Analysis of safety critical plc code against IEC 1508 development techniques

Williamson, Louise M.

Analysis of Safety Critical PLC Code Against IEC 1508 Development Techniques

Louise M. Williamson

MSc Thesis

Centre for Software Maintenance

Department of Computer Science

University of Durham

September 1998

ABSTRACT

The aim of this thesis is to assess the applicability of recommended software development techniques defined in IEC 1508 [8] to PLC (Programmable Logic Controller) code developed for offshore oil platforms. The draft standard IEC 1508 contains specific recommendations which have the objective of improving the safety characteristics of safety critical code. The recommended techniques could have one of the following characteristics with regard to offshore PLC code:-

- They are already used in the development of code.
- They could be used in the development of the code.
- They could not be used due to the application domain.
- They could not be used due to the specific programming environment analysed.

It was the aim of the thesis to characterise a subset of the IEC 1508 techniques into the above categories. The analysis was requested by the Health and Safety Executive (HSE) Offshore Division.

The analysis has been performed using two major case studies, taken from live industrial safety-critical systems operating on a North Sea Oil Platform; they both comprise 300K lines of code in total. Both systems were written in three high level PLC languages. It was decided to translate the code into one language, so the analysis was undertaken in terms of a single language. A translator has been written, and a number of static analysis tools, therefore allowing all the code to be analysed.

The key twenty two recommendations from IEC 1508 have been selected, and the case study systems correspondingly analysed, using a modified Goal Question Metric (GQM) approach as a unified framework.

The overall analysis method has been found to be successful in supporting the detailed analysis of IEC 1508 recommendations. The thesis presents detailed conclusions on each analysed technique, as well as more general observations on the PLC code.

ACKNOWLEDGEMENTS

This thesis is based on the work that I did as part of ROPES (Review of Offshore Programmable Electrical Systems) at the University of Durham, funded by the Health and Safety Executive (HSE) Offshore Division. The code was supplied by Kvaerner plc through Stirling Taylor.

This work could not have been completed without input from a number of people, to whom I am very grateful. Guidance, advice and comments came from Prof. Keith Bennett my supervisor and Norman Turner the HSE co-ordinator.

The discussions and help from members of the department were invaluable, especially the 'lodgers' and 'lofters'. A special thanks must go to Liz for proof reading the thesis, Mark for the slicing tool, James and Antony for being there when needed and all the help they gave me.

The thesis would never have been written without all the encouragement, nagging and support from my family and friends.

DISCLAIMER

This thesis has not been submitted to any university for a degree.

COPYRIGHT

CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

Computers are an integral part of today's society. Software is inherently complex and must perform to very high standards if it is to operate correctly. However there are many reports of failures of industrial scale applications in the computer literature. [1] The most serious of these failures are perhaps those that occur in life threatening, safety critical, systems which are the concern of this thesis. The size and diversity of problems that software solves ensures that failure will be an ever increasing problem. "Many solutions .... have been proposed which tackle a wide variety of issues such as: management of software projects; better software languages and tools; and methods for mapping high level descriptions of systems into executable code"[2]. This thesis spans development solutions and translation technology for safety critical systems using Programmable Logic Controllers.

The three main software engineering topics discussed in this thesis are:-

1. Safety Critical systems

   "A system is **safety critical** if failure of the system would result in loss of human life, personal injury or significant material loss" [3]. It is normally accepted from a software engineering point of view that the software is safety critical if failure of the software would result in the loss of human life.

2. PLCs

   "A **Programmable Logic Controller**(PLC) is an electronic device that controls machines and processes. It uses a programmable memory to store instructions and execute specific functions that include On/Off control, timing, counting, sequencing, arithmetic and data handling"[4]. A PLC "is in essence a device that is specifically designed to receive input signals and emit output signals according to the program logic"[5]. PLCs were developed as basic computers that could replace relay circuits.

As such they were developed so they could be programmed in a similar way to the design of a relay circuit. "It was possible to use them to take over all of the logic functions from relays and replace hundreds of relays with a more compact, solid-state unit." [5]

3. Translation Technology

A **translator** is "a program which converts statements written in one language to the format of another programming language"[6].

1.1 PROJECT AIMS

PLC systems are widely used in safety-critical and safety-related applications. Hardware reliability can be predicted by recognised techniques[7]. With respect to software reliability the situation has been less clear. The operating systems are given extensive onsite testing due to their use in many sites. The application software is normally developed for only one application so has had much less testing. In recognition of this, Standard IEC 1508[8] has defined the concept of **Safety Integrity Levels** (SILs), for PES-based systems (Programmable Electronic Systems). The SILs are organised as a series of levels of increasing rigour. Discrimination between levels is expressed in terms of the average time expected between failures (i.e. the system does not perform within its defined specification). SIL 1 is the least rigorous level while SIL 4 is the level of highest rigour.

For each SIL, recommendations are made in IEC 1508 for *highly recommended*, *recommended* and *not recommended* techniques which can be employed to achieve the SIL. These techniques include specification methods, design methods, programming techniques, languages and quality assurance techniques. Hence the system designer will decide the SIL, based on an analysis of the application and its domain, and then IEC1508 should list a selection of appropriate development techniques along with techniques that are deprecated. The concern of this thesis is entirely with the developed software and not the specification or hardware decisions.

A selection of techniques associated with SILs 1,2 or 3 were then analysed with respect to two different systems. The techniques were chosen on their relevance to the data available. The analysis was performed to identify if the technique:-

- had been used
- could have been used
- could not be used due to the application or the programming environment

The code that was analysed for this thesis is the Factory Acceptance Testing (FAT) source code for the safety systems that have been operational on a North Sea Platform for over two years. The code was the ESD (Emergency Shut Down) code and F&G (Fire and Gas) code on the platform. It was written using Siemens APT (Application Productivity Tool, version 1.6) which is designed to run on a PLC on the platform. The F&G and the ESD code were run on different PLCs. The code was developed using the three languages supported by the Siemens APT. It was chosen as it is thought to be representative of ESD and F&G systems on other oil platforms.

The project was divided into two separate parts - translation and then analysis of the code.

1.1.1 Translation

The code was translated from three languages to one language. The translation was beneficial for two reasons; the translation provided much useful data and the resulting code was in one language for the analysis. The PLC languages that were translated were:-

1. Sequential function charts (SFC)
2. Continuous function charts (CFC)
3. Math language

Three languages posed problems for analysis of and reasoning about PLC programs, because tools and methods would need to be developed for all three representations. It was thus decided to use a common single representation to ease analysis. The choice made was to use WSL, a Wide Spectrum Language with formal semantics, designed and used at Durham [9]. This had several potential advantages:-

1. The representation would allow a single 'language of discourse' i.e. one representation for all subsequent analysis based on IEC1508 recommendations
2. Tools need only be built for the one language -WSL - not three languages, and there were tools already in existence for manipulating WSL.
3. Defining a mapping document from the informally defined PLC languages to the formally defined WSL would highlight any language problems e.g. omissions, contradictions, ambiguity etc.
4. The suitability of WSL for representing PLC code could be assessed.

