode them is begun. Regardless of whether the system being designed to implement the control algorithm uses analog or digital hardware, the reliability of software is not an issue during the earliest phases of control system development.

Particularly for the most basic, safety-related control functions, the control algorithms are well understood [5]. For example, one of the functions of the code being used for this study [6] is to turn on a pump if the water level in a vessel is reduced beyond a minimum value (i.e., the setpoint). Once the current level and the setpoint level are known, the control algorithm is extremely simple: if the current level is less than the setpoint level then the pump is to be turned on, otherwise the pump is off. For such simple algorithms, small modifications can be made (e.g., for timing purposes) without complicating the control system beyond testable limits.

2.2.2 Hardware Concerns

Another reason often cited for the inability to test software systems completely concerns the hardware that must be used in conjunction with the software code itself [7, 8]. Hardware components used for data acquisition, for example, have a limited reliability that will determine how reliably the software receives its necessary input data. For critical, safety-related software, the consequences of misinformation could be extreme indeed. Therefore, there are many who believe that even if the software code were perfect, the entire system would be limited by the reliability of its supporting hardware -- which is true. Hardware reliability, however, is not a new issue to the nuclear industry, or any industry demanding high-reliability, and it has been dealt with successfully in the past. Indeed, the very same high-reliability systems which are currently trying to introduce software replacements are entirely made up of hardware components.

2.2.2.1 Computer Hardware Problems

With the rapid development of electronic technology, very few products are on the market long enough to provide the experience base needed to certify their reliability for safety-related operations. Typically, systems that have managed to exist long enough for an accurate collection of fault data are nearly archaic compared to the newer, improved systems. Fortunately, successive generations of hardware components are often based upon earlier ones, with improvements being made from experience, so that the quality typically increases with each generation of computer hardware technology. While such trends do not guarantee future good performance, they provide indications and can be used for initial reliability estimates.

2.2.2.2 Reducing Hardware Concerns

In order to minimize concerns about faulty hardware in safety-related systems, many techniques have been developed to ensure proper operation of both the mechanical and software systems. The most promising of these developments is the distributed, fault tolerant architecture which is being investigated extensively by Hecht and Agron at Oak Ridge National Laboratory and privately as well [9, 10]. The proposed concept is based on redundant hardware systems, not only to provide multiple concurrence of critical processes, but also to have systems perform on-line testing and backup of the input data and system performance, as well as functional fault detection; the result has been a more reliable system for safety-critical operations. Hecht and Agron have developed systems

such as the Extended Distributed Recovery Block (EDRB), which is specifically designed for real-time processing of critical control processes and takes advantage of multiple (redundant and/or diverse) hardware and software components. With such a system, primary control software could be optimized for performance and economy while alternate software "provides a 'life-boat' to prevent a reactor trip in the event that the primary system fails to properly execute its control functions."

Other researchers at the Oak Ridge National Laboratory, such as Battle and Alley [11] are investigating options that go beyond the software and hardware systems and take full advantage of modern digital technology. They claim that such options involve none of the complicated verification processes which are responsible for the stall in advanced software, safety-related control applications. Specifically, Battle and Alley have been investigating the use of application-specific integrated circuits (ASICs). These integrated circuits are developed and optimized using advanced software tools, but they are only practical for simple systems. Their advantage is that since the algorithms are contained in integrated circuits, backup and restart operations are not concerns for reliability, as they are with conventional software systems. Battle and Alley claim that ASICs provide all of the benefits for high-reliability systems that digital computers offer, but very few of their complications. They claim that this is primarily due to the fact that the production and verification of ASICs is well understood, whereas the development of software control is in its infancy. Advancements in integrated circuit (IC) production technology have also made the limited production of a small number of ASICs much less expensive than it once

was. Finally, another benefit of ASICs, due to their IC form, is that they are easy to replace and can be adapted easily to most remote computer control software.

The software code used for this study [6] uses both software and hardware redundancy (2-of-2 concurrence of software control activation signals). It is assumed that the hardware industry has developed sufficient methods to ensure the necessary reliability of the control system's hardware components. The reliability thus achieved is assumed to be comparable to the reliability of similar analog equipment used for similar safety-related purposes.

2.2.3 Human Factors

Another reason often given for the inability to test software completely concerns the role of human interaction. The greater the involvement of humans in the operation of software systems, the greater are the chances of encountering human-induced faults. There are several reasons for this relationship [4,12]: 1) programs with human input must typically be more complicated than similar, human-free counterparts; 2) humans can misunderstand which information needs to be provided to the system; 3) humans can be careless and accidentally provide the wrong information (especially in mundane or repetitious operations); and 4) humans are subject to psychological stresses, causing inefficient or false operation of the system.

A simple scenario demonstrates how humans can be the root of system failures: faults can be introduced into a system when a procedure alters the operational state of the software or hardware (e.g., by putting the code into a testing mode or service mode), which is then "forgotten;" it is clear that this results in faulty operation until the condition is noticed and remedied.

While hardware and software systems can theoretically be tested completely if given enough time and money, human beings cannot. Human behavior is a science, but the subject is extremely complex. Situations which might cause hardware and software failure might also cause unpredictable stress (and responses) from otherwise reliable operators. Hope in reducing human error rates is one of the key reasons for which advanced control is being given so much attention by industries requiring high-reliability performance. While human designers are required to anticipate new operating or accident situations, human operators are often fooled by circumstances or initial impressions (i.e., humans have preconditioned attitudes and understandings). For example, the plant operators at the Three Mile Island (TMI-2) nuclear power station were victims of just such an accident [13, 14]. While observers speculate about actions which might have prevented that accident, human error is certainly one of the factors that caused it [13]. Humans are susceptible to information overload and must therefore be provided with the information which will be most beneficial for the occasion. (Ironically, the best information to provide, even with software systems, must be determined by humans who have most likely not been in similar circumstances and cannot know what information is

necessary or helpful.) In accident scenarios, too much can happen too fast for humans to react rationally, often leading to an overreaction or even no action at all.

With the introduction of advanced control, some researchers are worried that human error rates might actually increase [15]. Complacency or complicated software procedures might lead to problems that cannot be predicted under current conditions, and operational circumstances might actually become more dangerous than those that currently exist [16]. The National Air and Space Administration (NASA) has conducted studies that show how such complacency can develop and the devastating results it can have on pilots traveling at speeds greater than the speed of sound. Also, there is some worry that humans will be excluded from software control too much and that the resulting automation might actually be counter-productive, reducing the reliability of safety-related systems instead of increasing it. Supporting this claim, NASA studies have shown that humans who remain an integral part of the operating system are more likely to perform well and operate with the system most effectively [14].

The software for this study does not required much human interface [5]. Also, since the human interactions that do exist with this system are not considerably different than those existing with the analog equipment which it potentially replaces, human error does not appear likely to increase the risk to the overall performance of the system. Additionally, the majority of all human interactions for this code are related to diagnostic testing [17, 18] which is used in the hope of creating a more reliable system.

2.3 Software Issues

Henceforth, focus is placed on the software codes themselves, which are the primary emphasis of this study. The scope of eligible systems has been defined in part by the previous sections in this chapter, but eligible software codes are further limited by problems described in this section. For example, in order to aviod the specification difficulties inherent in making user-friendly software, eligible codes need to have very limited interactions with human operators [19, 20]. Also, due to the difficulty of determining test independence when there exist too many flowpaths, eligible codes are limited to a small number of flowpaths such that all independent tests are easily understood [21]. Finally, in order for software control to be a reasonable alternative to current methods of control, the reliability of software systems must be better than the alternative non-software systems [1].

2.3.1 Software Development Processes

The entire software development process has been scrutinized closely in an effort to decrease the introduction and incidence of software related faults in control systems [22]. Since complete testing of software codes has commonly been viewed as impossible, various other means have often been sought to reduce the introduction of faults into the system, requiring a plan encompassing the entire software development process [23 - 29].

The software development process begins with the establishment of required specifications, continues through the design descriptions, test plans, and user-manual development, and finally terminates when the user accepts the complete and finished product [4]. During all four phases of the development process (e.g., specification, design, testing and documentation), conscious efforts are made to include extensive evaluations, inspections, and software code walk-throughs, to be certain that all possible errors are removed. The entire process also includes various external audits and reviews.

While these software development processes reduce the introduction of software errors and are a significant step in the correct direction, they are not likely to be sufficient to completely eliminate all software errors from the system since software is traditionally a user-defined product. Since the people involved in designing system specifications are not usually the same people who eventually use the software, the specifications for user-intensive programs often contain many user-specific "errors" and must be changed (e.g., "I didn't want the screen to clear before each run"). It is also customary in the software industry to categorize these specification faults, including unstated specifications, as "software defects," even though the code executes correctly. This custom makes error-free software codes difficult to achieve.

The solution to user-specific errors adopted for this study involves limiting the human-software interaction, which makes the necessary specifications much more concrete and less likely to contain unforeseen human-related specification faults.

Additionally, more narrowly defined system specifications are possible, which are very precise and leave very little room for misinterpretation.

Another concern in general software development that is not frequently addressed is the flexibility that must be incorporated into new versions of an old code [30]. Modern software codes must conform to so many different software standards that it is not surprising that there should be so many problems. For example, an improved word processor code, written to take advantage of a modern computer processor, must account not only for compatibility with older versions of the same word processor, but also for compatibility with older computer processors. System idiosyncrasies are also a concern, such as the various operating speeds, monitor types, display attributes, keyboard configurations and printer types. Also, with the ability of having so many different software applications running simultaneously, it is no wonder that there should be occasional configuration problems.

As a result of the complexities involved in making a code universally compatible, this study limits each processor to the execution of one application and does not require that the code conform to other processor or hardware standards. As an extra precaution, the code used for this study involves a minimal amount of machine-machine interaction as well as very limited man-machine interaction.

2.3.2 Software Complexity

The most powerful and relevant argument against the ability to completely test software relates to the extremely inconsistent topic of software complexity [31 - 35]. Although the various definitions of software complexity and their methods of calculation are inconsistent, the theory behind software complexity is simple: to be able to classify software codes into groups such as "complex" or "simple" according to some common numerical scale. It is important to understand that all of the software complexity metrics which have evolved are empirical, primarily because "complexity" depends largely on human perception, and also because it is relative to specific applications; complexity is not based on firmly established mathematical, physical, or other scientific grounds. By limiting this study to simple software -- software which has relatively few execution states -- the intention is to limit the complexity of software codes so that they are easily understood and can be tested completely.

2.3.2.1 General Complexity Metrics

For many years, software complexity has been associated with the "program length." Most frequently, program length is related to the physical size (e.g., the number of lines or words) of a software code [31], though it is sometimes considered to be more closely related to the number and variety of data objects (e.g., inputs) and command statements (i.e., software commands such as the *if..then* command) used in the code [35]; in both cases, each is related to a quantity referred to as the "program length." (While

these two views might seem to be closely related, consider the case where a physically short program -- one with very few lines -- repeatedly accesses data from a remote file and manipulates it; the physical length of the program would indicate that it is unlikely to have many errors, while the size of the data file might indicate otherwise.)

Another independent theory of software complexity suggests that it should be related to the logical structure, or state diagram, of the software code [31, 36]. These two unrelated theories of software complexity, one based upon the program length (also known as "linguistically" based) and one based upon the logical structure (or "cyclomatically" based), are a result of the fact that there are no universally accepted, rigorous definitions of software complexity. Software complexity is simply one of many software tools that software programmers have developed in order to quantify the quality of their codes on the basis of characteristics which they consider important in relation to software correctness. Regardless of the method chosen, however, the objective remains the same: to create reliable software codes with fewer defects. Ultimately, a higher complexity number theoretically reflects a less desirable software code.

While, theoretically, there are only two types of software complexity, there exist many different measures of software complexity (also known as complexity metrics), developed and proposed by software programmers or theorists based on their subjective observations of specific applications. There is a large variety of software applications and programming techniques, resulting in the fact that no single software complexity metric

has been widely adopted; there seem to be conditions where each metric is particularly well behaved but no metric that is universally applicable to a full spectrum of conditions.

2.3.2.2 Halstead's Metric

The most notable proponent of a linguistically based metric, Halstead [20, 34, 35] developed an empirical formula which he uses to estimate the number of software errors (i.e., "bugs") that one can expect to find in a given software code. Like other early complexity metrics, the formula Halstead developed is based "program length," although his metric has some unique modifications. Unlike other linguistic metrics, Halstead does not presume to "count" the number of lines, or even the number of words in a code in order to determine what he terms the "program length." Instead, Halstead's method counts the number of discrete data (i.e., specific information, such as a person's height or weight) and it counts the number of discrete commands (e.g., the *if..then..else, while..do,* and *repeat..until* commands), combining each of these counts with its total number of appearances (e.g., how many references there are to the person's height or weight, or how many times the *if..then..else, while..do,* and *repeat..until* commands are used). An example using Halstead's metric is presented in Chapter 3, section 3.2.2.

Although all linguistic metrics, Halstead's metric included, are no longer considered to be valid for determining the complexity of a program, they have given meaningful insights into certain types of defects commonly found in software codes and

can often be useful in judging the potential for problems during the earliest development phases of software systems. In order to keep the value of Halstead's metric as small as possible, a minimal number of discrete commands should be used, as well as a very limited amount of data. For this study, there are no complex data files that need to be accessed, and the program length of the code is short (far fewer than one thousand lines of code).

2.3.2.3 McCabe's Metric

One of the prominent advocates of a cyclomatically based complexity metric, McCabe [31, 36] employs flowgraph theory, which is mathematically based and involves the logical structure of the executed code rather than the code's particular linguistic properties. One of the advantages of basing his complexity metric on a program's array of alternative execution flowgraphs (or "state diagrams") is the firm mathematical foundation on which the flowgraph analysis is based.

The biggest disadvantage (and source of debate) of McCabe's complexity metric in particular is largely semantic: high "complexity" values obtained using McCabe's metric do not always correspond to highly complex program structures [31, 34]. The reason for this discrepancy is that the "complexity" number obtained using McCabe's metric actually corresponds to the number of alternate ways a program can be executed (i.e., the number of "flowpaths" through the code) -- programs which can be executed in a large number of different ways (i.e., programs with a large number of flowpaths) inherently have larger

"complexity" measures according to McCabe's metric, but they are not necessarily more complicated programs (i.e., the logic for each flowpath could be easy to determine and check). To illustrate this point, it is only necessary to demonstrate that it is possible for such so-called "complex" codes (according to McCabe's metric) to be simple, as it is with the *case* statement.

EXAMPLE: consider a program that is required to display different information depending on the day of the week -- in this case, there would be seven different "cases" (Sunday through Saturday) corresponding to seven different ways (seven flowpaths) for the code to be executed. Since each of the seven flowpaths is independent (i.e., the code cannot possibly execute more than one of those seven flowpaths, and it cannot ever switch flowpaths), *case* statements are not considered to be more complex than their close relative, the *if..then..else* statement.

It is relatively clear that software codes with even more "cases" than the example above, would have higher "complexity" numbers according to McCabe's metric (the value of the measure increases by unity for each case) but the code would actually be no more complicated than a code with just two simple cases (e.g., the *if..then..else* command). Nonetheless, it is true that *case* statements require as many tests as McCabe's metric indicates, although those tests are easily determined and are therefore not "complex."

Based upon experience in developing his complexity metric, McCabe chose a metric value of 10 as the upper limit for simple, easily-testable software [36]. He has also categorized codes with metric values between 10 and 50 as "moderately complex," and those with values over 50 as "complex." McCabe justifies his selections by arguing that while metric values less than or equal to 10 are sometimes extremely simple, they are testable even in "worst case" scenarios.

For this study, McCabe's metric is not used as a measure of complexity, but rather it is simply taken for what it is: a measure of the number of tests that are required to completely test the software code. For this reason, the term "complexity" is not used this context.

If the difference between "complex" codes and "easily tested" codes has been confusing, consider this: regardless of the number of flowpaths, if the logical conditions for all of the flowpaths are easily determined (exactly what "easily determined" means is left for future discussion), then the code is "easily tested;" if the tests are not easily determined, for any reason, the code is "complex." As was discussed above, McCabe considers simple codes to be those with metric values of 10 or less (i.e., 10 or fewer flowpaths and tests required). Since McCabe's measure is actually a count of the number of tests required, his reasoning is simply that the tests for codes with 10 flowpaths are generally simpler to determine than codes with 50 flowpaths -- a reasonable expectation.

2.3.3 Software Reliability

When complete testing of software is not possible, software reliability can be calculated using techniques similar to those used for hardware reliability [30, 33]. Measures of hardware reliability are traditionally accepted, and similar procedures are proposed for software reliability. In both cases, the procedure involves accurate documentation of historical errors or problems [33]. More formally, assuming that the rate of use of a program is constant, the reliability (R) of a software code can be measured by using the mean time between failures (MTBF) and the probability of success on demand (P). The value of the MTBF can be estimated by dividing the cumulative operating time by the number of failures experienced during that time. To measure the probability of software success upon demand, one can use data accumulated from system testing. The actual software reliability can then be calculated as the product of the two values thus obtained: R=MTBF*P. The biggest problem with this type of software reliability measurement (a problem which is not present in hardware reliability) is that once a defect has been corrected, it will never again occur; also, all future defects are already present but remain undetected until accessed, unlike the hardware analogy where defects are created as a function of time.

2.4 Current Work and Progress

In order to facilitate the introduction of computers into the nuclear power industry, many large studies are being conducted to determine what tests must be performed and

what procedures must be followed in order to ensure plant safety and proper operation [37 - 40]. The Nuclear Regulatory Commission (NRC) has been working with the Electronic Power Research Institute (EPRI) to produce a guideline for new nuclear reactor power plant systems [37]. The resulting manual is expected to provide helpful guidelines to vendors and operators on which systems are "safety-related" and which are not. Such guidelines would help to ensure that the procedures followed by the nuclear power industry to update and replace its critical high-reliability systems would result in systems with more confidence and higher system reliability, and an overall reduction in licensing procedures [41]. This would ideally reduce the verification and testing costs which are currently very prohibitive.

Also working toward a more systematic approach to the refitting of nuclear power plants, the International Electrotechnical Commission (IEC), the International Organization of Standardization (ISO) and the International Atomic Energy Agency (IAEA) have been instrumental in establishing procedures and guidelines for system development [42]. The standard released as IEC 1226, "The classification of instrumentation and control systems important to safety for NPP's" will have major applications to the various digital computer systems currently used for monitoring, control and protection functions for NPPs. The current requirements for the design of both the hardware and software are graded based on the importance to safety of the systems and equipment that perform these functions. Collectively, these groups are also responsible for circulating documents which they hope will be of use in the future, to better

understand the comments others in the nuclear power industry have to make. Two documents currently being circulated are concerned primarily with the design of control rooms, the area most likely to heavily use computers: "Verification and validation of control room design of NPPs" and "VDU applications to main control rooms of NPPs."

2.5 Review

Literature that is relevant to software verification, validation and testing deals primarily with complex programs or with factors beyond the scope of this study, such as hardware or operator concerns. While the literature is helpful in limiting the scope of this study, it is not specifically applicable to the complete testing of simple, high-reliability software codes. Suggestions were found for improving the software development process, but none were found which might be used to confirm the ability to test simple software completely. The most promising case for complete simple software testing is the related technology of application-specific integrated circuits (ASICs).

Part 3
Understanding Software Structure

3.1 Nature of Software Uncertainties

The nature of software uncertainties is elusive. Unlike those of hardware, the population of latent software errors is not time-dependent and does not appear after a certain number of hours of use. Software errors are programmed errors -- they exist ab initio in the software and lurk waiting for the proper conditions to arrise in order for the errors to be revealed. However, software errors do not create themselves and are not in any way mysterious. The only way for a software error to exist is for it to elude, by whatever means, the different testing phases of the software.

With this concept in mind, it should be possible in principle to test every conceivable operational state of the software and to reveal the existence of any software errors before the software is packaged for use. In order to perform such a complete battery of tests, however, several conditions must exist: there must be relatively few tests to run (so that there is enough time to run them all before the software becomes obsolete); the test paths must be very well defined and understood; and the proper operation of the program must be well understood. Where these three conditions apply, complete testing of software should be possible.

There is a prolific amount of literature discussing the best way to create software specifications, as was discussed in Chapter 2. Simple software is in a sense classified by the ease of its specification. If there is one task to be accomplished, it is easy to specify that task in great detail, including the very limited number of ways it interacts with other components. The question that will have to be addressed at some point (and not here) is how much is too much for "simple" specification? It seems evident that the specifications will finally determine how reliably tests can be executed and software codes be verified as correct.

3.2 Complexity Measures

3.2.1 General

In order to quantify how difficult a software code is to understand and test, many researchers have been trying to determine a measure of software "complexity." There are literally hundreds of different complexity measures [31], but two are most notable since they are the standards against which most of the others are compared.

As is discussed in Chapter 2, complexity measures can be separated into two different classifications, commonly referred to as "linguistic" and "cyclomatic." Linguistic measures are those that deal with the written code itself (e.g. line counting, word counting, and data counting). Cyclomatic measures are those that deal with the logical flowgraph dictated by the code rather than the actual text itself. Both methods are under

constant revision and improvement; neither method is accepted as a best measure since each seems to apply better to different types of codes. There are also hybrid complexity measures which incorporate qualities of both methods, but to-date none is considered superior.

3.2.2 Halstead's Metrics

Halstead [13] developed his complexity metric by evaluating many independent program codes and empirically fitting error data to them. There are linguistic metrics subsequently developed which have tried to improve upon Halstead's basic premise, which leads to the conclusion that Halstead developed one of the most fundamental linguistic metrics being used to determine software complexity.

In order to calculate the number of software errors (i.e., bugs) which one can expect to find in a program code, Halstead developed a function to determine what he considered to be the "program length" and then related that length to the total number of anticipated software errors. The general Halstead metric is simply scans the entire program code and counts the number of distinct operators (e.g., program "keywords" such as *if..then* or *while..do* loops), designated by the term n_1, and also counts the number of distinct operands (e.g., variables or data objects used by the operators), which will be designated n_2. Halstead could then define his version of the "program length," designated here as H, as being a function of both n_1 and n_2 as given by Equation 3.1:

$$H = n_1 \log_2 (n_1) + n_2 \log_2 (n_2). \quad (3.1)$$

After defining the program length according to his method, Halstead further defines N_1 and N_2 which are the program's total operator count and total operand count, respectively. The sum of these two counts is known as the "actual Halstead length" and is given by Equation 3.2:

$$N = N_1 + N_2 . \quad (3.2)$$

By using a combination of all of these parameters, Halstead determined the total number of software errors one could anticipate finding in the code by Equation 3.3, which is given by

$$Errors = \frac{(N_1+N_2)\log_2(n_1+n_2)}{3000} . \quad (3.3)$$

As an example, consider a program which uses 75 discrete variables (or data objects) which are accessed a total of 1300 times. The program also uses 150 discrete operators a total of 1200 times. This program, according to Halstead's metric, would be expected to have a total of $(1300+1200)\log_2(75+150)/3000 = 6.5$ errors.

One of the greatest advantages of this type of calculation is that it can often be applied for estimates of software errors long before actual programming starts, which can aide in determining which proposed software code to pursue. To apply Halstead's formula, all that is required is some general knowledge of the data base to be used and the

code operators that will be needed, as well as some general estimates of the procedures that are to be employed.

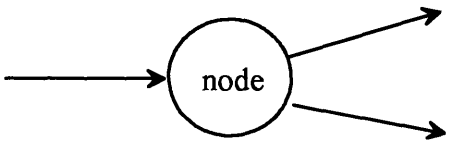
The biggest disadvantage to this type of measure is that it is very empirical and becomes unreliable as the programmers learn how to "correct" for the errors that Halstead has observed. Also, there is no guarantee that there will be only 6.5 errors in this particular code (e.g., testing cannot be terminated simply because the anticipated number of errors have been revealed), and experienced programmers might have far fewer errors than anticipated (e.g., testing cannot continue until the anticipated number of errors are found). This metric is an excellent tool to use as a means of comparison with other codes being considered, however, or as a means of "simplifying" the present code by reducing the number of operators being used or the number of times data is accessed, for example.

3.2.3 McCabe's Metric

As discussed briefly in Chapter 2, McCabe [14] developed a method for determining the relative complexity of a software code that is not dependent on any formal definition of the "program length." In fact, McCabe indicates that codes with greater numbers of lines actually tend to be less complicated per unit length than shorter, more compact codes.

Instead of counting the number of lines in a code, McCabe found that using a flowgraph of the code structure resulted in far better results and could then be directly applied to the test development phase. McCabe's metric yields a number which he terms "complexity" but is essentially a measure of the number of tests required to completely test the structure of the code. McCabe's metric itself does not reveal much about the tests which must be executed, but there are relatively simple procedures which can be employed for specific types of code, especially if the number of flowpaths is kept to a minimum.

FIGURE 3-1
Sample node of a flowgraph



McCabe's metric involves breaking the program code into simple units of function (which can be thought of essentially as Halstead's program operators). Each function performed in the program code is represented as a circle, or "node," as shown in Figure 3-1. The lines shown entering and exiting the node represent the number of inputs and possible outputs for that node, thus indicating the possible flows of information through it.

In order to link several nodes together, one must determine the relationship between the nodes. It is important to keep in mind the fact that the links between nodes

are not necessarily consistent with the physical layout of the program code and may depend exclusively upon the logical conditions necessary to execute the function represented by that node. The theorem of flowgraphs used as a foundation McCabe metric states that for a strongly connected set of nodes, the number of flowpaths (F) is a function of the number of links (L) and the number of nodes (N), as shown in Equation 3.4:

$$F = L - N + 1. \tag{3.4}$$

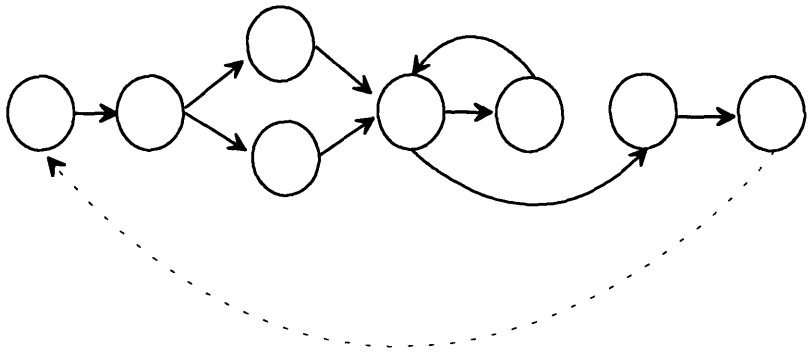
A "strongly connected set of nodes" reflects the fact that a completely closed loop exists -- every node has at least one input and one output. In practice, a strongly connected set of nodes can be created by connecting the first and last nodes (shown as a dashed line) for calculation purposes only, as shown in Figure 3-2. As an alternative, McCabe introduces a specialized relationship, shown in Equation 3.5, accounting for the fact that the first and last nodes are not connected (but could be), and his formula therefore applies to "otherwise" strongly connected nodes:

$$M = L - N + 2 \tag{3.5}$$

EXAMPLE: To demonstrate that both formulas produce the same result, apply Equation 3.4 on Figure 3-2. Evident from the flowgraph, there are 10 links (counting the dashed line) and 8 nodes, which results in F = 10 - 8 + 1 = 3 flowpaths. By removing the dashed line, the set of nodes is altered from a "strongly connected set of nodes" to an "otherwise strongly connected set of nodes." McCabe's formula applies to an otherwise strongly

connected set of nodes. The number of links has been reduced to 9 but the number of nodes has not been changed, so Equation 3.5 gives M = 9 - 8 + 2 = 3 flowpaths.

FIGURE 3-2
A strongly connected set of nodes



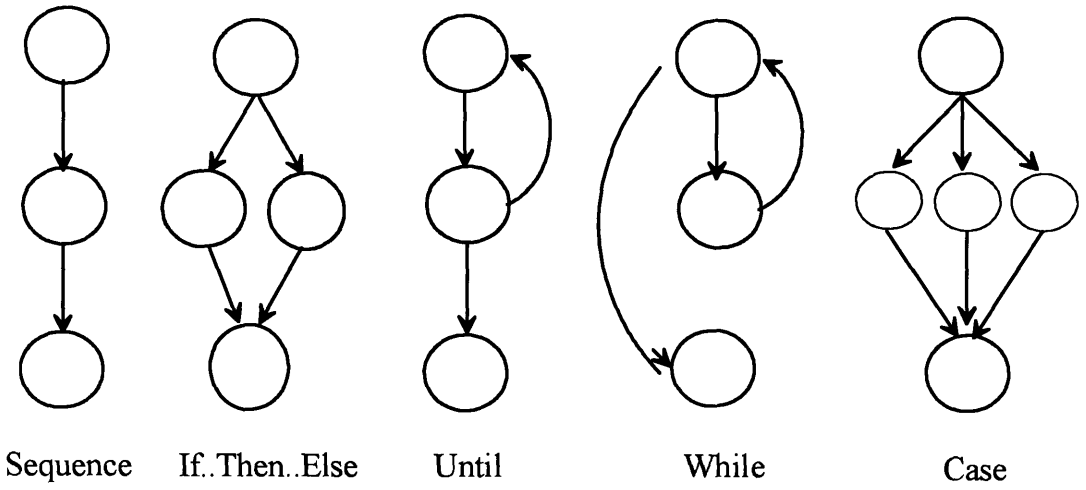
3.2.3.1 Simple Structures

There are five functional structures that McCabe recommends that programmers use in creating their code. The five general functions include: *sequence*, *if..then*, *while..do*, *until..do*, and *case* statements, as shown in Figure 3-3. The reason for choosing these particular functions is the ease with which they can be modeled using flowgraphs. If the entire program consists only of these five functions, it can quite often be simplified by grouping the functions together into modules.

There is one function in particular that is troublesome and not recommended for use in general programming: the *conditional goto* statement. Seen frequently in the BASIC programming language, the *conditional goto* statement "jumps" around the code

arbitrarily at the will of the programmer, often creating flow discontinuities which cannot be reduced into modules. While *conditional goto* statements often reduce the linguistic metric values obtained by Halstead's, similar benefits are not present when evaluated using cyclomatic metrics.

FIGURE 3-3
Graphic model of the five basic command structures



McCabe also describes a metric for "essential complexity" which can be applied to a set of nodes grouped into a module. In this manner, a set of connected nodes is simplified to an equivalent "large" node representing the entire module, where the modular "essential" complexity is a function of the number of inputs and outputs to the module, according to Equations 3.4 and 3.5. For example, each of the structures shown in Figure 3-3 has an essential complexity value of unity even though the "internal" complexity values range from M=1 (for the sequence) to M=3 (for the case statement). The essential complexity is useful in reducing the general code complexity and is valid because, for a

given set of inputs, the output of the module is known in the same way that the output of a single functional node is known. Where the links on a functional node are scalar, representing the logical state of one variable, the links on a modular node are vectors, representing possible combinations of variables where each input state determines a specific output state.

Using the essential complexity, the structural relationship of modules is determined in the same was as the structural relationship of the nodes is determined. McCabe found that applying this technique to modules, as well as within modules, results in codes which are considerably more reliable.