WSL is a Wide Spectrum Language (see section 3.2 for more detail), which is a high level language. It supports externally defined functions and procedures without needing them to be included in the code. None of the variables have a specific type nor do they have to be pre-declared. The fact that declaration of variables is not required enables blocks of code from within a program to be manipulated by MA (Maintainers Assistant). MA is a tool produced locally at Durham for aiding restructuring of code written in WSL.

WSL supports a construct known as an *action system* which takes code that contains GOTOs and makes each GOTO jump a block of its own. In reverse engineering it is necessary to deal with code as it is and not in ideal format. So representing GOTOs in a form for restructuring is mandatory. The *action system* construct avoids the problems of continuation semantics which are not suitable for transformation based analysis tools.

The two pictorial languages (SFCs and CFCs) were translated into WSL to provide the structure and the framework of the program. The SFC was represented as an action system, while the CFC was represented as function calls from each unit to CFB (Continuous Function Block) procedures. CFBs are similar to procedures and are located in the CFCs. When the PLC is compiled by the Siemens APT system the ordering of each of the CFBs is automatically generated. Unlike most text based languages the programmer cannot specify the ordering of CFBs or CFCs. Once compiled, the code runs in that order until recompiled. The translated code was only generated once and not in all the possible different orderings. The CFBs, CFCs and SFCs were maintained as procedures. The code internally within the CFBs and SFCs retained their internal ordering.

The translated ESD and F&G code was too large to be handled by the tool MA so it was necessary to partition the code. The code was sliced procedurally (see 1.1.2.1) along an output variable. A record of the dependent variables was stored so the output could be deemed dependent on the set of input variables. A list of the procedures relating to a CFB was maintained and then these were reassembled to form the sliced code.

1.1.2 Analysis

The second phase of the project was the analysis of the code.

IEC1508 defines a set of highly recommended techniques for each SIL level. The analysis was to assess the extent to which the highly recommended techniques (for SIL 1,2 or 3) were effective, considering in particular the ESD and F&G PLC software from the offshore platform. This involved assessing whether the technique could have been used, was used, or why it was not used. It was also beneficial to see if the technique could theoretically affect the safety of the system. Only techniques recommended for SIL 1,2 or 3 were analysed as SIL 4 was regarded by the HSE as inappropriate for software systems on an offshore platform. The code was chosen because it was felt by the HSE that the two systems were representative of other ESD and F&G PLC systems on Offshore Platforms.

It was not feasible to undertake dynamic analysis of the software. So the problem was expressed in terms of the static analysis of the PLC code. It was decided that the set of IEC1508 analysed techniques should be analysed using a unified framework to answer the following questions:-

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any of the safety features?

During the analysis process, the code, in both its original format, and WSL was used. During the translation process valuable information was obtained including names and types of variables, number of CFCs, number of CFBs and lengths of blocks of code etc. The analysis was structured so that the above two questions could be answered for each technique. Various further questions about the code were used to answer the two main questions. The final conclusions made were regarding the viability of the technique and whether it would aid the development of safety critical ESD and F&G PLC systems on Offshore Platforms.

To perform the analysis on the code, data collection techniques were used. These included:-

1.1.2.1 Program slicing

Program slicing was used to slice the code along an output variable. All the inputs that were dependent on an output were recorded and all of the procedures that were relevant to the output were put into the sliced code. [10] The number of inputs on which an output was dependent could then be studied. A slicing tool was developed to assist the analysis process.

1.1.2.2 Graph tool analysis

Graph tool analysis was used so that call graphs of the code could be drawn. Call graphs were also drawn of the SFCs to demonstrate the structure and to demonstrate that the translation agreed with the SFC description.

1.1.2.3 Transformations

A transformation is the re-ordering of code so that the meaning and overall outcome of the code remain the same but the syntax or form is different. This is beneficial if it enables easier understanding of the meaning of the code. Subsets of the code were transformed using MA to determine how the code could have been restructured and to determine if it made the code more understandable. In the case of nested conditional statements experiments were undertaken as to how they could be transformed.

1.1.2.4 Frequency of variable usage

One in ten global variables were studied to see in how many units the variables had been read, written and used. The programs were divided into units which are equivalent to modules.

1.1.2.5 Variable usage

All the variables were analysed (automatically) to determine which ones had been read but not written, written but not read, and not used. This was done using the information provided by the intermediate representation used within the slicing tool.

1.2 CRITERIA FOR SUCCESS

The top priorities of the thesis were:-

1. To identify key highly recommended techniques from SIL 1,2 or 3 that can be analysed using the data available.
2. To analyse the code to assess the feasibility of using the technique with the specific safety critical PLC code.
3. To identify the general characteristics of the ESD and F&G PLC code on an offshore platform.

The secondary priorities of the thesis were to determine:-

1. If a single language could be used to replace the three PLC languages.
2. If any language deficiencies were identified in the PLC languages.
3. If it is helpful to perform analysis in this way and what the benefits and problems were.

1.3 THESIS STRUCTURE

Chapter 2 introduces the three main subjects areas around which the project was developed. These were:-

2.1 Safety critical systems

2.2 PLCs

2.3 Translators

Chapter 3 discusses phase one of the project. An introduction to the source and target languages is given. The chapter gives an overview of how the source language is mapped onto the target language. The final part of the chapter describes the result of building the translator.

Chapter 4 summarises the characteristics of the code that was analysed. This includes the numbers, types and usage of variables. The number of units (modules) and the number of functions in each unit. The length of the code, the types of statements used and the level of nesting of the conditional statements. Control flow diagrams of the code can also be found in this chapter.

Chapter 5 discusses phase two of the project; it details all the analysis that was performed on the code against IEC 1508 (section 2.1.4 gives an introduction to IEC 1508). The safety critical techniques are discussed separately within the chapter. Each technique is defined, the analysis process described and then the conclusions were discussed.

Chapter 6 compares the achievements of the project against the criteria for success from section 1.2.

2. GENERAL BACKGROUND

The three software engineering areas that are used during this thesis are now discussed.

2.1 Safety Critical systems

2.2 PLCs

2.3 Translation Technology

2.1 SAFETY CRITICAL SYSTEMS

"The use of software in safety critical applications has grown rapidly in the last decade and continues to increase. Prominent applications such as railway signalling, nuclear power stations, chemical plants and fly by wire aircraft"[11] and the ESD or F&G system on a North Sea Oil platform, are all high prominence examples of safety critical systems. "A system is **safety critical** if failure of the system would result in loss of human life personal injury or significant material loss." [3]

A more general definition of a "safety critical system is one that has at least one safety critical service"[12]. A "**service** is judged to be safety critical in a given context if its behaviour could be sufficient to cause the control equipment to inflict or prevent the equipment from inflicting, absolute harm on resources for which the organisation operating the system has responsibility"[12].