3.3 Flowgraphs

McCabe uses the logical structure, or flowgraph, of a code in an effort to determine the level software of complexity, but flowgraphs are also used by computer programmers to "map" the execution of software codes. McCabe relates the complexity of software code to the number of flowpaths through the it, and that number is the same as the number of tests which are required to exercise each flowpath. The benefit of flowpath analysis is that, although actual software programming methods are more of an art than a science (it is unlikely that two programmers would write code in the same way, even if they were given exactly the same set of specifications), the development of flowgraphs for software codes is relatively straight forward and independent of the art

employed. That is not to say, however, that every set of specifications has a fixed flowgraph, and like mathematical equations, a set of flowgraphs can be functionally the same while their physical layout and appearance are not. Furthermore, while each flowgraph has an easily determined number of "independent" flowpaths through it, there may be an infinite number of "dependent" flowpaths associated with them.

A firm understanding of the nature of flowgraphs is necessary in order to fully comprehend the arguments made in this thesis; what follows is a review of some of the more basic flowgraph properties. A more thorough examination of flowgraph theory can be found in McCabe's document published by the National Bureau of Standards.

3.3.2 Flowgraph Theory

It has already been discussed that every node of a flowgraph represents a module, function or instruction that gets executed in the code. The links leading into the node represent the conditions required to execute the node while the links leading out of the node indicate the possible logical results of the function. There are several possibilities to consider when dealing the exit links, though they can all be dealt with by examining two cases: 1) one exit, and 2) more than one exit.

Where only one link exits the node, the condition must always be "TRUE" and there is no corresponding conditional statement required. This type of node consists of

functions which perform tasks having no logical result (e.g., print commands or algebraic functions).

Where two or more links exit a node, one link must correspond to "all cases not represented by other exiting links." For example, two exit links would represent an *if..then..else* statement where one link represents the TRUE condition of the statement and the other link represents all other conditions (e.g., the *else* condition). Where three or more links exit a node, the situation is that of a *case* statement and one of the cases must include all conditions not otherwise specifically covered. Also note that there may never be more than one TRUE exit link for any given pass, so that there is only one discrete path corresponding to every set of input conditions.

A series of linked nodes which leads from the first node in a flowgraph to the last node is referred to as a flowpath, with the number of independent flowpaths in an otherwise strongly connected flowgraph being given by Equation 3.5. A close, personal examination of the flowgraph might indicate that there are additional flowpaths, but Equation 3.5 insures that they are not independent flowpaths. Therefore, if you execute all of the independent flowpaths, all possible flowpaths (including all dependent flowpaths) will at least have been exercised piecewise. Figure 3-4 illustrates an example of this relationship between independent and dependent flowpaths. For this case, there are 12 links and 9 nodes, leading to only 5 independent flowpaths. A close visual examination reveals that there are a total of 6 possible flowpaths, however, as outlined in Table 3-1.

The discrepancy in the number of flowpaths indicates that there is one dependent flowpath. (Do not let the simplicity of this system of nodes be misleading, however; it is not always so easy to determine the total number of possible flowpaths through a system of nodes, much less determine which of them are dependent or independent.)

FIGURE 3-4
A simple flowgraph illustrating dependent flowpaths

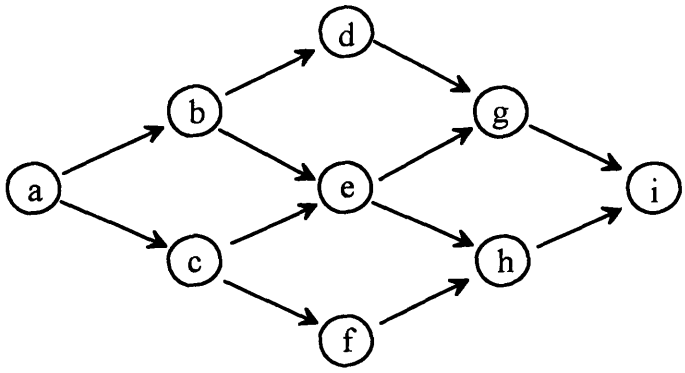


TABLE 3-1
A list of the possible flowpaths for Figure 3-4

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

Examining Figure 3-4 reveals that paths 1 and 6 are independent, which only leaves the other four paths to consider for dependence, three of which must be independent. It turns out for this example that any three of the four are independent and the last is a function of the other three. For example, consider paths 2, 3, and 4 as the independent flowpaths. Path 5 can be expressed mathematically as a linear combination of the others: path 5 = path 4 + path 3 - path 2.

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

In the same way, path 2 is a function of paths 3, 4, and 5; paths 3 and 4 can be similarly defined as functions of the remaining flowpaths. A more elaborate method of determining the independent flowpaths was described in detail by McCabe, which is now summarized.

3.3.2 Determining Independent Flowpaths

The method described by McCabe to determine an independent set of flowpaths is in very mechanical in nature and is therefore easily employed. Note, however, that even using this mechanical method many different independent sets of flowpaths can be generated. (For the simple example shown in Figure 3-4, there are five different, though equivalent sets of independent flowpaths).

The first step in establishing an independent set of flowpaths is to determine the "basepath" that will be used. Since any of the possible paths can serve as the basepath, this is where most of the alternate sets are eliminated. For this step, McCabe suggests picking a flowpath that exercises the greatest number of nodes, as this facilitates the ease with which subsequent flowpaths are determined.

The next step in determining independent flowpaths is to locate the first node which has a conditional split, and follow the alternate path, rejoining the basepath as quickly as possible. The flowpath thus defined is independent. In the same way, again following first alternate flowpath at the first node, exercise the second conditional split, rejoining the already-tested flowpath as quickly as possible. This process is repeated until all conditional splits in the alternate flowpath have been exercised, or the alternate flowpath has rejoined the basepath.

After all the first node's alternate flowpaths have been fully exercised, the basepath is followed once again until the second conditional split is encountered, at which point the alternate flowpath is followed, again rejoining the basepath as quickly as possible. As before, the second alternate flowpath is exhausted, until in the end, every conditional split encountered along the basepath and all alternate flowpaths have been exercised. Note that if a flowpath joins another previously tested flowpath, the current flowpath ends.

Although determining independent flowpaths might be confusing (and lead McCabe to limit the number of independent flowpaths to 10 for simple software), an example with clarify it. For this example, refer once again to Figure 3-4. Each of the steps will be explained again as they apply to this example.

First, a basepath is chosen from the list of possible paths. Since it is not necessary (and is sometimes impossible) for all possible flowpaths to be known, simply start at the first node and choose one of the two paths randomly, making similar random choices at each node encountered, with a conscious effort to make the longest possible flowpath. In all cases (if the links are properly followed), the resulting flowpath can serve as a basepath. For this discussion, the basepath will be path 3 of Table 3-1, consisting of nodes **a+b+e+h+i**.

The second step is to exercise the first conditional split, which occurs at node a, creating an alternate path which rejoins the basepath quickly. This leads to the flowpath **a+c+e+h+i**, which is path 5 of Table 3-1. The next step is to follow the alternate flowpath, exercising each conditional split encountered until it rejoins the basepath, but in this case, there is only one conditional split encountered, yielding the flowpath **a+c+f+h+i**. If there had been other splits along the **a+c+f** path, or the **a+c** path, they would have been followed next. Having no other possibilities, however, the basepath is followed again until the second conditional split is encountered at node **b**, which yields path **a+b+d+g+i**, which again terminates alternate flowpath because there were no conditional splits

encountered along the way. Following the basepath once again, this time to the third conditional split at node **e**, the result is the flowpath **a+b+e+g+i**, which not only ends that alternate path but also the baseline path since there are no more conditional splits in either. Thus, using path 3 as the basepath, the other independent paths selected by this process are (in the order they were encountered) paths 5, 6, 1, and 2.

Choosing a different initial basepath, for example path 6 of Table 3-1, and following the same steps results in paths 3, 2, 1 and 5 respectively -- exactly the same paths that resulted from path 3 as the basepath, although that is not always the case. For example, choosing path 4 as a basepath (not part of either example above), results in paths 2, 1, 6, and 5 respectively, after following the necessary steps.

The same process works well for more complicated structures too, although the nesting of alternate paths can become quite complex and often leads to confusion. That is the primary reason McCabe chooses to limit the number of flowpaths to less than 10.

3.4 Testing Methods

McCabe indicates that in order to completely test a software code, each flowpath must be exercised and verified. It is not enough simply to ensure that each node can be reached, however, or that each node can be exited. While it is true that for each node every entrance to it must be exercised as must every exit, it is not necessary to exercise

every combination of the entrances and exits. Only the independent flowpaths, as determined in Section 3.3.2, need to be tested, and the test conditions necessary to execute each flowpath can be obtained by examining the flowgraph.

As an example of determining the necessary test conditions, refer once again to the flowgraph shown in Figure 3-4. To aide in discussion, each of the testable nodes (which could be viewed as variables) will be named. In particular, the nodes that need names are those with multiple exit links, where specific conditions are applied to each exit link: here, let node **a** test the logical state of Boolean variable "A," nodes **b** and **c** will both test the logical state of Boolean variable "B," and node **e** tests the logical state of Boolean variable "C." (Note that nodes **b** and **c** could also test two different Boolean variables with no logical consequence.)

Once named, the logical conditions required for all necessary flowpaths can be specified for each of the three variables. Table 3-2 shows the logical definition of each flowpath for Figure 3-4, corresponding to the respective paths listed in Table 3-1.

In reference to Table 3-1, note that node **e** (variable C) is never encountered, so that the conditions necessary to test flowpaths 1 and 6 appear only to include the states of variables A and B. For the remaining flowpaths, however, the condition of variable C must also be specified. In order to illustrate the feasibility of this point better , let us examine a possible "real" application of this system. Note that since any system of three

(or four) variables which are related as shown in Figure 3-4 would be suitable, and that the example presented here is only one of many possibilities.

TABLE 3-2

Logical definitions of flowpaths for Figure 3-4

Flowpath	Condition of A	Condition of B	Condition of C
1	0	0	N/A
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	N/A

EXAMPLE: Consider a software code designed to "poll" three digital responses and reproduce the response that occurred 2 out of 3 times. This system would have as input three variables (A, B, and C) whose values were all either unity (1) or zero (0). The output of the code is a single value equal to either unity or zero. With the definitions of variables given in Table 3-2, each time node **g** is reached the output of the system is zero, while node **h** corresponds to a value of unity -- as a result, the function of node **g** is to

make the value of the output zero, and the function of node **h** is to make the value of the output unity.

3.4.1 Flowpath Tracing

Unfortunately, a computer programmer cannot be content just with the fact that the output of the code corresponds to expectations -- there are too many cases where the correct answer is obtained in the wrong way. However, by analyzing the answer as well as the flowpath, a programmer can be certain that the code executed correctly and produced the desired output. Verifying that the correct flowpath is executed, however, is not always a simple task and it can require using laborious code tracing techniques which are often very confusing.

One method that can be used to determine the actual, executed flowpath takes advantage of the equation-like properties of flowgraphs, which properties are readily apparent in Table 3-1. By equating the flowpath to the sum of the nodes involved, it is possible to derive a value associated with each node such that the value of the sum of the nodes executed is equal to the value of the path taken. For example, using the flowgraph of Figure 3-4 and the path definitions of Table 3-1, it is possible to associate a value with each node and have the sum of the node yield the path number: since both **a** and **i** are common to all paths, let them both equal to zero; for path one, **b** equals the opposite of **g**, so set both to zero and let **d** be unity; then for path two, **e** must equal two; for path three,

h is found to be unity; for path four, **c** is determined to be two; and finally, for path six, **f** is three. If each node then adds its own value to the tracer variable, the executed path number will result.

A whole series of tests could then be designed to exercise each of the independent flowpaths. One disadvantage of this tracing method, however, is that all possible flowpaths must be accounted for, not only the independent flowpaths which are produced by the techniques discussed in Section 3.3.2.

In order to avoid laborious searches for all possible paths, however, there are other tracing methods available. Assigning prime numbers to each node and using the product of the path nodes is one such alternative. Another alternative is to create an output file which could record the progress of the flowpath, indicating the execution of each node encountered. In all cases, it would be necessary to compare the actual flowpath and the expected flowpath as well as the actual and expected outputs.

Part 4
Simple Software Example

4.1 PLC Software Code

For many years, industry has been trying to integrate software codes into their high-reliability systems and there has been a considerable amount of research done toward that end. One of the results of these investigations was the use of relay ladder logic [2, 25]. Programmable logic circuits (PLCs) are used to implement the code in relay ladder logic style, and it therefore has many of the same benefits as pure relay ladder logic. There are also several properties of PLCs that make them particularly attractive in light of complete software testing, as will now be discussed.

4.1.1 PLC Characteristics

For high-reliability systems, the most valuable characteristic of PLC code is its execution method [43, 44]. Since relay ladder logic exercises each command during execution, the possibility of accidentally nesting a command such that it is not executed properly is minimal. Also, this method of execution eliminates loops of all types from the code, including the possibility of infinite loops.

Secondly, the logic of execution is entirely discrete, as with traditional, electronic logic circuits. Thus, although the code has the ability to access and manipulate memory

locations during execution (e.g., to perform arithmetic), the execution of the code is based upon the discrete values of specified input states. This allows the code to be mapped logically in the same way that integrated circuits are mapped (e.g., Carnot mapping).

Finally, PLC codes (as employed by the nuclear industry) have limited access to input data. This is accomplished by having the input buffer updated only once per scan (once per execution). By ensuring that none of the input values is altered during execution, the output of the code is essentially determined as soon as the input states are known. This fact makes the output of the code easy to associate with the corresponding input data set and to validate, since each input set can be mapped directly to an output set.

4.1.2 PLC Structure

The structure of PLC code is organized into a multi-level execution matrix [43]. On the lowest level, a single PLC node is executed. (Unfortunately, the term "node" is used in both flowgraph theory and PLC structure, so a structure node will be referred to by the term "PLC node" as much as possible to reduce confusion.) The next level is a set of networks, which together comprise the third level, identified as a segment.

A PLC node is analogous to a memory location -- it's an intersection of the current column address and the current row address. There are a variety of simple commands that require only one PLC node address, while other commands occupy several PLC nodes.

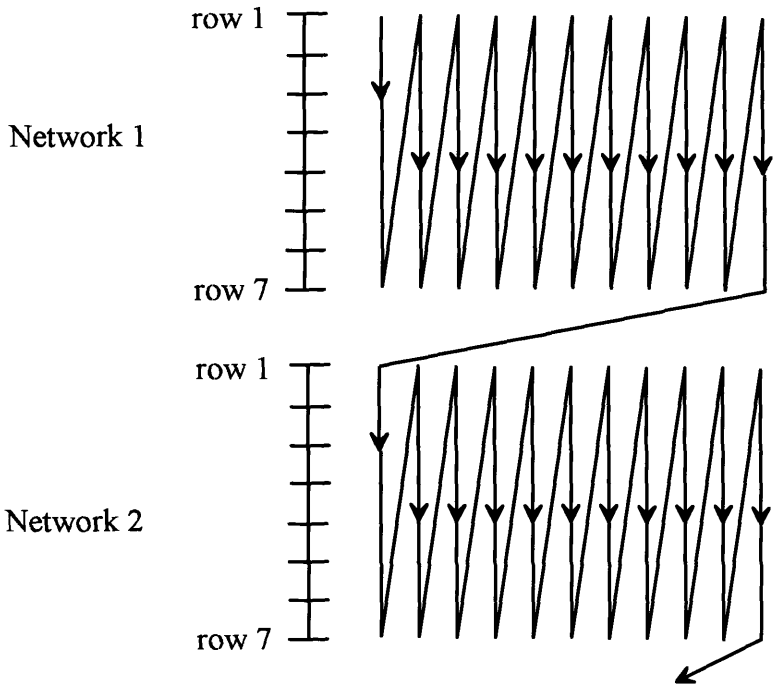
In general, each command occupies the same number of nodes as there are inputs or outputs associated with the command. The PLC nodes are organized into 7 rows and 11 columns which together form the next level of the PLC structure, the network.

Networks are the second level of the PLC code structure and each one consists of 77 addressable PLC nodes organized as described above. Each network is evaluated node by node before proceeding to the next network. The PLC nodes within the network are executed from top to bottom, left to right, as shown in Figure 4-1 taken from the Modicon Manual. Every network and PLC node is evaluated each time the code is executed unless the "skip" command has been used in the previous network. (Using the skip command is not recommended, however, and if it is used, it should be done sparingly and carefully.) A complete functional set of networks forms the next level of the PLC structure, which is called a segment.

PLC segments are executed according to their arrangement in the order-to-solve table (called the "segment scheduler") which is in system memory. As a whole, the segments are executed in their pre-assigned order, executing each network within that segment and each node within every network. Segments can be thought of as individual programs, or even modules of a larger program.

The sample program being evaluated for this study, which was supplied by the research sponsor, only employs one PLC segment with a total of 73 networks.

FIGURE 4-1
Execution order for nodes and networks



4.2 Conversion of PLC Codes into a Flowgraph

Due to many of the characteristics of the structure of PLC codes, especially those mentioned in the previous section, PLC codes develop a unique flowgraph structure when they are "translated" into flowgraphs similar in appearance to those described by McCabe and discussed in the previous chapter. Initially, the flowgraphs derived from PLC codes appear to be complicated or meaningless, but duplication of testing is the primary explanation. Realizing this, the flowgraphs derived from PLC codes can be simplified or structurally reduced, allowing them to be more accurately evaluated using methods such as those described in Chapter 3.

Although not necessary for PLC codes in general, it is important to recall that for the high-reliability codes like that being evaluated for this report, the input to the code is received at only one point, staying fixed it for the remainder of the code's execution. This property allows all of the input variables to be held constant for the entire execution of the code, thereby eliminating the need to "double check" variables for a change of state. This property allows the code to assume a form consistent with established flowgraph theory.

4.2.1 PLC Function Modules

Like other programming languages, PLC ladder logic has a limited command function vocabulary, and each function can be "modularized" into an equivalent flowgraph form. For example, a subtraction routine like the one taken from the Modicon Manual, shown in Figure 4-2 (which occupies three PLC nodes of a network and corresponds to the three possible outputs) can be represented as a single command node in flowgraph form with three possible exit cases: one exit representing a positive value, another exit representing a negative value, and the third exit being equal to zero.

Since PLC logic is discrete, each of the values exiting is either unity (1) or zero (0), but actually only one of the three outputs can be satisfied and therefore be equal to unity. In fact, for every command function, one and only one exit can be equal to unity; the remaining outputs must be equal to zero. Besides the exit logic, the memory value

associated with "difference" has also been altered by the subtraction subroutine. The full flowgraph that would represent this subtraction routine is given in Figure 4-3.

FIGURE 4-2
Example of PLC command function

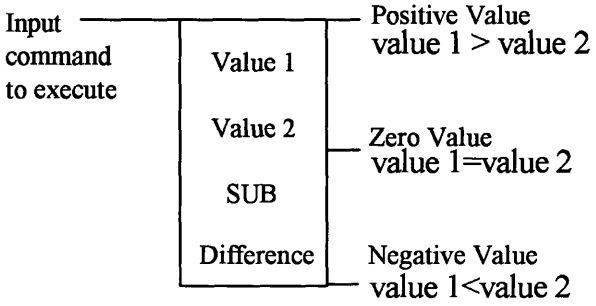
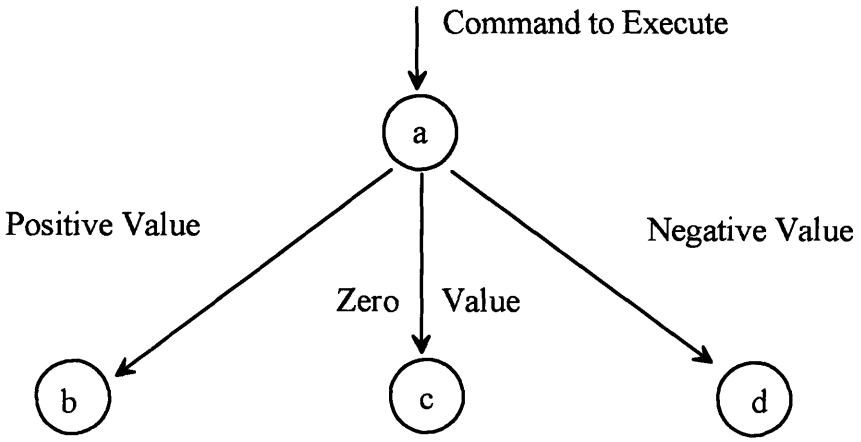
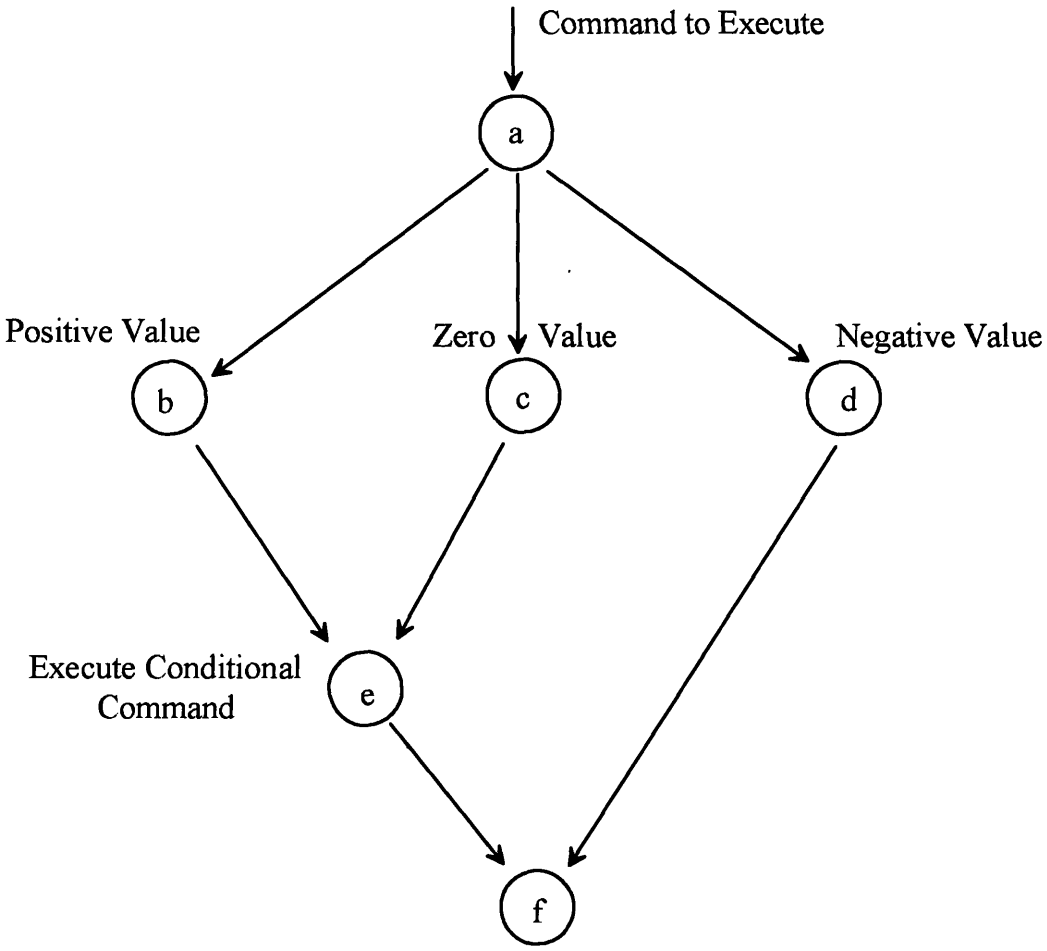


FIGURE 4-3
Flowgraph of a subtraction module



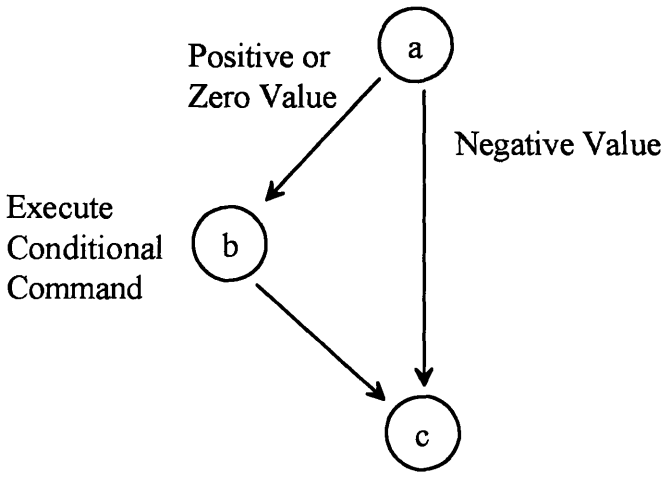
If the general structure of the program were known, the command function flowgraph shown in Figure 4-3 would appear where all subtraction modules are used for a comparison between numbers (though not where it was used simply to perform the subtraction of two numbers with no logical result). Due to the nature of outputs and their functional uses, which allows several of the outputs to do nothing while those that affect the program are often used to perform a common task, it is tempting to combine output links together. For example, if a conditional command function following the subtraction command were only to be executed if the result of the subtraction were greater than or equal to zero, the exits associated with positive and zero values could then be linked to the conditional command function while the exit associated with the negative value could be linked directly to the next command in the ladder logic. Figure 4-4 illustrates the more elaborate, "extended" flowgraph, while Figure 4-5 shows the same logic in a simpler, function-only form. While the functional flowgraph of Figure 4-5 produces all of the results associated with the extended flowgraph of Figure 4-4, it does not include the explicit boundary conditions associated with the extended flowgraph. The extended flowgraph, however, allows for a more accurate count of the number of tests that need to be executed for a complete test of the flowgraph network including all boundary conditions. Notice that in the simplified flowgraph, there are only two outputs: one corresponding to the conditions leading to the execution of the conditional command, and one corresponding to "all others." The extended flowgraph clearly indicates that three tests are required. Due to its explicitly clear form, the extended flowgraph is preferred for high-reliability systems over the simplified flowgraph.

FIGURE 4-4
Flowgraph for a subtraction module with conditional command



In order to understand better the complete flowgraph and how to derive it from the PLC code (so that its correctness can be scrutinized), the flowgraph models for each of the additional components in the sample program are introduced. This is necessary so that the overall PLC code can be modeled as a McCabe-format flowgraph.

FIGURE 4-5
Simplified functional flowgraph equivalent to Figure 4-4



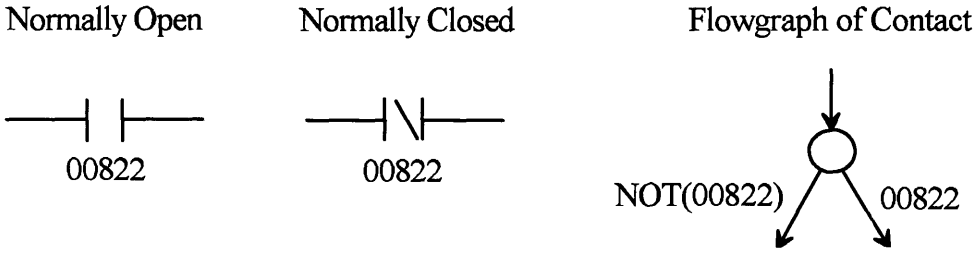
4.2.2 Contacts and Coils

In order to describe the flow of logical signals from one command to another, PLC logic uses a series of contacts and coils which can assume either of the two logical states. In the software code, both of these constructs occupy only one PLC node.

Contacts are logical "switches" that can be normally open or normally closed. A normally open contact passes a logical unity signal if the value of the memory location being accessed is equal to unity, while a normally closed contact passes a value of unity if the value of the memory location being accessed is equal to zero. Figure 4-6 shows these two contacts as they appear on PLC code and in flowgraph form. Note that the standard terminology for logical variables dictates that "NOT(00822)" means that the memory

location 00822 has a value of zero while "00822" means that the memory location associated with it has a value of unity. Note also that both contacts have the same flowgraph model -- the type of contact distinguishes which link to follow normally closed contacts take the "00822" path while normally open contacts take the other.

FIGURE 4-6
Normally-open and normally-closed contacts with flowgraph model



Coils in PLC ladder logic represent a module used to alter memory values by placing the logical value reaching the coil through the logical "circuit" in the appropriate memory location. Coils are often used to "set" or "reset" memory locations within the logical framework of the PLC code which can then be used in future contacts (in the same or a future network) or the value can be passed to other programs through communications ports. (Note that "set" means to place a value of unity into the appropriate memory location while "reset" means to place a value of zero into the appropriate memory location.)

In order to demonstrate the use of contacts and coils, as well as their translation into flowgraph form, let us look at an example network from the sample code [6]. Figure

4-7 illustrates a simple PLC code (network #31) employing five contacts and five coils.

Figure 4-8 is the flowgraph corresponding to the PLC code in Figure 4-7.

FIGURE 4-7
Simple PLC code employing only contacts and coils

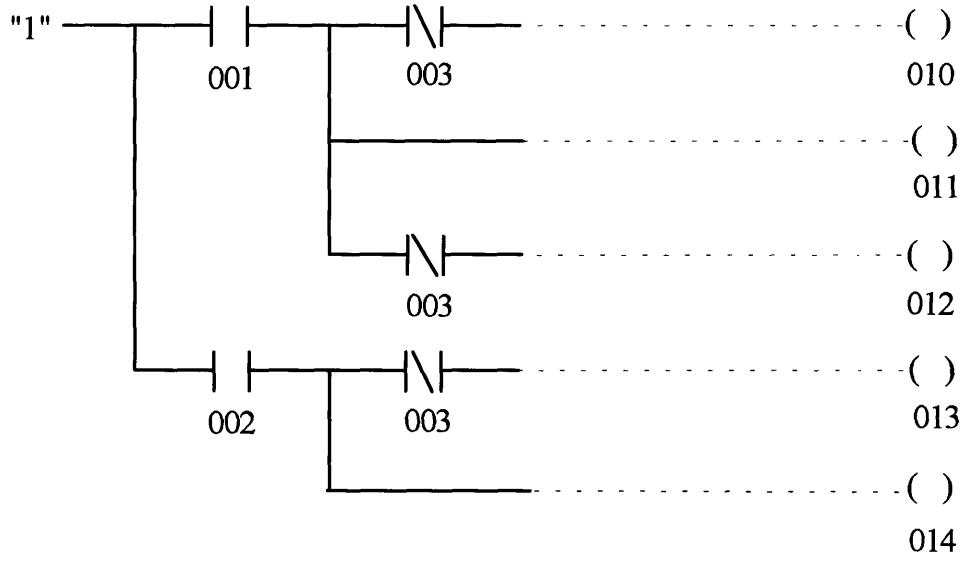
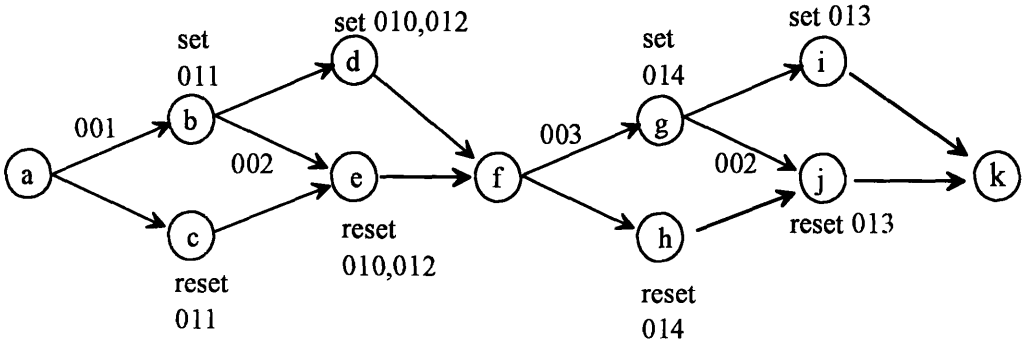


FIGURE 4-8
Flowgraph for PLC code displayed in Figure 4-7



In order to proceed to more complicated codes and flowgraphs, this simple example must be reviewed thoroughly and understood completely. It is important to remember that one property of the PLC code is that a logic "circuit" is not always evaluated as a whole before others begin to be evaluated. Nonetheless, the order of their execution and final result can be found by the position they occupy on the network "grid." PLC logic makes it easy to ascertain what the timing of the "circuit" will be, so that the flowgraph can connect the entire logic "circuit" together in its appropriate location. To understand this concept better, a closer examination of the flowgraph of Figure 4-8 is now presented.