A safety critical system does not necessarily involve computers, but the safety critical systems discussed here required computer software. Many safety critical systems were in use before the wide spread availability of computers. The PLC (section 2.2) replaced relays and hardwired circuits. A computer system though can only be safety critical if it reacts with the outside world. "**Safety critical software** is any software that can directly or indirectly contribute to the occurrence of a hazardous system state" [13]. Where "safety critical systems must strive to maintain safe behaviour even in the presence of failures of system components and when the behaviour of the environment deviates from that expected"[14]. "Software on its own cannot cause harm - only when it is embedded in a system and put into use can it be hazardous" [15].

An **accident** is "an incident with detrimental consequences (due to insufficient control of one or more hazards)"[16]. An **incident** is "a significant occurrence or event with potential detrimental consequences"[16]. A **hazard** can be defined as "a source of energy, or combination of factors that can lead to an accident if inadequately controlled"[16].

An accident can also be defined as "an undesired and unplanned (but not necessarily unexpected) event that results in (at least) a specified level of loss"[17]. An accident should be avoided if at all possible, an incident though is acceptable but not desirable, where an "incident is an event that involves no loss (or only minor loss) but with the potential for loss under different circumstances"[17].

There are two different types of safety critical systems. The first is **primary safety critical software**, which is "software embedded in a hardware system used to control or monitor some other process. Malfunctioning of such software can result directly in human injury or environmental damage"[17]. The ESD and F&G systems studied are both primary safety critical software. **Secondary safety critical software** is very difficult to identify and is "software which can indirectly result in injury"[17]. A design tool used in a safety critical system's design or a database storing records important to a safety critical system, are both examples of secondary safety critical software.

A system provides safety for the users and those around only if there is "freedom from accidents or losses" [13]. Often when manufacturing is involved extra safety aspects are thought to be too difficult or too expensive to implement until they are insisted upon by standards or laws. A prime example is that children used to become stuck in fridges because they could not be opened from the inside but manufacturers insisted it was not possible and too expensive to implement it otherwise. A law was passed that fridges had to be able to be opened from the inside so magnets were used, which was cheaper than the previously used latch.

There are many standards in existence that cover all forms of safety critical system development. They include:-

- **IEC 1508** Functional safety: safety related systems[8]
- **UK MoD 00-56** Safety management requirements for defence systems containing programmable electronics [18]
- **MIL-STAN 882C** System safety program [19]
- **MISRA** Motor Industry Software Reliability Association Report 2: Integrity[20]
- **STANAG** Safety design requirements and guidelines for munition related safety critical computing systems [21]

[22] (IEC 1508 will be discussed in more detail in 2.1.4)

Standards are in existence to ensure that those who would be affected by a system are safe. Without standards it is easier to justify cost cutting exercises that are detrimental to the system safety.

When developing safety critical software it is important to remember that the entire system has to be built with safety in mind during the whole life cycle. Safety has to be given a priority by the entire development team. "One way management can demonstrate true commitment to safety goals is through assignment of resources" [13]. This commitment has to be demonstrated during the life of the system as well as during its development. Lord Cullen's report into the Piper Alpha disaster observed "The safety policy and procedures were in place: the practice was deficient" [23].

"**System safety** deals with systems as a whole rather than with subsystems or components"[13]. It is easy as a software engineer to think only of software but "software does not harm directly; but only as part of an overall system; it is important to assess the software contribution to and responsibility for; overall system safety" [22].

At all times during the development of the system, safety should be considered. In the waterfall life cycle model (Figure 2:1) it would be from the specification phase to the maintenance phase.

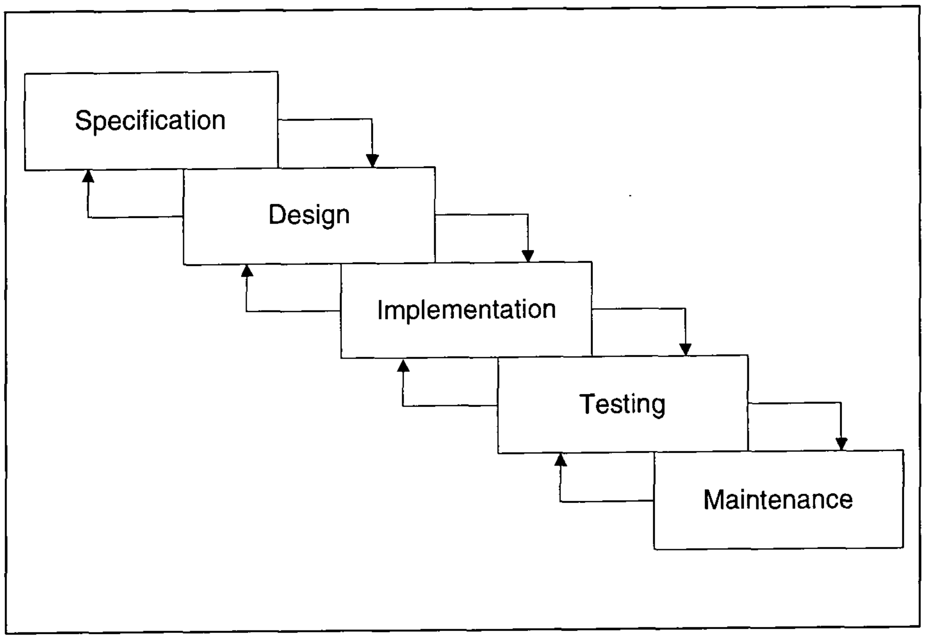


Figure 2:1
Waterfall Model

It is much easier and more reliable to plan safety originally than to tag it on at the end. "Early detection of errors significantly reduces the cost of the production process" [14]. This implies that "Requirements analysis plays a vital role in the development of safety critical systems since any faults in the requirements specification will corrupt the subsequent stages of system development"[24]. The "early phases in the development life-cycle, such as requirements and specification, are extremely relevant for dependability. It is crucial to identify hazards early in the design process, then to take appropriate design measures to eliminate or control these hazards"[14]. Many errors that percolate through to the final product can be traced back to an incorrect specification.

2.1.1 Characteristics of An Ideal Safety Critical System

A safety critical system must be safe, should be available, reliable, dependable and run in real time. The system has to be tolerant of hardware, software and human errors or faults. The software should be free of errors. The system should be true to the specification and the specification should be correct. (All terms are described below.)