According to the original code and PLC execution, the first command to be executed is the normally open contact labeled "001." Following the determination of this contact, the code evaluates the contact labeled "002." Then the contact labeled "003" at the top of the next column is determined (using the output of the connected contact), which is followed in turn by each contact below it in the same column, each time using the outputs determined in the previous column as appropriate. Finally, the coils 010, 011, 012, 013 and 014 are executed in that order. (In each case the order is because of their relative positions, not their numbers). So, although the individual "circuits" are not executed as connected wires, their timing is carefully controlled such that coil 010 is determined first, followed by the other coils in order.

The order of execution can be very important to some codes, and in those cases PLC codes offer great control to the programmer. In Figure 4-8, node **b** indicates that the coil "011" is set and node **c** indicates that coil "011" is reset. Note that this can be done here because the logic used to determine the configuration of coil 011 is a subset of the logic used to determine those of coils 010 and 012. While it is true that the state of coil 011 has essentially been determined, the actual memory manipulation does not occur until after that of coil 010 has been established.

The nodes on the flowgraph of Figure 4-8 indicate the following states and actions:

NODE **a**: Here, the contact evaluates the state of memory 001. If the value is unity, the next node to be evaluated (logically) is node **b**. Otherwise, the value of 001 is equal to zero, and node **c** is executed.

NODE **b**: This node represents the logical condition "001" and determines the state of coil 011 to be unity, though the coil cannot be energized yet due to timing conditions. The exit of this node is a contact determined by the state of memory 003. If 003 is equal to zero then node **d** is executed, otherwise node **e** is next.

NODE **c**: This node represents the logical condition "NOT(001)" and determines the state for coil 011 to be equal to zero, although the actual memory location is not yet altered. The exit of this node has no contact and leads directly to node **e**.

NODE **d**: This node represents the logical condition "001 and NOT(003)" which determines the value of coils 010 and 012 to be equal to unity. Coils 010, 011 and 012 can all be set to unity at this point. The exit of this node leads to node **f** where the next logical circuit begins.

NODE **e**: This node represents the logical condition "003" which determines the value of coils 010 and 012 to be equal to zero. The three coils can now be placed into their proper states: 010 is reset to zero, 011 is set to unity or reset to zero (depending on the previous node), and 012 is reset to zero. The exit of this node leads directly to node **f** where the next logical circuit begins.

NODE **f**: This node is the beginning of another logical circuit. The states of the previous circuits (and coils) have no effect on this one, so all paths lead to this single node. Technically, the order of these two circuits could be switched because they are independent, but the in the code the coils of the second circuit are programmed to be determined last (arbitrarily) so this circuit appears last in the flowgraph. The exit of node **f** is determined by a contact evaluating the memory location "002." If the value in that memory location is equal to unity then node **g** is executed next, otherwise node **h** is next.

NODE **g**: This node represents the logical condition "003" and determines the value of coil 014 to be equal to unity. However, because coil 014 is to be established last, the memory is not set at this point. The exit of this node is determined by a contact evaluating

memory location 002, which was previously evaluated at node **b**. If memory 002 has a value of unity then the circuit proceeds to node **j**, otherwise it proceeds to node **i**.

NODE **h**: This node represents the logical condition "NOT(003)" and determines the value of coil 014 to be equal to zero, but the memory is not reset at this time because coil 013 has not yet been determined. The exit of this node goes directly to node **j**.

NODE **i**: This node represents the logical condition "002 and NOT(003)" which determines the value of coil 013 to be unity and sets it. At this point, coil 014 can also be set. Note that memory location 002 was already evaluated at node **b** so that if node **d** was executed, so must node **i** be executed from node **g**. The exit of this node is the terminal node **k**.

NODE **j**: This node represents the logical condition "002" which determines the value of coil 013 to be equal to zero and resets it. Here, coil 014 can also be set or reset depending on the previous node. The exit of this node is the terminal node **k**.

NODE **k**: This is the terminal node for this code. No coils are set or reset, and the next node would be the first node of the next network.

As this elaborate description shows, it is sometimes more convenient to show the state of "determined" coils before they are actually executed in the code. Care must be

taken, however, that logic depending on those states do use it until they have actually been set physically. Also, while the coil appears at the end of the row, the logic actually manipulates it where the dashed line begins.

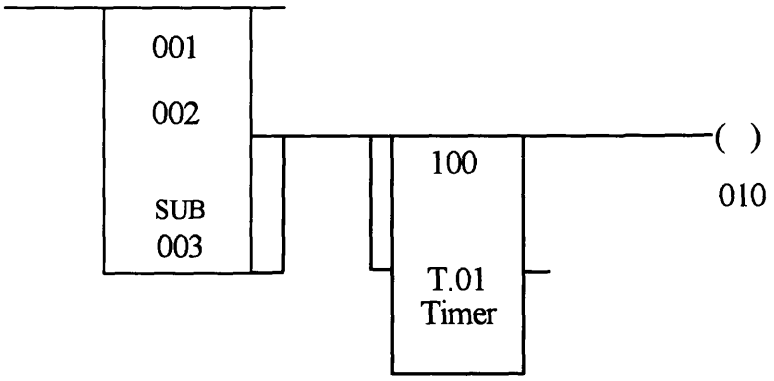
4.2.3 Modules and Functions

More complicated programs naturally include functions other than contacts and coils. Examples of functions and modules frequently employed are the subtraction, addition, multiplication, division, MOVE, and timer modules. There are a few other functions which are less frequently used, but all of them use flowgraph techniques similar to the subtraction module which has already been discussed. Figure 4-9 illustrates part of the sample PLC code (a portion network #29, among others) which executes a subtraction function and one other, in this case a timer module. The result is saved as the state of a single coil. Figure 4-10 is the complete flowgraph for the logic code shown in Figure 4-9.

Before the flowgraph is presented for this code, its function should first be understood The subtraction module, when executed, will subtract the value in memory location 002 from the value in memory location 001 and store the result in memory location 003. If the result was positive (i.e., value 002< value 001), the uppermost terminal (not connected to anything in this case) would be true (also referred to as "energized.") If the result is zero (i.e., both values are equal), the center terminal would

be energized, and if the result is negative (i.e., value 2> value 1), the bottom terminal would be energized. (This is described in detail earlier in this chapter.)

FIGURE 4-9
PLC code of subtraction module with timer function and coil



In this case, if the result is either zero or negative (i.e., value 001 is less than or equal to value 002) the lead to the timer will be energized. (Unlike traditional electronic circuits, the input to the timer is not linked directly to the subtraction module; rather, the input to the timer module is not received until the timer node is executed in its turn.) When the program examines the timer (remember, other events could take place below the subtraction routine, as will be demonstrated soon), both timer inputs would be energized since they are linked together (i.e., they are "short circuited"). The timer module is controlled by two inputs: 1) the top input indicates whether the timer should be counting (the timer counts if the input is equal to unity); and 2) the lower input resets the timer if it has a value of zero or enables the timer if it has a value equal to unity. Thus, in order for the timer to count effectively, the lower input must be enabled and the upper input

must be counting (e.g., both inputs must be equal to unity). The two outputs of the timer indicate the status of the count with respect to the preset timer value: if the upper terminal is equal to unity, it indicates that the accumulated time is equal to or greater than the value stored in memory location 100, while if the lower terminal equals unity, it indicates that the accumulated time has not yet reached the value stored there. (Note that if the upper terminal is equal to unity, the lower terminal must be equal to zero; the opposite is also true.)

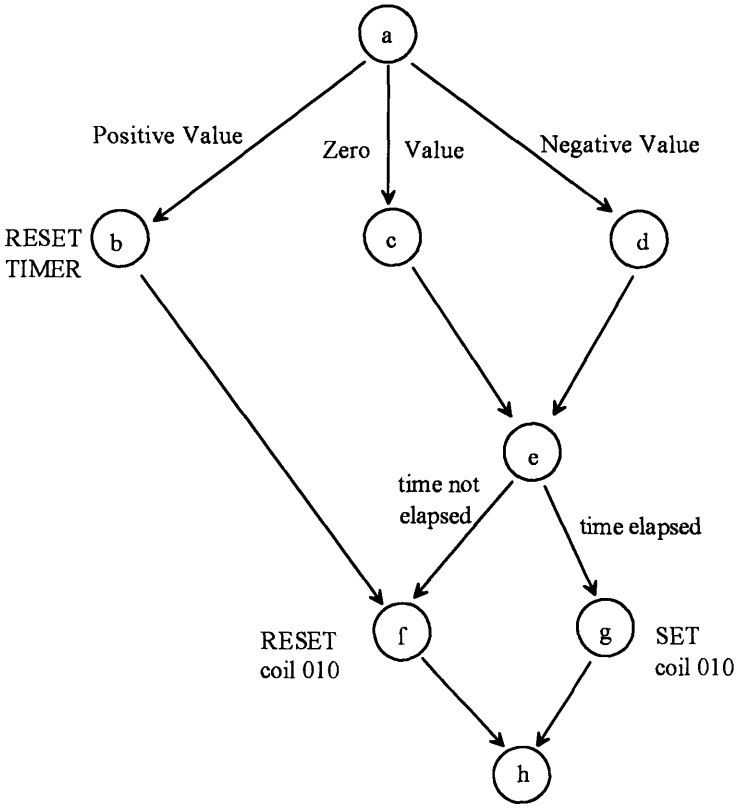
For this circuit, coil 010 indicates whether the timer has "elapsed" the preset value stored in memory location 100. If coil 010 has a value of unity, it indicates that the preset time has elapsed, while a value of zero indicates that time has not elapsed. More globally, the timer counts only when the value stored in 002 is larger than or equal to the value stored in 001, so that coil 010 actually indicates that the value 002 has been equal to or greater than value 001 for a time equal to or greater than the value stored in location 100.

The flowgraph shown in Figure 4-10 illustrates the states that must be passed for coil 010 to be set or reset. Clearly, the value of the subtraction module cannot be positive (i.e., value 001 greater than value 002) and result in setting coil 010, so that path seems to reset coil 010 directly -- the actual reason coil 010 resets is slightly more complicated: since the timer module resets and stops counting when it does not get an input of unity from the subtraction module (e.g., when it receives a value of zero), the timer's output, which is linked to coil 010, is reset. Functionally, these conditions are the same, but the

issue once again is timing. Remember that a flowgraph does not always reflect timing (e.g., when a coil is actually executed), but rather indicates if the module is executed and what results are possible.

It is important to note here that between executions the states of inputs and outputs are not changed. Therefore, until the subtraction routine is executed again, the input to the timer will remain as it was previously, either counting or not counting. The interval between executions can be determined by timing the entire segment, which naturally depends on the length of the segment and the routines that are executed in it.

FIGURE 4-10
Flowgraph for the logic circuit of Figure 4-9



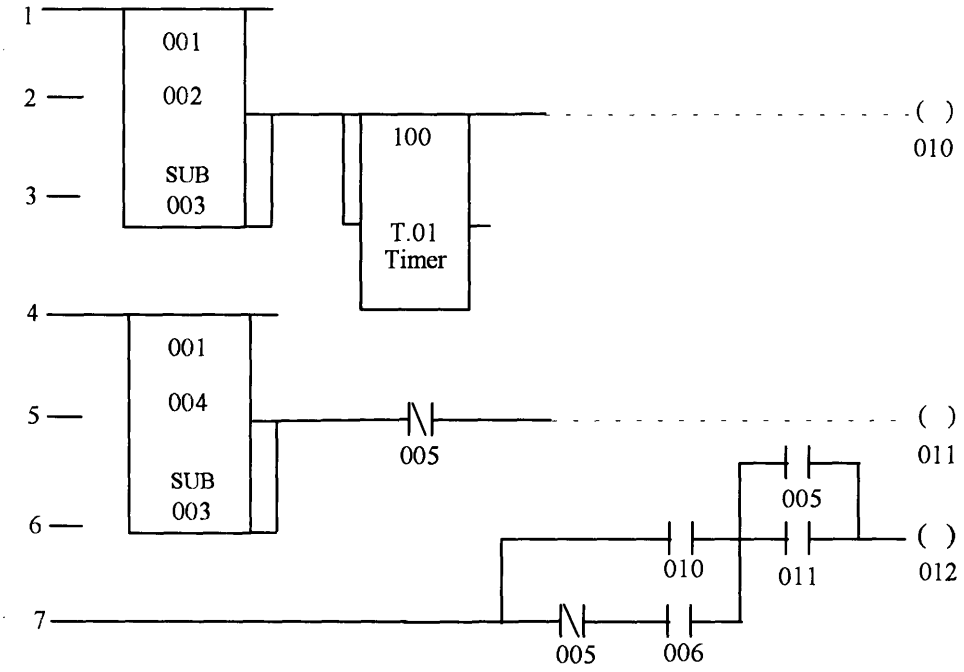
4.2.4 Complex Networks

As networks become more complex, it is reasonable for timing to become more crucial. For example, if the subtraction and timer modules discussed in the previous section are found working on the same network as another subtraction module with a conditional command, as well as a complicated array of contacts and coils, the order in which the coils are determined is significant. Figure 4-11 illustrates a complicated network taken from the sample PLC code (network #29) which has all of these properties. Future discussions will be related to this more complex network.

Eventually, this network and others must be executed together, for which set the overall performance of the code must be included. Section 4.3 will introduce methods developed by this research which help to simplify the flowgraphs created here into clean McCabe-format flowgraphs.

As the circuit diagram in Figure 4-11 makes apparent, several of the memory locations are accessed more than once, and each time the same memory location is accessed, the value must be the same since they are fixed for the duration of the segment's execution once the set of input/outputs network have been completed. Also note that the settings of coils 010 and 011 are established before they are used in the lower contact array, as is required.

FIGURE 4-11
Complex network with multiple "parallel" logic circuits



It is perhaps not clear that each of the three logic circuits is executed piece-wise rather than as a whole. Recall that this PLC software determines the value of each PLC node column-wise first. Thus, the subtraction module of circuit one is executed to determine the values of each of its PLC exit nodes (e.g., column 1, rows 1-3), and then the subtraction module of circuit two is executed to determine its exit values (e.g., column 1, rows 4-6); finally, the PLC node corresponding to circuit three (e.g., column 1 row 7) is detemined to be unity becuase it is linked directly (i.e., "shorted") to a source of unity value (i.e., column 0, also known as the "hot wire"). The PLC nodes of the second

column are then determined, with those directly related to the three circuits simply becoming the value detemined by the links to column 1 -- all PLC nodes not specifically defined are reset to zero since they have no link to the previous column. In column 3, the input for the first circuit (column 2, row 2) is used by the timer module to determine its outputs (column 3, rows 2-3), while the second circuit defines its output by using a contact to evaluate the value at memory location 005; the PLC node corresponding to the third circuit is simply carried over from column 2 again (i.e., it performs yet another "short"). The determination of the PLC nodes in column four establishes values for coils 010 and 011 (i.e., column 4, rows 2 and 5). Similar processes continue until the value of coil 012 has been established in column 7.

Clearly, the flowgraph for this network cannot be constructed as piecewise as it is actually executed, but neither is that necessary for an accurate model. Since the PLC nodes for the first circuit are the first to be determined within each column, before they can potentially affect other circuits, it can clearly be modeled first. Likewise for the second circuit, although their mutual independence makes this choice arbitrary. The third circuit, however, uses memory values manipulated by coils of the first two circuits, so it must be modeled last.