Safety is the state of a system that cannot cause any harm. In ideal situations all safety critical systems should always be safe. The system would be **intrinsically safe** "when there is no possibility of it causing or failing to prevent absolute harm"[12]. But in a safety critical system the possibility of harm will be there so engineers strive for **engineered safety** which is "when a system has been designed to minimise risk or to reduce it to an acceptable level"[12]. A safe system is one that will produce the correct output or that an incorrect output will be detected[25]. A safe system has an acceptable amount of risk where **risk** is a function that identifies the chances of a hazard occurring, and what the probability is of the hazard leading to an accident. This is known as the ALARP (As Low As Reasonably Possible) principle - where the cost of reducing the risk is too great in comparison to the amount of risk reduced.

Reliability is a measure of the delivery of a proper service even if parts of the system are failing, or there should be no delivery of the service at all. Output of the system is correct, and the output that is being delivered at a time of failure is as specified. [26], [25]. A system cannot be 100% reliable, so it is up to the designer to determine what an acceptable level of reliability is and try and measure it. Reliability is also defined as "the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions"[13].

Availability is defined as a measure of the service being delivered irrespective of whether it is giving a correct or incorrect output.[26] Safety is not possible without reliability and availability, as a safe system is required to be correct and always functioning.

Dependability is the property that allows the system to be relied upon to provide a continuous, reliable and safe system that will be available for an acceptable amount of the time.[26]

Correctness is defined as the system behaves exactly as it is specified to behave. This implies that a correct system will not necessarily be dependable or safe. It will only be dependable if the specification is correct and error free.[26]

Many safety critical systems are required to run in **real time**. "A real time computer system may be defined as one that controls an environment by receiving data, processing it and returning the results sufficiently quickly to affect the functioning of the environment at that time"[27]. The real time factor is crucial as safety critical environments often use computer systems to enable high speed reaction times. An aeroplane system that took even an hour to detect a faulty engine would be worthless.

The software should be error free where an **error** is the divergence of the state of the system from that expected or required of it, (adds two numbers rather than subtracts, or in a case statement does not consider an important case etc.). In theory it is possible to make software error free, with respect to a specification. This is very difficult; so failing fault free software, the system should be fault tolerant, of both hardware and software faults. A **fault tolerant** system is one that is resistant to faults in the system and can continue producing correct output. A **fault** is "a defect in the system which may, under certain operational conditions, contribute to a failure"[16]. A fault can lead to an error but the system should be designed in such a way that the error can be prevented or removed. Being fault tolerant enables the system to provide a dependable service even if there are faults in the hardware or software.

A safety critical system is one that could cause irrecoverable harm to persons or property. The aims of safety critical software is for it to be dependable, run in real time, be error free or at least fault tolerant. The system should also deliver a continuous service that is free from failures[26].

2.1.2 Accidents - Why Do They Happen

Accidents are often dependent on the safety culture within an industry, or an organisation. The safety culture "is the general attitude and approach to safety reflected by those who participate in that industry: management, workers and government regulators. Major accidents often stem from flaws in this culture especially (1) over confidence and complacency, (2) disregard or low priority for safety, or (3) flawed resolution or conflicting goals."[13]

Although accidents are often preceded by warning signs that are ignored they do tend to fall into one of the following categories:-

1. **Human error** - humans make mistakes either **omissions** where something is not done, i.e. testing was not carried out on all the data. Or **commissions** where a human does something but wrongly. In the Bhopal accident "employers were apathetic about routine mishaps and about the value of emergency drills"[13].

2. **Unpredicted combinations of events**- accidents often happen when events occur in an unexpected sequence of events, or there are multiple faults. All the faults individually may have been catered for but when they combine it is more difficult to prepare for them.

3. **Worn out components** - if the hardware fails then there is likely to be an accident. The hardware in safety critical systems should be regularly checked for faults.

4. **Poor or incorrect design**- if the design is wrong then even if the system is correct with respect to the design accidents will still happen. Examples of poor design can be in user interfaces where not enough information is provided to the operator. Alternatively where a designer has made incorrect assumptions e.g. in a nuclear power station the dial relating to a control switch was out of sight at the switch.

5. **Complexity**- as systems get more complex humans can no longer understand them and as understanding decreases, so the chances of faults entering the system increases.

"A common thread in most accidents involving complacency is the belief that a system must be safe because it has operated without accident for many years"[13]. "There is an awful sameness about these incidents, they are nearly always characterised by lack of forethought and lack of analysis and nearly always the problem comes down to poor management"[23], was stated by Tony Barrell an expert in offshore safety.

2.1.3 Why Computers?

"Hardware backups, interlocks and other security devices are currently being replaced by software in many different types of systems, including commercial aircraft, nuclear power plant and weapon systems. Where hardware interlocks are still used, they are often controlled by software"[13]. Some of the reasons for replacing hardware systems and human controllers with software systems are that "software does not exhibit random wear out failures as does hardware"[13]. The computer can control and read a device with greater frequency and accuracy than a human. It does not make mistakes due to tiredness. Computer control systems can perform calculations faster than their human counterparts, and speed can be very important in safety critical systems.

The software replacement is not all beneficial because software is not always as reliable as the mechanical parts it is replacing. We make better software today by using tools but the size and complexity is also increased.[28] "Many basic mechanical safety devices invented long ago are tested, cheap, reliable and failsafe, and they are based on simple principles of physics"[13]. This is unlike the craft of building software which is based on trial and error[28].

2.1.4 IEC 1508

IEC 1508 is a draft standard that is aimed at improving the safety of systems built in conjunction with Programmable Electrical Systems (PES). The standard identifies many ways of creating safe code. One important feature identified in IEC 1508 is its use of SILs (safety integrity level). "**Safety Integrity** (SI): The probability of a safety related system satisfactorily performing the required safety functions under all the standard conditions within a stated period of time. .... The higher the level of safety integrity of the safety related systems the lower the probability that the system will fail to carry out the required safety functions." [8]

"**Safety Integrity Level** (SIL): One of four possible discrete levels for specifying the safety integrity requirements of the safety functions to be allocated to the safety related systems. Safety integrity level 4 has to be the highest level of safety integrity; safety integrity level 1 has the lowest" [8]. It has been suggested that no computer system should be expected to be SIL 4 and most should not be required to be SIL 3. If a system needs to be of SIL 4 (and sometimes SIL 3) then there should be a hardwired system around the software system so the entire system is not totally dependent on software.

IEC 1508 provides a number of definitions including:-

"**Fault**: The cause of an error is a fault (e.g. hardware defect, software defect) which resides, temporarily or permanently in the system.

Error: An error is that part of the system state which is liable to lead to failure. A failure occurs because the system is erroneous.

Failure: A system failure occurs when the delivered service deviates from the intended service. A failure is the effect of an error on the intended service" [8].

IEC 1508 also suggests many techniques that are *recommended, highly recommended* and *not recommended* so the SIL can be reached. This is done by listing all the techniques and stating how recommended they are for each level. A complaint about IEC 1508 is that there is "little or no guidance on how to assess whether the desired levels of integrity have actually been achieved" [22].

"Safety critical systems (scs) normally need to be certified for use, and this certification is usually done on the basis of a safety case. A safety case presents a reasoned argument that a system meets its safety requirements and will be safe for use"[29]. A **safety case** is something that is written to prove for an authorising body that the system is safe to be put into operation. "The purpose of a safety case is to present the argument that a system, be it physical or procedural, is acceptably safe to operate. .... Safety cases will ultimately be specific to a particular system"[30] Kelly also suggested that patterns emerge about what should be put into a safety case. There will always be specific evidence that has to be included and known when developing a safety case; the ordering and layout, etc. will generally be the same.[30] How IEC 1508 was followed would be put into the safety case. "The argument within the safety case is normally based on engineering judgement rather than strict formal logic. This is generally supported by some form of probabilistic risk assessment"[31].

"For safety critical systems it is essential that various aspects of the dependability of the complete system e.g. probability of failure per unit time, either be assessed or predicted before deployment"[32].

2.1.5 Developing Safety Critical Systems

When developing a safety critical system the aim is to remove as much complexity as possible since "complexity is a source for design faults. Design faults are often due to failure to anticipate certain interactions between a systems components"[28]. The system should be made as reliable as possible, but care has to be taken because

“software reliability can be increased by removing software errors that are unrelated to system safety thus increasing reliability while not increasing safety at all”[13].
As many faults as possible should be removed from the code. It is rarely if ever possible to remove all the faults. Similarly it is not possible to remove all the hazards from a system and have it still perform beneficial work. The desire therefore is to provide a system that can deal with and counteract the hardware hazards. “A hazard is a state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object) will lead inevitably to an accident”[13].

Before a system is commissioned the risk should be reduced as much as possible. “**Risk** is the hazard level combined with (1) the likelihood of the hazard leading to an accident (sometimes called danger) and (2) hazard exposure or duration. .... usually the most likely hazards are controlled but hazards with high severity and (assumed) low probability are dismissed as not worth investing resources to prevent”[13].

The methods of providing safety in a safety critical system is divided into two parts, fault prevention and fault tolerance. **Fault prevention** is an “attempt to ensure that a computer system is, and remains free from faults”[33]. Potential faults are avoided and those that are present are removed before the system becomes operational. The second approach is **fault tolerance** and “accepts that an implemented system will not be perfect, and that measures are therefore required to enable the operational system to cope with the faults that remain or develop”[33].

2.1.6 Fault Prevention

Fault prevention is used in all of the stages of the waterfall and other development model and is divided into the following methods:-

1. Safety Analysis
2. Fault Avoidance
3. Fault Detection

2.1.6.1 Safety Analysis

This is required very early on in the development process. It is required to identify the safety critical parts of the system. It can identify statistics and levels of safety for the design. This method can be used to set targets for the reliability and availability of the system. Safety analysis is also used after the implementation phase to estimate the likelihood of failure and what the impact of failure is likely to be.[16]

There are various methods of safety analysis including:-

i. Hazard analysis
ii. Fault tree analysis
iii. HAZOP analysis
iv. Safety assurance.

2.1.6.2 Fault Avoidance

Fault Avoidance is used to avoid faults being introduced into the process rather than relying on removing them later. This method should be used in three of the waterfall stages, specification, design and implementation. This method minimises the design faults and "it is generally accepted that a higher quality and cheaper product can be produced if design faults can be avoided altogether rather than removing them later"[16].

Some of the methods of fault avoidance are:-

i. Formal Methods - VDM, Z
ii. Quality Assurance
iii. Structured Programming
iv. Walk throughs

2.1.6.3 Fault Detection

Faults can be introduced during the design and implementation of the system, so it is beneficial for them to be removed as soon as possible. This involves techniques of identifying the fault and then removing them.[16] Care has to be taken to assess the impact of fault removal as it is essential not to introduce more faults. Software does not wear out with time (unlike hardware) but maintenance can introduce faults. [28]

Methods of fault detection are:-

i. Testing
ii. Inspections and walk throughs
iii. Prototyping
iv. Verification and Validation

2.1.7 Fault Tolerance

In fault tolerance it is accepted that there will be faults in the delivered system and so a method of dealing with these faults must be built into the design of the system. When a fault has occurred in the running system there is a four tier process that has to be performed by the system so that normal running can recommence. There is an accepted method of performing fault tolerance which consists of the following:-

1. Failure Detection.
2. Failure Containment and Diagnosis.
3. Fault Recovery.
4. Fault Repair.

[33]

2.1.7.1 Failure Detection

Once the system has been completed there will still be faults present.[34] There will also be random hardware failures. To reduce the effect of these faults, the fault has to be identified by the system. These faults are normally revealed by additional run time checks to detect errors.

There are various methods of carrying out these checks including:-

i. Redundancy - hardware, software
ii. Control flow checks
iii. Self testing
iv. Plausibility checks

2.1.7.2 Failure Containment / Damage Assessment

Once a failure has been detected in a system then the fault should be contained. Errors tend to propagate from where they originate (known as the domino effect). An ideal software failure would lead to human intervention but in many cases this is impractical. This is known as **fail safe**. Most modern systems are **fail operational** which is where the computer has to continue running the system for a limited length of time. An example would be an aeroplane system where the pilot cannot take over entirely nor can the aeroplane stop flying. The closer a system is to a fail safe system the easier it is to design and implement.[12] The system must also identify which parts of the system have been affected by the failure.[17]

Methods of failure containment are:-

i. Defensive programming
ii. N-version programming (diversity)
iii. Redundancy - voting
iv. Return to manual operation

2.1.7.3 Fault Recovery.

"The system must restore its state to a known 'safe' state. This may be achieved by correcting the damaged state (forward recovery) or by restoring the system to a known 'safe' state (backward error recovery). Forward error recovery is more complex"[17].

2.1.7.4 Fault Repair.

Fault repair "involves modifying the system so that the fault does not recur. In many cases, software failures are transient and due to a peculiar combination of system inputs. No repair is necessary as normal processing can resume immediately after fault recovery."[17]

2.1.8 Fault Avoidance and Fault Tolerance

With all the above methods the aim is to have simplicity, intelligibility and traceablity. These reduce the complexity of the system, as complexity makes it more difficult to understand which increases errors.

It is also accepted that the "human factor issues which encompass all aspects of human involvement ...... are always the weakest link in the chain"[35]. This includes the human involved with the design, fault tree analysis, maintenance and the end user interacting with the interface. The "very high standards of reliability can only be achieved through application of fault prevention and fault tolerance; ...... despite the application of fault prevention, complex systems will always be affected by faults"[33].