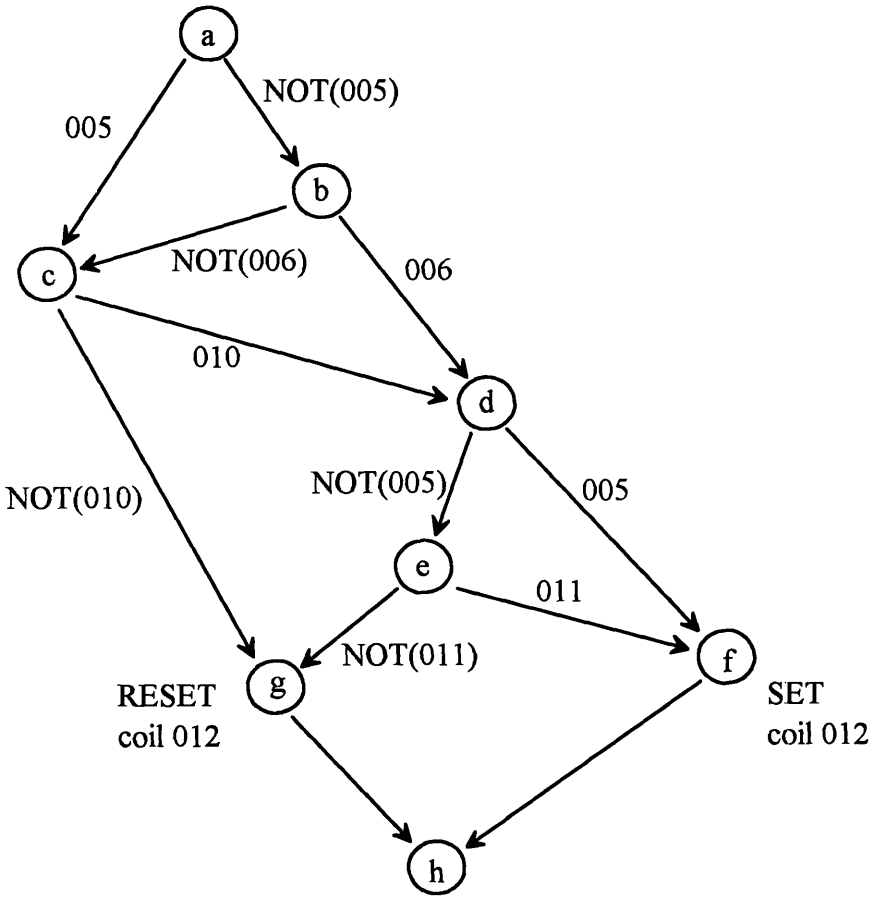
The flowgraph for the first circuit has already been presented and explained, so for this discussion it will be ignored until those of all three circuits are brought together again.

The second circuit has a flowgraph similar to the first, where the timer is simply replaced by a contact which evaluates the state of memory location 005, connecting the NOT(005) branch to the 011 coil. Figure 4-12 illustrates the flowgraph for the third circuit.

When all three of the individual circuits have been modeled as appropriate flowgraphs, they can be linked together serially with the terminal node of the preceding circuit functioning as the start node for the next circuit. As a note, this is the same way that individual networks are linked, so that each of these circuits could have been given its own network with no overall functional change. However, before the third circuit is connected to the other two circuits necessary for proper operation, a closer examination should be made in order to be sure of its correctness.

Note first that the flowgraph of the third circuit has two nodes evaluating the condition of memory location 005. Assuming, for the sake of argument, that this flowgraph has a format compatible for use with McCabe's metric, the metric would give this flowgraph a value of six (i.e., there are apparently six independent paths which require six independent tests, as discussed in Section 3.2.3). For a flowgraph to be compatible for use with McCabe's metric, however, no condition (i.e., memory value) can be tested more than once. The multiple appearance of memory value 005 is one indication that this particular flowgraph network could be simplified, eliminating at least one of the paths and reducing the number of flowpaths to five.

Figure 4-12
Flowgraph for the third logic circuit of Figure 4-11



4.3 Flowgraph Reduction

Before the flowgraphs of the individual circuits (or networks) are linked serially, each flowgraph should be reduced to its simplest, though complete, form. One of the criteria for this simple, complete form is that there must be no flowpaths which are

impossible to execute. As is noted in the preceding section, it is sometimes possible to reduce the flowgraph directly derived from the PLC code. In some cases, this flowgraph reduction also results in a simplification of the PLC code.

Where impossible paths occur, the first step is to evaluate the underlying logic that is required in order to define the circuit. For the third circuit of Figure 4-12, this logic is given by the relation

$$\text{coil } 012 = \{[\text{NOT}(005)*006]+010\}*\{005+011\}, \tag{4-1}$$

which can also be written as

$$\text{coil } 012 = \text{NOT}(005)*006*005 + \text{NOT}(005)*006*011 + 010*005 + 010*011, \tag{4-2}$$

where the algebra being performed is binary multiplication (AND) and binary addition (OR). It is clear from Equation 4-2 that the first term, NOT(005)*006*005, is an impossible path since the memory at location 005 cannot be both unity and zero simultaneously (represented in the flowgraph as "a+b+d+f"). The remaining terms of Equation 4-2 dictate that

$$\text{coil } 012 = \text{NOT}(005)*006*011 + 010*(005 + 011), \tag{4-3}$$

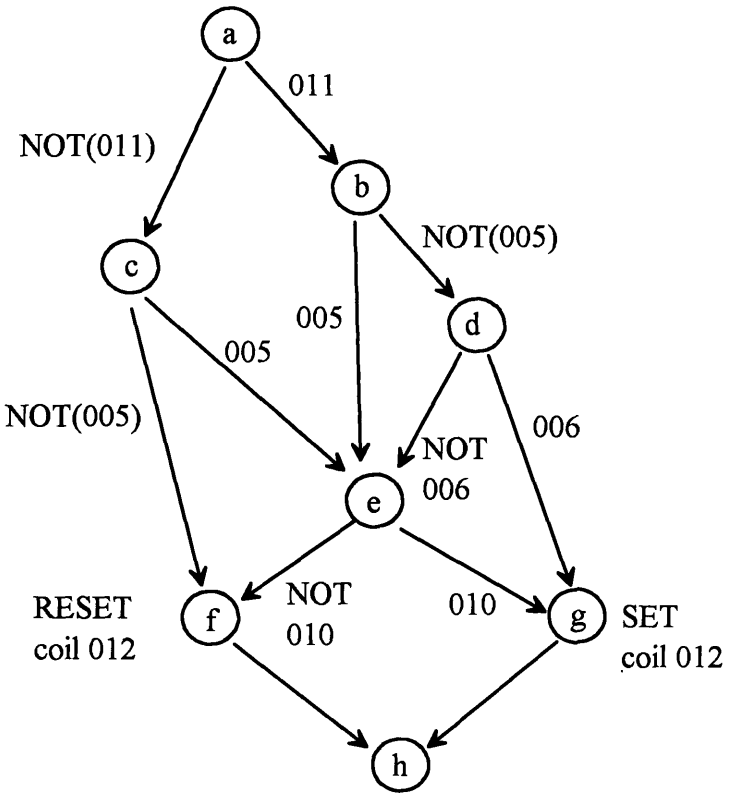
which can also be written as

$$\text{coil } 012 = 011*(\text{NOT}(005)*006 + 010) + 010*005, \tag{4-4}$$

where all of the term seem to be valid.

A new flowgraph, shown in Figure 4-13, can be created using the Equation 4-4, which is equivalent to the Equation 4-2 but does not include the impossible flowpath. It would seem that this equation eliminates the impossible flowpath, but since the total number of flowpaths has not changed, an impossible flowpath must still exist in the logic.

FIGURE 4-13
Rearranged flowgraph for third logic circuit of Figure 4-11



What is not clear from the flowgraph in Figure 4-13 is the subtle fact that coil 011 incorporates into it the condition NOT(005), making the link from node **b** to node **e** impossible to traverse. Since this is the case, node **b** can be eliminated altogether, which reduces the total number of flowpaths and required tests by one, as shown in Figure 4-14.

Once each flowgraph has been reduced as far as possible, and is therefore as simple as it can get (i.e., there are no impossible paths), all related flowgraphs can be connected together serially, as shown in Figure 4-15. In the case of the three flowgraphs presented here, recall that both coils 010 and 011 are used in the third circuit.

FIGURE 4-14
Simplified flowgraph for the third logic circuit of Figure 4-11

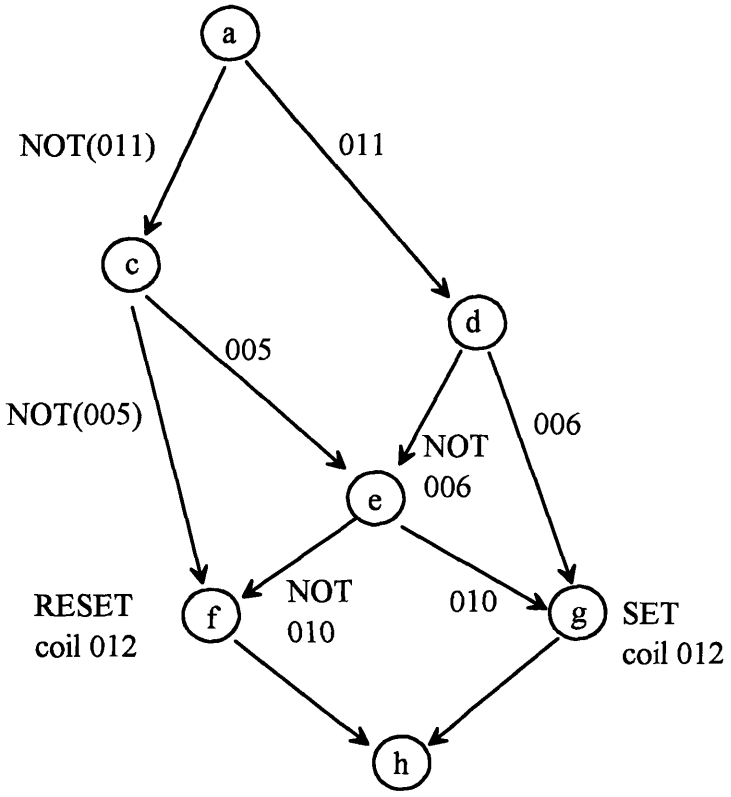
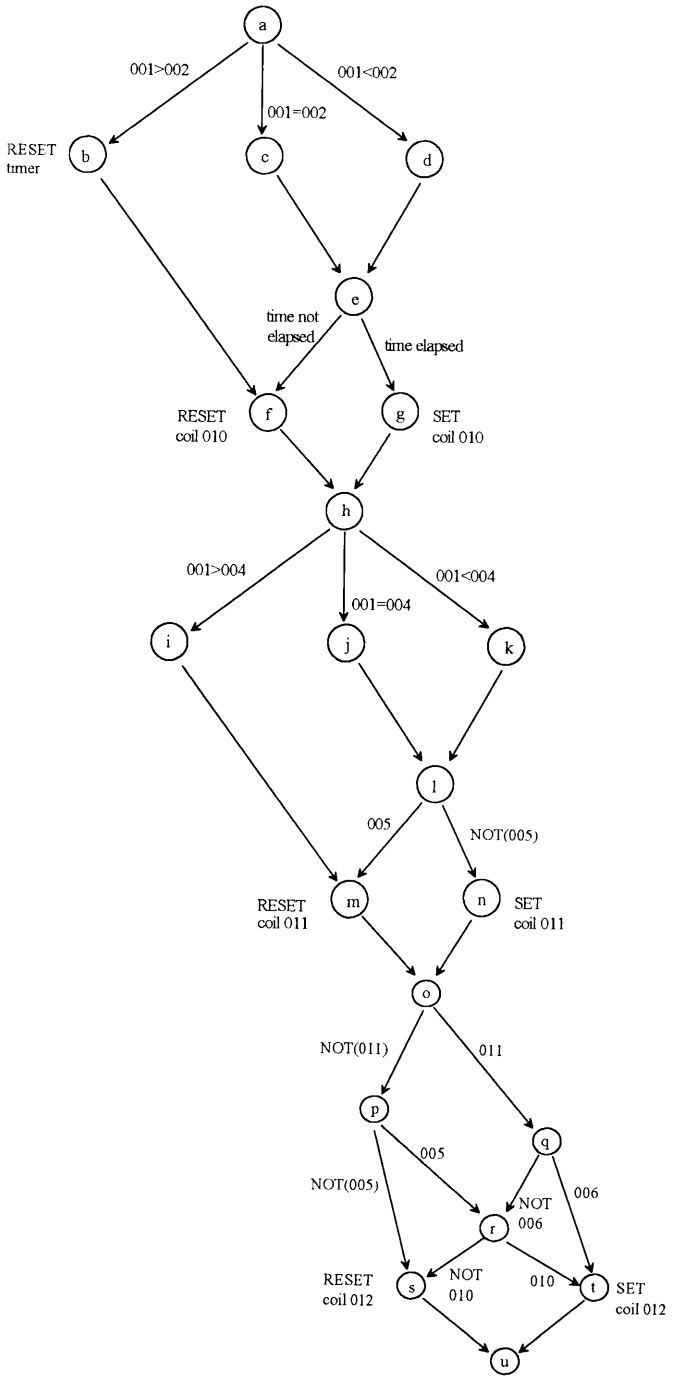


FIGURE 4-15
Flowgraphs for logic circuits of Figure 4-11 connected serially



Once all of the related networks have been linked together in the appropriate order, further simplification of the composite flowgraph might be possible. In this case, the initial flowpath count (which falsely assumes that the overall flowgraph is in a correct McCabe format) indicates that there are 11 independent flowpaths due to its 21 nodes and 30 links. However, once all three individual flowgraphs are connected, redundant nodes are once again introduced. In order to eliminate the newly introduced redundancies, it is important to note which nodes are extraneous.

There are two primary cases to consider when checking for redundant nodes, as has been discussed previously: 1) a contact re-evaluates a memory location without the value of that memory location having been altered, or 2) a contact evaluates the value of a memory location previously established by a coil. Since both of these cases are essentially the same (the logical value of the memory being checked is already known), the only distinction is the form they take in the flowgraph.

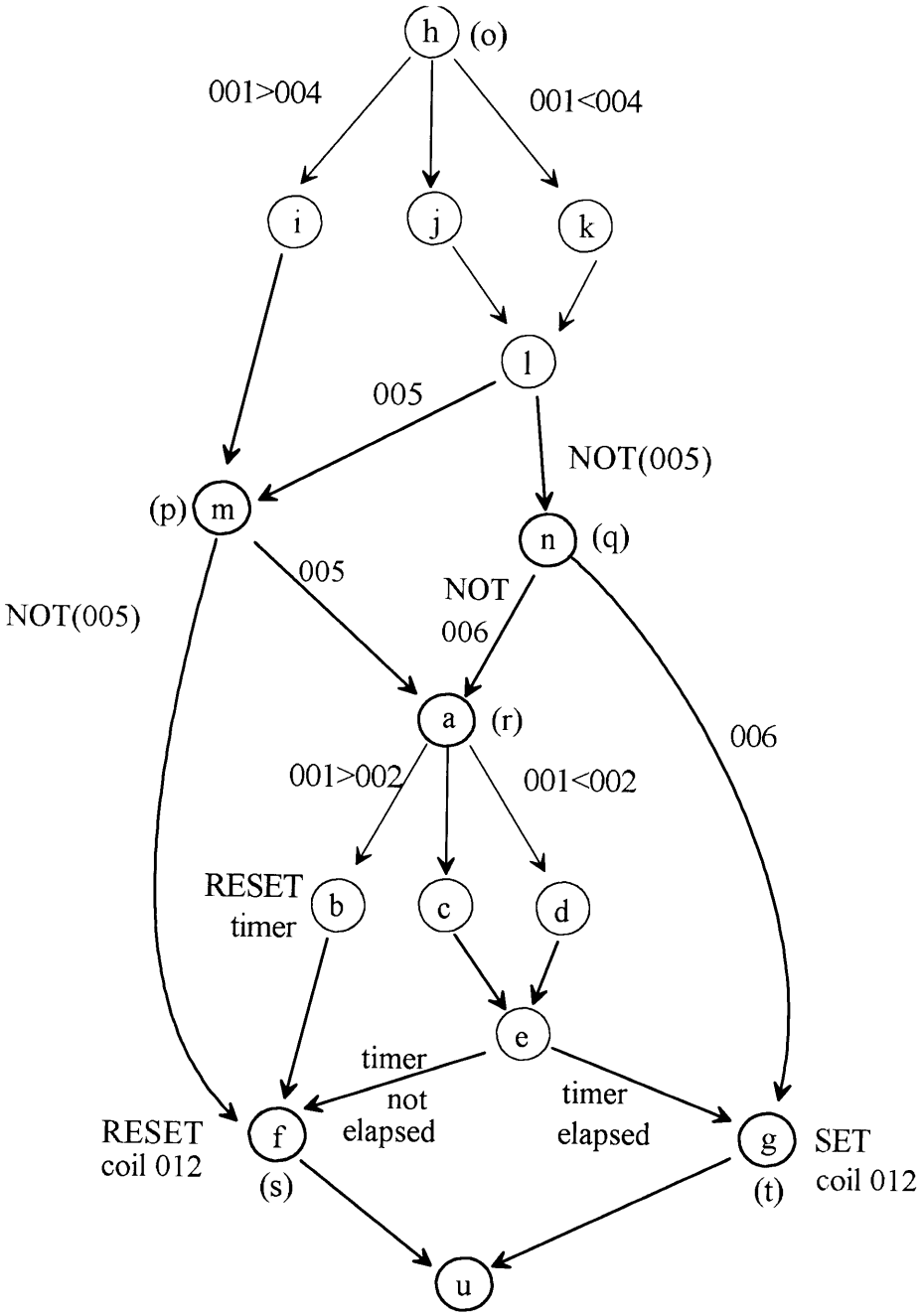
The first case, which corresponds to using the same contact more than once in PLC code, is seen in the flowgraph as repeated nodes. Because the logic of the exiting links of repeated nodes is identical, they are very easy to locate. The second case, which corresponds to contacts evaluating previously established memory locations, is not quite as easy to recognize, however, and therefore requires more extensive searching.

In this example, Figure 4-15 has one instance of the first case (there are two contacts evaluating memory location 005) and two instances of the second case: nodes which evaluate previously established memory values. (Values for memory locations 010 and 011 are established by coils in the first two circuits of the flowgraph -- since the process corresponds to the specific function of a node rather than the logic of the flowgraph, instances of this case are not as easy to isolate; the same memory locations are evaluated by contacts in the third circuit, which does correspond to flowgraph logic.)

To eliminate multiple instances of the same contact can require extensive flowgraph manipulation, or may not even be possible. Ultimately, such manipulations require that the logic to reach all nodes remains unchanged. (Since the flowgraphs are drawn on two-dimensional paper, it might be necessary to cross lines, but that is of no consequence.) This case will be dealt with momentarily.

To eliminate any instances of the second case, the entire set of nodes associated with the logic necessary to execute the node which establishes the value of the desired memory location must be moved to occupy the location of the node associated with the PLC contact. That is to say for this example, node **a** replaces node **r**, where the exit nodes of **r** are replaced by nodes **f** and **g** as the two possible exit conditions (memory location 010 is set or reset). The same is process done for nodes **h, m** and **n** into nodes **o**, **p**, and **q** respectively. The resulting flowgraph is shown in Figure 4-16.

FIGURE 4-16
Compressed flowgraph of Figure 4-15

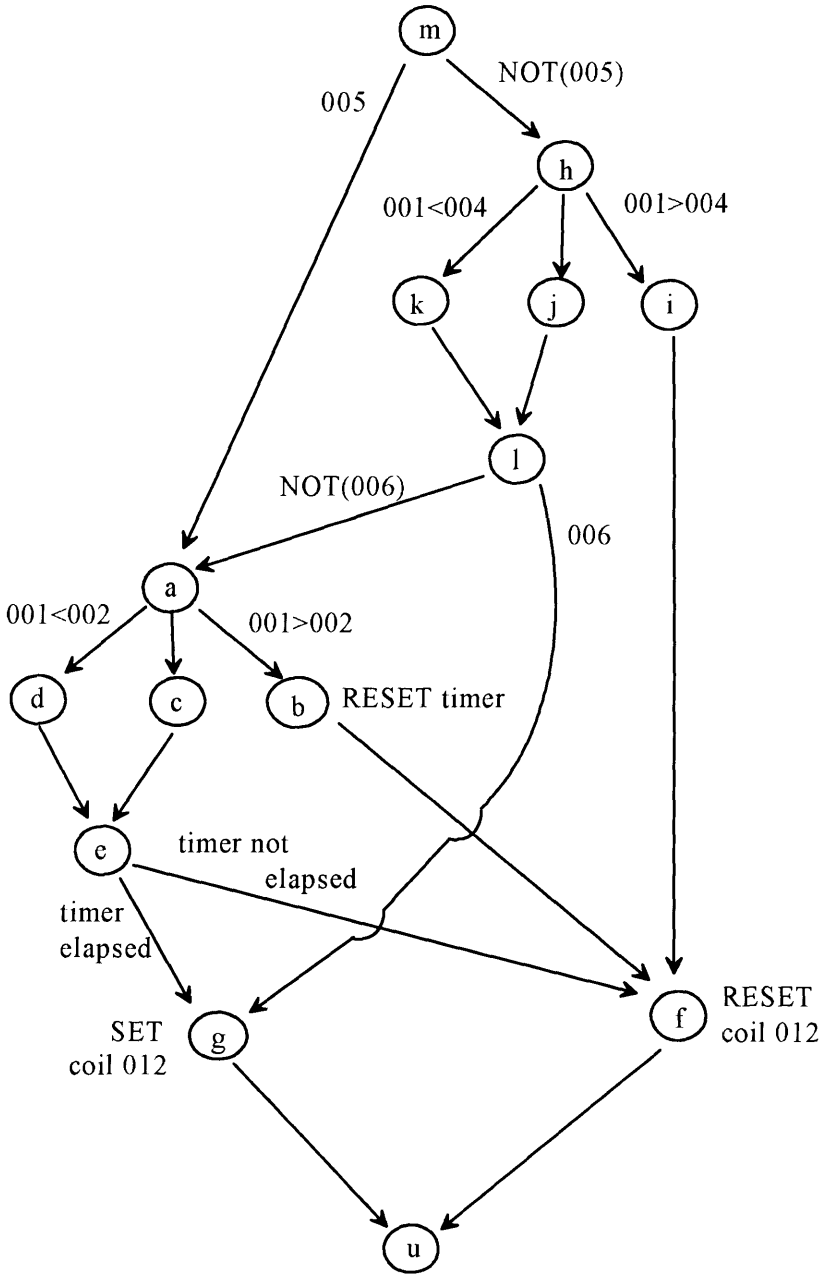


The result of this entire manipulation reduces the flowpath count by two, to a total of nine flowpaths. However, it becomes apparent that there is yet another impossible flowpath -- a result of the one repeated contact which was not dealt with previously. In order to elimiate the impossible flowpath, the flowgraph in Figure 4-15 could be altered slightly to yeild the flowgraph of Figure 4-16: the link that binds nodes **l** and **m** could be moved to bind nodes **l** and **a** since any flowpath taking the **l-m** link could never use the **m-f** link and would invariably traverse node **a**. Note, however, that the memory location 005 is still evaluated twice, a condition that is usually best avoided. Re-arranging Figure 4-16 gives an equivalent "ideal" flowgraph with no impossible paths and no duplicate nodes, as shown in Figure 4-17. This "ideal" arrangement is in McCabe format and further reduces the network by one, resulting in just eight independent flowpaths. Recall that this is compared to eleven flowpaths for the original flowgraph presented in Figure 4-15 and nine flowpaths for the flowgraph presented in Figure 4-16. It is interesting to note, too, that the three original flowgraphs had thirteen flowpaths, which can be explained by the fact that when any two flowgraphs are connected serially, one of the paths is "eliminated" simply because the basepath is common to both (instead of counting both basepaths as two separate flowpaths, they are counted as the same flowpath). Similarly, when three flowgraphs are joined and the common basepath is accounted for, two flowpaths are "eliminated."

The primary difference between the flowgraphs in Figure 4-16 and Figure 4-17 is the location of node **m** which evaluates the state memory location 005, though many of

the node locations have been switched superficially. It is not difficult to verify that the logic for all valid flowpaths has remained unchanged, but it will not be done here.

FIGURE 4-17
An "ideal" flowgraph for the combined logic circuits of Figure 4-11



4.4 Interpretation of Flowgraphs

After finally achieving the "ideal" flowgraph presented in Figure 4-17, having a McCabe metric value equal to eight, it must follow that the actual metric values corresponding to each of its equivalent, though unreduced flowgraphs must also be equal to just eight (not to eleven or nine, as the false McCabe counts would indicate). With this knowledge as a basis, a simpler method of establishing the fundamental number of paths was determined, eliminating the need to reduce flowgraphs altogether.

In order to simplify the method used to determine the actual number of flowpaths through a flowgraph (i.e., the number of tests that need to be run for any given PLC code), one property of flowgraphs proves particularly invaluable: once a variable has been evaluated at any node, future tests of that variable cannot produce a dissimilar result. Thus, while every unique binary decision node (one input and two outputs) increases the overall complexity by one, duplicate nodes do not alter the complexity at all. Therefore, since an "ideal" flowgraph has no duplicate nodes (as such a node is the source of all impossible flowpaths), eliminating duplicate nodes from "non-ideal" flowgraphs will result in the actual number of flowpaths (the actual McCabe complexity value).

It is important to recall that duplicate nodes are not only those that correspond to repeated contacts, but also those nodes which correspond to contacts evaluating memory locations which were previously established using coils.

Using this method of flowgraph simplification can now be used to examine Figure 4-15. Recall that there was one instance of the first case (memory location 005 is evaluated by contacts a total of two times), and two instances of the second case (one contact each which evaluates memory locations associated with coils 010 and 011). The original, false count of eleven flowpaths is therefore reduced by three, resulting in an "idealized" count of eight flowpaths, just as it must have. Note that the impossible flowpaths do not effect the final count and do not need to be dealt with separately.

Tracing even further back into the flowgraph reduction effort, recall that the circuit of Figure 4-11 is represented as a series of three flowgraphs. These three flowgraphs, in their "rawest" forms, are shown in Figures 4-10 and 4-12. Figure 4-10 represents (in form only) both of the upper two circuits and has four flowpaths. Figure 4-12 represents the third circuit having six flowpaths, yielding a sum of fourteen flowpaths. Taking into account the fact that the three circuits are connected serially, thus combining the basepaths, reduces that complexity by two, resulting in twelve flowpaths. Subtracting the number of duplicate nodes yields (accounting for both cases) further reduces the count by four (the two duplicate 005 contacts and one contact each evaluating memory locations established by coils 010 and 011), correctly produces the actual count of just eight flowpaths, even though impossible paths are present.

This revised method of determining the flowpath count can be applied with nothing more than the individual flowpaths for each circuit or network as long as the number of

duplicate nodes can be determined. Since network flowgraphs are relatively easy to create, and duplicate nodes (including those referring to previous coils) are usually easy to identify, determining the number of tests for any given PLC circuit is a relatively simple task.

The following simple formula can be used in determining the number of tests necessary:

$$\text{Number of Required Tests} = F - N + 1 - D, \tag{4-5}$$

where F is the sum of all flowpaths for the individual networks (or circuits), N is the number of networks (or circuits), and D is the number of duplicate nodes.

The only note of caution applies to determining the total number of flowpath for networks with multiple circuits: although the above formula can be employed on the individual networks (where the number of flowpaths in the network is given by one minus the sum of the number of flowpaths for each circuit, minus the number of circuits, minus the number of duplicate nodes), the duplicate nodes which are eliminated for this step should not be counted as duplicate nodes again for the complete network. A simple way to avoid any confusion with regard to this over-counting is to create separate networks for each circuit and use the "circuit networks" to determine the number of tests that will be required.

Finally, the revised flowpath counting method outlined above leads to a direct count of the number of tests necessary directly from the PLC code itself. While it seems intuitive in retrospect, the work leading up to it is necessary to prove its validity. Recall that this process can be applied directly to the PLC code and does not require any flowgraph generation.

The first step in the process is to evaluate each network individually. For each module on the network, a predetermined number of tests is necessary which does not depend in any way on the logic of the network. For example, a contact (other than a repeated contact) adds one to the required test count while a subtraction module (if it is used for a comparison of two numbers) adds two to the test count. A timer also adds one to the test count. In every case, each component (e.g., a contact, a timer, a comparitor) always contributes the same number of tests to the overall test count -- contacts which were introduced in earlier networks are identified as such (on each network as they appear) and those contacts are not counted as additional tests (since they are repeated contacts). The number of tests derived in this way is the number of tests necessary in addition to the basepath test, and the sum of all additional network tests plus one is the number of tests required for a complete test.

In summary, to arrive at the number of tests necessary for a complete PLC code software test, count the number of unique contacts, comparitors, timers and other modules used in the code and multiply the occurrence of each component by the number of

additional tests required for each one. The sum of the result, plus one, is the number of tests required:

$$\text{Number of Required Tests} = 1 + \Sigma_c (F_c N_c), \tag{4-6}$$

where **c** represents each component being used in the PLC code, F represents the "test factor" (number of tests required for that component), and N represents the number of occurrences of that component.

Part 5
Implementation

5.1 General

In order to take advantage of the complete-test counting process developed in this report, the differences between currently employed processes and the proposed process should be discussed. This discussion does not necessarily suggest that the process currently being used is not correct or adequate, but it simply illustrates how the proposed process could be implemented and substantiated.

5.2 Current Process

The currently accepted testing process involves an extensive procedure which attempts to utilize all possible input data combinations and variations. The resulting output data set for each input data set must be verified with respect to the required specifications governing them.

5.3 Comments on the Current Process

For programs which have even a moderate number of discrete input data, the testing of all possible combinations of input data is prohibitively time consuming. For example, for fifty binary data there are 2^{50} (1.23×10^{15}) input combinations. If each test

were executed on a computer requiring an average of 1 millisecond per test, the time required to run the complete test would be nearly 40,000 years! At that rate, in fact, only 3.16×10^{10} tests can be completed in a single year. If tests were conducted all year-round, only about 35 binary data combinations could be tested. In two years, the number of testable binary data would increase only to 36, and 37 input data sets would take nearly 4.5 years to fully test! To put this into another perspective, if analog input data were converted into 8 bit digital signals, then only the interaction of 4 analog input signals and 3 binary signals could be fully tested in one year.

The inspiration for this exhaustive method of input testing comes from the domain of hardware wiring, where each combination of possible logic configurations must be tested due to possible miss-wiring or poor soldering. Such extensive testing is not necessary for software codes, however, since there is no analogy to "miss-wiring" or "poor soldering" in software codes. Testing four individual analog-to-digital converters for proper operation does not take a full year (providing they are not 35-bit converters), and once the hardware has been certified correct, testing them in software applications is trivial. Since each of the data converters functions properly, the software code only needs to be tested for proper operation given that the input data fall into certain ranges. An example is given below to better illustrate this point.