"Glitches in computer programs are annoying when they cost an hour's work. In critical applications such as telephone networks, nuclear power plants or missile guidance systems, insidious faults can spell disaster. Since even the best proof cannot pinpoint the extent of vulnerability ..... the use of computers should be restricted wherever safety is a primary consideration"[34].

2.1.9 Summary

A computer system can never be made 100% safe. The hardware can fail, the software can fail or the operator can fail. “Problems in human machine interactions have been identified as a major cause in safety critical computer systems”[36]. Safety cannot always be a key design feature since “safety acts as a constraint on the possible system designs”[13]. An important factor that should be noted is that “carrying out an operation in a particular way for many years does not guarantee that an accident cannot occur. Yet informal risk assessments appear to decrease quickly when there are no serious accidents” [13].

In conclusion it is interesting to note that “an engineer once compared designing a new passenger ferry or an aeroplane to throwing knives in a circus act. If everything works that is fine, that’s what you are paid for. But one fatal slip-up and your knife throwing days are over”[37].

2.2 PLCS

“A **Programmable Logic Controller** (PLC) is an electronic device that controls machines and processes. It uses a programmable memory to store instructions and execute specific functions that include On/Off control, timing, counting, sequencing, arithmetic and data handling”[4]. It “is in essence a device that is specifically designed to receive input signals and emit output signals according to the program logic” [5]. PLCs were developed as basic devices that could replace relay circuits. As such they were developed so they could be programmed in a similar way to the design of relay circuits. “It was possible to use them (PLCs) to take over all of the logic functions from relays and replace hundreds of relays with a more compact solid-state unit”[5].

PLCs are much easier to change and maintain than the counterpart hardwired relay system, even so “many PLC systems are designed for one off applications in process control and industrial plant applications. They are bespoke and often need changing as the plant is upgraded"[38].

Today PLCs have taken over much of the logic functions from relay circuits in machine and control applications. The PLC is more compact, cheaper, more reliable, easier to identify faults and to perform maintenance on than the relay circuit. Also it is much easier to change the logic of code than of a hardwired system. [5] The great benefit of the PLC is that "nowadays PLCs have outgrown their rather limiting name, and can do many things that logic relay circuits cannot, such as text handling, sophisticated communications and mathematics"[39].

It is normal to design a PLC in such a way that, should there be a power failure, the logic will return the system to a safe mode. A safe mode is a state where the system cannot allow an accident to occur even though the system has failed, e.g. turn on sprinklers in case of a fire when it is not possible to detect if there is a fire. When PLCs are used in safety critical applications this is even more important and it is normal for it to be taken one step further. There is often a relay circuit hardwired around the PLC software so if there is a failure the hardwired system can prevent an accident from occurring and take the application to a safe mode. This is the practice in many industries, oil platforms or modern cars that are 'drive by wire'.

2.2.1 Structure of a PLC

A PLC consists of 6 parts:-

- CPU (Central Processing Unit)
- Memory
- Power supply
- Analogue input and output cards
- Digital input and output cards
- Programming port

There are two types of PLCs. One type is the 'brick' which has a specific number of I/O ports and cannot be extended. The other type is the 'bus' which is built on a rack so more I/O cards can be continually added and hence the number of I/O ports can be increased.

2.2.1.1 CPU (Central Processing Unit)

The CPU enables the communication between the input, output, memory and the programming terminal. The user's program (which is stored in memory) is executed one rung of the ladder at a time (see 2.2.2) Background programs such as timers are also run at the same time effectively in parallel. The CPU executes all of the user's program and then returns to the start of the program.

2.2.1.2 Memory

There are two types of PLC memory and in each part is stored the following:-

- In ROM (Read Only Memory) is the system program which is basically the operating system.
- In battery protected RAM (Random Access Memory) are the:-
    - PLC variables which are system variables and cannot be seen by the user
    - User's program; this can be changed by input from the programming terminal. Often once the program is finalised the program is copied onto EEPROM (Electrical Erasable Programmable Read Only Memory)
    - User variables, which contain the results of calculations.

System program	ROM
System variables	Battery - protected RAM
User program	
User variables	
User program	Optional read only memory area EEPROM
User fixed data	

Figure 2:2
PLC Memory Allocation [5]

2.2.1.3 I/O

Inputs and outputs are either analogue or digital. The analogue inputs are always read into a buffer at the start of each execution. In some systems this is also true with digital inputs, in other systems the digital input is read in from the hardware as and when required. Reading the inputs into the buffer means that they are consistent throughout the life of a scan, this is one execution of the program as the PLC operating system continually loops the program.

2.2.1.4 Programming Port

The programming port is used to download the software onto the PLC since programs are often written using PC packages.

The response time of a PLC is taken to be the time taken between a rung being executed and then re-executed.

2.2.2 Relay Ladder Logic (RLL)

PLCs were originally coded using **Relay Ladder Logic**. "A ladder logic program is written in graphical notation and is directly equivalent to the circuit diagram that would be used to interconnect a set of relays to perform the same function". [39] Ladder logic is drawn as a ladder, the power rails are vertical bars on the left hand side and right hand side of the diagram (see Figure 2:3). The link between the power rails can either be on or off and represents the flow of power. The left hand rail is on at all times. A vertical link represents an **or** with the horizontal link that it is joining. The state of the vertical link is copied to all horizontal links on the left hand side of it. Contacts will take the value of the left hand side, **and** it with itself and pass the value to the right hand side link. Coils will take the value of the link for themselves and then copy the value to the left side. If power reaches an output then it is turned on while the power is reaching it. The output can either be a hardware device or an internal variable.

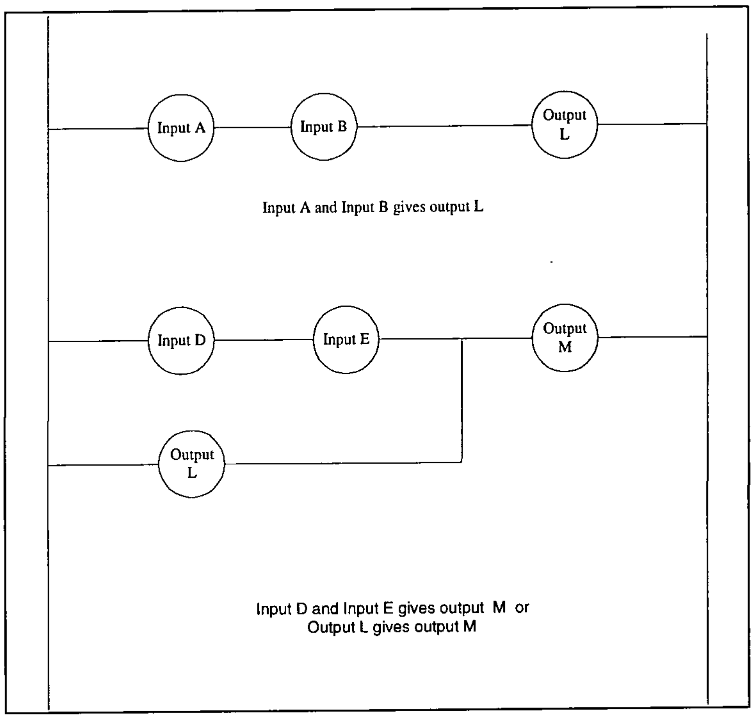


Figure 2:3
Ladder Logic Diagram

In Figure 2:3 input A and input B produce the output L. Input D and input E will give the value of output M or output L will give the value of output M. The logic of the system is as follows:-

L = (A and B)

M = (D and E) OR (L)

which implies M = (D and E) OR (A and B)

The ladder diagram is the program that is executed by the PLC. It reads the inputs and affects the outputs.