EXAMPLE: Consider that a software program is written to compare an analog input signal (which is converted to an 8 bit digital signal) to two different setpoints, labeled

Setpoint 1 and Setpoint 2. If the input signal is less than Setpoint 1, the code should generate Signal A; if the input signal is greater than (or equal to) Setpoint 1 but is less than Setpoint 2, Signal B should be generated; finally, Signal C should be generated if the input signal is greater than (or equal to) Setpoint 2. Assuming that all of the comparitors within the computer also function properly, testing this program for proper operation requires only five (carefully selected) inputs: one which is less than Setpoint 1, one which is equal to Setpoint 1, one which is between Setpoints 1 and 2, one which is equal to Setpoint 2, and finally, one which is greater than Setpoint 2. Thus, though an 8 bit signal has 256 different variations, there would be only 5 test cases, and if the code executes properly in each case, then the software functions properly.

The example above gives rise to another possible type of error: what would happen if Setpoint 2 were somehow less than Setpoint 1? The exact nature of the output would depend on the structure of the software code, but it is fairly safe to say that the output would not make sense. The solution to this type of error is to be certain that Setpoint 1 is less than Setpoint 2. (If the setpoints are determined using analog hardware, mechanical interlocks could be used. For digital control, logical interlocks could be employed to secure against such errors. For safety-related applications, software interlocks are preferable since they are not subject to mechanical failure and can be tested.)

Another characteristic of software testing is the possibility of independent inputs or outputs. It can be shown that certain inputs are not related to some outputs, and in those cases independent modules can be defined. Each independent module can each be tested separately, with another set of tests confirming the structural relationship between the modules. A general rule is that if each module can be reached as necessary, and the independent modules execute properly, then the entire code will function properly.

5.4 Alternative Process

The alternative process is to examine the code and determine the number of independent modules that need to be tested and the corresponding number of tests, as was described in the previous chapter. With each independent module defined by its own specifications and tests, the sum of the tests would be far fewer than the current process requires, though the coverage of the tests would be just as thorough.

5.5 Comments on the Alternative Process

There are several issues to be dealt with in this discussion: the validity of the tests, the completeness of the tests, the format of the specifications and the completeness of the specifications. Each is addressed separately.

5.5.1 Validity of Tests

In order to demonstrate the validity of the proposed tests, it is important to understand the concept of independent modules and their relationship to one another. McCabe [31, 36] describes the structural relationship between modules as the "essential complexity" of the code, as discussed in Chapter 3. This concept simply establishes the number of tests that are required to ensure that all modules are executed given the correct logical conditions. Within each module, additional tests are executed to ensure proper operation of the module. Given that all independent modules operate correctly internally and that they are invoked properly, the entire code must also perform according specifications. That is not to say that the overall modular structure and the individual modules do not need to be tested as a whole unit -- it just simplifies the process of defining which tests need to be executed.

In this phase of testing, it is important to distinguish between a code's functional modules and PLC code networks. While each PLC network might appear to be its own module (because of the structure of the PLC code), such is not the case, as was shown in the Chapter 4. Each of a code's independent functional modules comprises one or more PLC networks, where each network included in the module either supplies input data to or requires input data from other PLC networks. Thus, the overall effect of the module is such that all of the included PLC networks must operate together to achieve the desired output. As is the case with all software codes, it is possible for two PLC networks (or

even functional modules) to operate on the same inputs and produce entirely independent outputs; such PLC networks would be independent of each other and could therefore be part of independent functional modules.

5.5.2 Completeness of Tests

There might be some concern that eliminating so many tests from the current process might also delete some of the necessary tests accidentally, thus leaving crucial errors undiscovered. Such is not the case, however, since in employing this testing process, all functional paths of the code are tried and tested. It is true, however, that to ensure complete test coverage, the limits of the program (and hardware) must be understood and tested as well. For example, if the registers can overflow (during addition, subtraction, or other routines), those failure modes must be accounted for and eliminated. It is common practice during software testing to exercise limit conditions to ensure that all such error modes have been dealt with, and those tests fall into a separate category (as do hardware tests, for example). While addressing such problems in the software itself would bring them into this realm of software testing, hardware and machine specific tests (such as register overflows) should always be performed.

5.5.3 Validity of Specifications

Since PLC modules follow an extremely simple structure with only a single flowpath, they lend themselves easily to simple specification tasks. For highly reliable

software, specifications could be categorized into two types, both of which would serve to define the structure and operation of the code: 1) task-oriented specifications; and 2) control-oriented specifications.

Task-oriented specifications would be those that pertain to actions needing to be accomplished and would most likely account for a majority of all specifications. This group would be responsible for defining the logic necessary for actuation as well as implementing the actions to be taken. This category of specifications could be broken down into independent modules where applicable.

Control-oriented specifications would define the relationship between task-oriented specifications. For example, it might be necessary to insist that readings from one instrument be taken before taking readings from another due to the nature of the readings (e.g., one reading changes more rapidly than the other). This category would include the overall structural relationship of the task-oriented modules, clarifying those that are dependent or independent.

5.5.4 Completeness of Specifications

A major concern in the nuclear-power industry is the issue of complete specifications. While this is not an issue with other high-reliability components (such as hardware), it is argued that software cannot be fully specified. However, since application

specific integrated circuits (ASICs) are digital hardware and are governed by the same logic that governs PLCs, there is no reason to preclude the possibility of complete specifications. If the computer hardware meets all of the required reliability standards, completely tested software (which is being executed by that acceptable hardware) should not alter the reliability of that hardware. The issue here is to be sure the software is completely tested on the actual hardware which is to be employed. Since all applicable hardware circuits will have been exercised during such a complete software test, future performance of the equipment will be ensured by hardware reliability.

5.6 Additional Comments

5.6.1 Reverse Specifications

As the PLC program code is written and tested, it is possible to reverse the specification process in order to validate the original specifications. To accomplish this, an independent consultant could analyze each module and establish a complete set of independently derived specifications based upon the code itself. Once documented, both sets of specifications could be compared and corrected as necessary. Reverse specifications should reveal extraneous conditions that were introduced into the logic while satisfying the original set of specifications.

Finally, there are some arguments against complete software testing that are based on software/hardware failure modes. For example, if the hardware begins to fail, but does

not fail noticeably (e.g., a memory location will not change values properly), how will the software react? While it is necessary to consider such circumstances, such concerns can also be addressed with redundant hardware and polling techniques, thereby reducing the probability of such an event. There are other similar concerns about the use of software in high-reliability applications, but most should be addressable with hardware (and sometimes software) modifications.

Part 6
Future Work

The reduction and testing procedure developed by this study should be applicable to all varieties of safety-related software systems written using PLC ladder logic. The process of collecting data must include further examples of simple software as well as investigating the applicable limits of complexity or length on the reliability of the procedure. The exact definition of simple software might be defined as that limit which precludes the use of this procedure, whether it be physical length or, more likely, the scope of the specifications.

Finally, work could be done on creating reliable specifications for PLC codes, or alternatively, to develop a method which creates code specifications from the structure of the code itself. Such a method would serve as a means of verification for the specifications.

Part 7
Review and Conclusions

In order to establish a basis for complete software testing of simple, safety-related software, it is necessary to limit the scope of the problem in several ways. By insisting that certain properties apply to the software system, the problem can be limited to concerns about software testing, making it possible to directly address statements generally made that one cannot completely test software. There are three areas in particular that must be addressed and limited, two of which are not directly related to the software testing process but are often cited as obsticals: 1) hardware concerns; 2) concerns about human interactions with the software systems; and 3) software concerns.

First, the hardware requirements for software systems are no different than those for hardware systems, so the methods used to ensure hardware reliability should still be used where applicable (on computer hardware components). This problem is essentially no different than that of using reliability methods employed for current safety-related systems. Hardware redundancy and periodic tests are the primary means of improving reliability, where the required mean time between failures (MTBF) is specified. The additional complexity of equipment being used for software systems should not be a concern since the technology is well developed for hardware reliability in these environments.

Secondly, the human interactions involved in the new safety-related software systems must remain as simple as possible, so that the possible introduction of human errors is minimized. Since even the current safety-related systems, which are being considered for replacement by software systems, must have some limited human interaction, it would be ideal to have similar, though improved interactions with the new software systems.

Finally, a major obstical for complete software testing is the issue of software complexity, which is not a well defined term and means different things to various groups of researchers. For this study, a variation of McCabe's cyclomatic complexity theory is investigated and adapted to PLC codes. It is found that flowgraph theory is particulary well-suited to PLC code testing and offers insights for how to structure a campaign of complete software testing, suggesting that such a goal is attainable.

Research done by McCabe indicates that simple software codes, those with ten or fewer flowpaths, can be written, combined and tested with a high degree of confidence. In fact, if every flowpath of a software code is tested, validated and verified, the code is completely tested. Since independent modules can be written and tested separately, a set of modules can also be completely tested by exercising the entire code. This process guarantees the correct operation of the code, given that the computer hardware functions properly.

In order to avoid unforseen interations, the PLC software processor never executes an item of code other than the safety-related code concerned. Also, there are no unexpected runtime conditions since all conditions have been fully exercised. It is important to keep in mind the four assumptions used in determining the number of tests necessary using the test counting method developed by in this study. All four points apply to a single pass of the PLC code:

1. each network is executed serially and cannot be skipped;
2. every element in each network being executed is evaluated;
3. all circuits within a network can all be written as separate networks; and
4. once established, external memory values associated with flowpaths cannot be altered by external equipment.

The objective of this initial study is to provide evidence that software reliability can equal or surpass the reliability of similar analog systems. To do this, it is necessary to keep as many features common to both systems as possible. By maintaing the reliability of the present hardware systems and by using comparable human interactions, the only remaining variable which can affect reliability is found in the application and implementation of the software code versus the development of analog equipment. The differences between the development processes of the two systems is therefore the crucial element. As this study shows, completely testing simple, safety-related PLC software is feasible and offers great potential to all industries requiring high-reliability of safety-related systems.

References

1. M. W. Golay and D. D. Lanning, "Methods for Development and Demonstration of High Reliability Nuclear Safety-Related Software," proposal to M. Novak of Combustion Engineering in Windsor, CT, Revision 1, MIT, Cambridge, MA, 22 January 1993.

2. M. Novak, "Items related to Graduate Research Project on Software Reliability," letter to Professor M. Golay of MIT, ABB-C/E, Windsor, CT, 13 April 1993.

3. R. Bell and S.S.J. Robertson, "Programmable Controllers in Safety-Related Systems: HSE Guidelines on Programmable Electronic Systems," HMSO, London, 1992, pp. 41-46.

4. T. Daughtrey and J. Scecina, "Engineering of Software for Acceptability," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 51-54.

5. ABB-Combustion Engineering, Inc., "Software Design Specification for Diverse Protection System for Yonggwang Nuclear Power Plant Units 3 and 4," Revision 01, Status 1, ABB-C/E, Windsor, CT, 1992.

6. ABB-Combustion Engineering, Inc., Sample PLC code, provided by ABB-C/E, Windsor, CT, 1993.

7. M. E. Hardy, "Reliability and Availability within Programmable Control Systems," HMSO, London, 1992, pp. 47-49.

8. M. M. Mano, Computer Engineering Hardware Design, Prentice Hall, Englewood, Cliffs, NJ, 1988.

9. M. Hecht and J. Agron, "A Distributed Fault Tolerant Architecture for I&C Applications," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 55-62.

10. J. K. Munro, Jr., "Considerations for Control System Software Verification and Validation Specific to Implementations Using Distributed Processor Architectures," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 496-501.

11. R. E. Battle and G.T. Alley, "Issues of Verification and Validation of Application-Specific Integrated Circuits in Reactor Trip Systems," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 487-491.

12. J. M. O'Hara, "The Effects of Advanced Technology Systems on Human Performance and Reliability," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 253-259.

13. D. D. Lanning, "Human Factors in Design and Operation: TMI-2 and Chornobyl Case Study," MIT, Cambidge, MA, 1993.

14. D. D. Lanning, Course Notes for 22.32, Nuclear Power Reactors, MIT, Spring 1993.

15. J. P. Jenkins, "Human Error in Automated Systems: Lessons Learned from NASA," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 269-272.

16. W. R. Nelson, et al., "Lessons Learned from Pilot Errors Using Automated Systems in Advanced Technology Aircraft," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 189-194.

17. ABB-Combustion Engineering, Inc., "Diverse Protection System Functional Test Procedure for Yonggwang Nuclear Power Plant Units 3 and 4," Procedure 10287-IC-TP620-1, Revision 01, Status 1, ABB-C/E, Windsor, CT, 1992.

18. ABB-Combustion Engineering, Inc., "Software Quality Assurance Plan for Diverse Protection System for Yonggwang Nuclear Power Plant Units 3 and 4," Revision 00, Status 1, ABB-C/E, Windsor, CT, 1992.

19. S. J. Andriole, ed., Software Validation, Verification, Testing and Documentation, Petrocelli Books, Princeton, New Jersey, 1986.

20. K. R. Apt and E. R. Olderog, Verification of Sequential and Concurrent Programs, Springer-Verlag, New York, 1991.

21. P. Shewmon, "Letter to the Chairman of the U.S. Nuclear Regulatory Commission," 18 March 1993, released 29 March 1993.

22. C. Jones, chairman Software Productivity Research, Inc, "Software Quality: What Works and What Doesn't?," Burlington, MA, November 1993.

23. S. Clatworthy and P. Hickford, "The Attainment of Optimum Systems Reliability -- a Structured Approach," HMSO, London, 1992, pp.51-52.

24. O. J. Dahl, Verifiable Programming, Prentice Hall, London, 1992.

25. B. K. Daniels, Safety and Reliability of Programmable Electronic Systems, Elsevier Applied Science Publishers, London, 1986.

26. E. B. Eichelberger, et al., Structured Logic Testing, Prentice Hall, Englewood Cliffs, NJ, 1991.

27. A. Ghosh, et al., Sequential Logic Testing and Verification, Kluwer Academic Publishers, Boston, 1992.

28. P. Gray, ed., "Europe's Conference on Programmable Controllers," Conference Proceedings, Metropole Hotel, NEC, Birmingham, 11th-12th November 1987.

29. M. Treseler, Designing State Machine Controllers Using Programmable Logic, Prentice Hall, Englewood Cliffs, NJ, 1992.

30. Musa, Iannino, and Okumota, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, New York, 1987.

31. H. Zuse, Software Complexity: Measures and Methods, Walter de Gruyter, Berlin, 1991.

32. B. Littlewood, ed., Software Reliability, Blackwell Scientific Publications, Oxford, 1987.

33. J. R. Matras, "Criteria for Computer Systems Used in the Design of Nuclear Generation Plant Safety Systems," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 397-404.

34. B. Beizer, Software Testing Techniques, 2nd ed., Van Nostrand Reinhold, NY, 1990.

35. M. H. Halstead, Elements of Software Science, Elsevier, North-Holand, 1977.

36. T. J. McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," National Bureau of Standards special publication 500-99, December 1982.

37. W. B. Reuland and R.T. Fink, "Digital I&C Upgrade Licensing Guidelines," pp. 410-414.

38. S. Koch and K. Andolina, "Qualifying Software for Class 1E Equipment," Nuclear Plant Journal, January-February 1993, pp. 66-69.

39. L. C. Oakes, "Transition from Analog to Digital Technology: Current State of the Art in Europe," WTEC Panel Report on European Nuclear Instrumentation and Controls, December 1991, pp. 27-43.

40. P. Rook, ed., Software Reliability Handbook, Elsevier Applied Science, London, 1990.

41. S. C. Bhatt and J. A. Naser, "Software Verification and Validation for Instrumentation and Control Systems in Nuclear Power Plants," Proceedings of the Topical Meeting on

Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 492-495.

42. J. M. Gallagher, "The Role of International Standards in the Design of Modern I&C Systems for Nuclear Power Plants," Proceedings of the Topical Meeting on Nuclear Plant Instrumentatoin, Control and Man-Machine Interface Technologies, 18-21 April 1993, pp. 361-365.

43. Modicon, Inc., Modicon Modsoft Programmer User Manual, GM-MSFT-001 Rev B, North Andover, MA, September 1991.

44. Modicon, Inc., Modicon 984 Programmable Controller Systems Manual, GM-0984-SYS Rev. B, North Andover, MA, May 1991.