When dealing with relay circuits the current flowed and outputs (valves, coils etc.) were all effected simultaneously. With ladder logic each rung is executed sequentially so this can cause ordering and delay problems. It may require an entire loop of the program before an output that is dependent on an input is changed. Care also has to be taken about the ordering; it may require more than one loop of execution of the logic before an internal variable is actually set by the operating system.

Due to the delay that can occur preventing inputs from being read some PLC systems allow interrupts. Where code can be written to execute immediately inputs are changed. Interrupts introduce problems; if they occur too frequently code may be starved. Interrupt driven code may overwrite non interrupt driven code.

2.2.3 IEC 1131- 3

Since ladder logic was first conceived, manufactures have all developed their own syntax and semantics for programming PLCs. Languages have developed from RLL to high level languages of both textual and graphical format. In many cases it is possible to program the code in more than one PLC language. This has increased the problems for system developers, since with each new system developed requiring a different type of PLC, the programmer has to learn a new language. Also code cannot be easily understood by programmers of differing languages.

IEC 1131 was introduced as a standard to attempt to counteract these problems. The aim was to generate an 'open system' for PLC programming that would allow:-

- Lessen the training time
- easier online support and maintenance
- reduced errors and thus improved safety
- flexibility to tackle a wide range of monitoring and control problems.

In some circles it is also felt that this should be taken further and provide portability between development environments and maybe in the future portability between PLCs. [40]

IEC 1131 defines the syntax and semantics of five PLC languages. The high number of languages described is due to the diversity of PLC languages that were on the market prior to IEC 1131.[31] The standard defines a program as a "logical assembly of all the programming elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system"[41]. The fifth language is SFC (Sequential Function Charts) and defines how a program written by a combination of the other programming languages can be combined to one program. The other languages are:-

- Instruction lists (IL)
- Structured text (ST)
- Ladder diagrams (LD)
- Function block diagram (FBD)

IL and ST are both text based languages while LD, FBD and SFC are graphical languages.

All of the diagrams and text are made up of characters. Identifiers can be characters (at least one), numbers and the underscore. There are defined keywords which are not allowed as variables. The literals of the language are Boolean, reals, integers, character strings and time. The integer can be represented in either base 2, 8, 10 or 16. Predefined and user defined types are made up of the above literal types. The access of a variable is assigned when declared i.e. global, externally changed only, constant etc.

In all the languages a function is defined as something that when executed yields one external data value and contains no internal state information. Functions are defined graphically or texturally, dependent on the language. A function may have many inputs but only one output.

All the languages have the same functionality and constructs that can be represented in one language can be represented in all of the others. They all have the same common functions that are pre-declared. They are combined to form a program by using the SFC.

An informal review of each of the languages is now given.

2.2.3.1 Sequential Function Chart

The SFC is written using a combination of the other languages. "The SFC elements provide a means of partitioning a programmable controller program organisation unit into a set of steps and transitions interconnected by directed links. Associated with each step is a set of actions, and with each transition is associated a transition condition"[41]. A step is either active or inactive so the state of the program can be defined by which of the steps are active. Execution of an SFC always starts at an initial step. Each step contains zero or more actions. Actions are instructions in IL, statements in ST, rungs in LD or a collection of networks in FBD or an SFC.

"A transition represents the conditions where by control passes from one or more steps preceding the transition to one or more successor steps along the corresponding directed link."[41]. Each transition contains a single Boolean condition written in either LD, IL, FBD or ST. The transitions provide divergence where the highest priority true transition is followed, convergence, or parallel execution (all the transitions must be true for parallel execution to converge). Convergence and divergence of parallel execution is represented by a parallel line. A step remains active and continues to execute until a transition condition from a step becomes true. The flow of control is from the bottom of one step through a transition to the top of the next step.

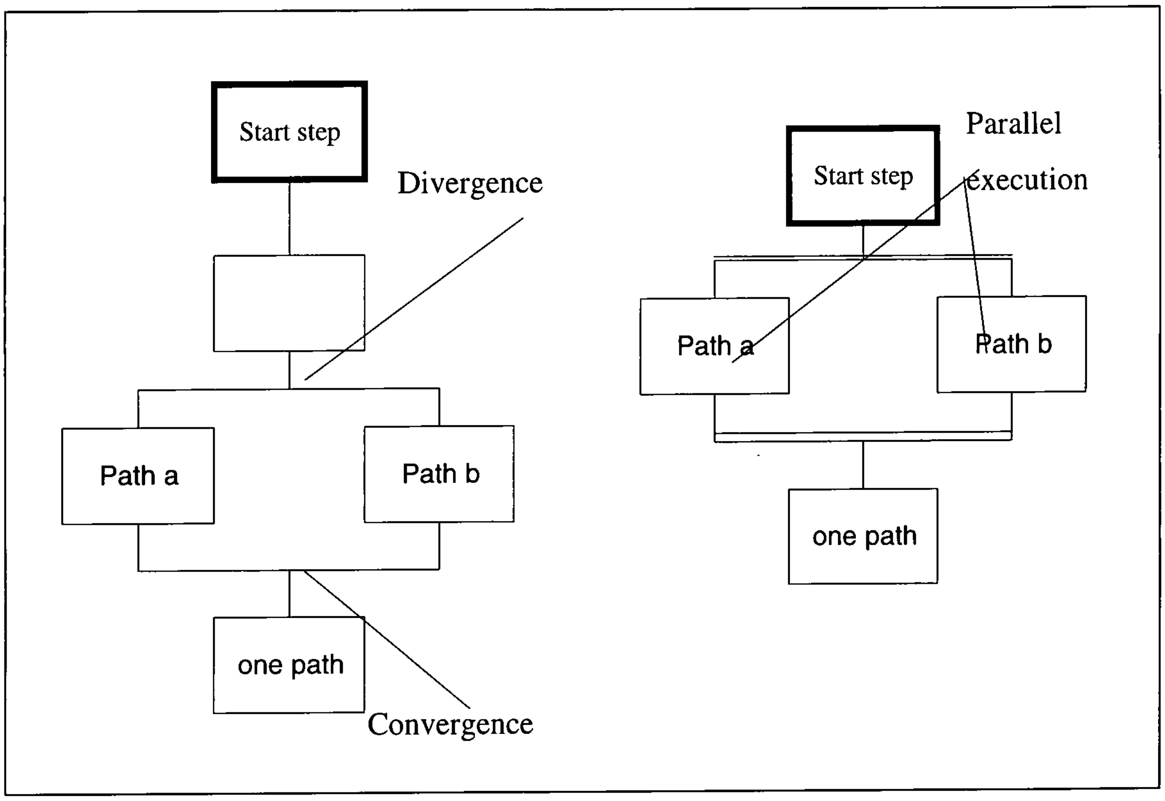


Figure 2:4
Graphical representation of an character drawn SFC

2.2.3.2 Instruction Lists

An IL is composed of a sequence of instructions. Each instruction begins on a new line and contains an operator with optional modifiers (e.g. not) and one or more operands separated by a comma dependent on the operator. The instruction may be preceded by a label followed by a colon (:). The label can be jumped to by putting the name as the operator. A function block can be called using the same method. A comment is an optional extra at the end of each line. With most operators the result of the previous operation provides the first operand of that operation.

Label	Operator	Operand	Comment
START :	LD	%IX1	(* PUSH BUTTON *)
	AND*N*	%MX5	(* NOT INHIBITED *)
	ST	%QX2	(* FAN ON *)

Figure 2:5
An example of instruction list.

A definition of Figure 2:5 would be given as:-

set current result to be equal to operand %IX1
new current result = old current result and not %MX5
store result in location %QX5

i.e.

%QX2 = %IX1 and not %MX5

2.2.3.3 Structured Text

This is a textural high level language that allows:-

- assignments
- case / if statements
- for loops
- while loops
- repeat loops
- exits (from loops)
- function calls with returns.

Figure 2:6 gives an example of the structure of each piece of code scanned in from IEC 1131-3 [41].

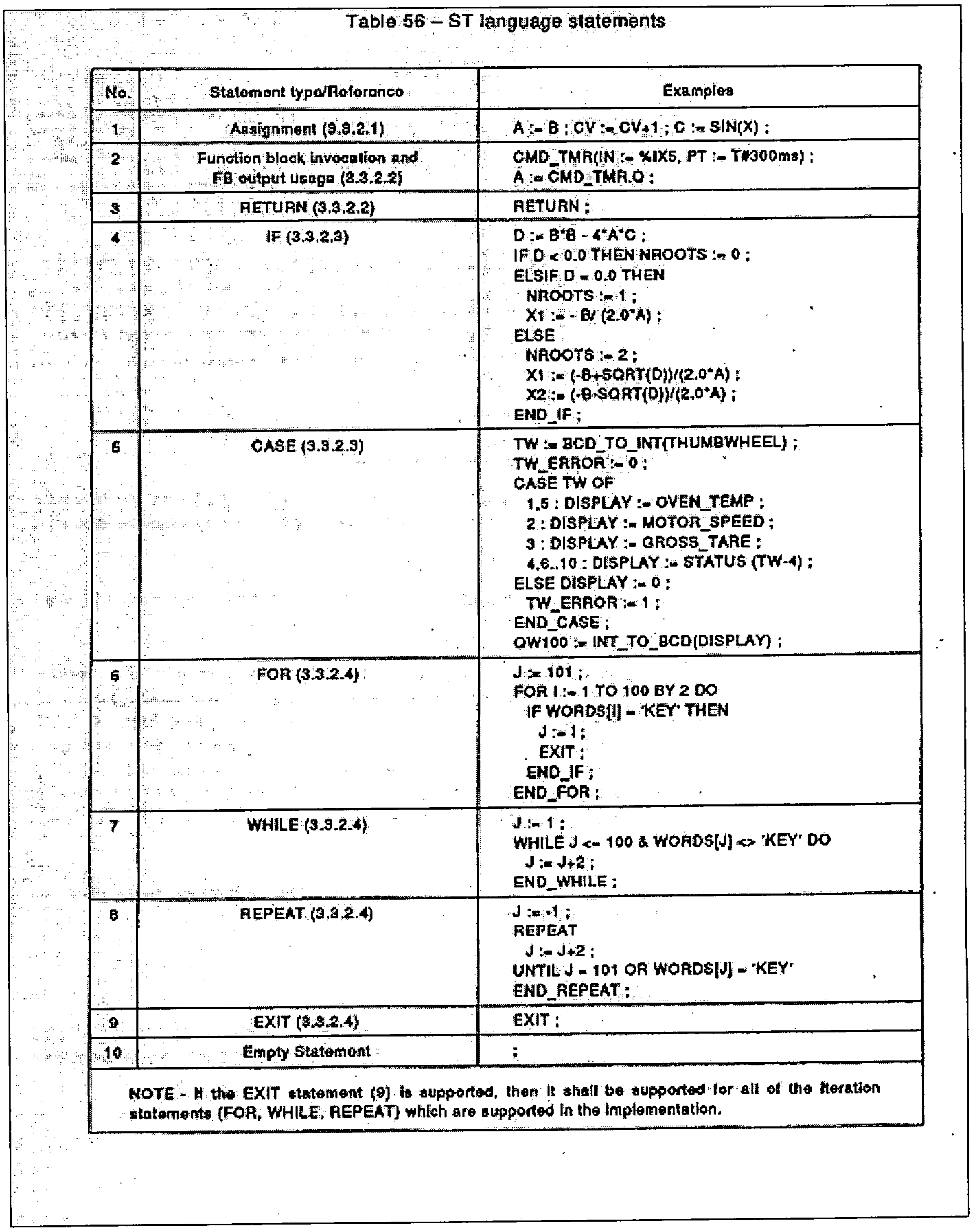


Table 56 – ST language statements

No.	Statement type/Reference	Examples
1	Assignment (3.3.2.1)	A := B ; CV := CV+1 ; C := SIN(X) ;
2	Function block invocation and FB output usage (3.3.2.2)	CMD_TMR(IN := %IX5, PT := T#300ms) ; A := CMD_TMR.Q ;
3	RETURN (3.3.2.2)	RETURN ;

4	IF (3.3.2.3)	D := B*B - 4*A*C ;
		IF D < 0.0 THEN NROOTS := 0 ;
		ELSIF D = 0.0 THEN
		  NROOTS := 1 ;
		  X1 := - B/ (2.0*A) ;
		ELSE
		  NROOTS := 2 ;
		  X1 := (-B+SQRT(D))/(2.0*A) ;
		  X2 := (-B-SQRT(D))/(2.0*A) ;
		END_IF ;

5	CASE (3.3.2.3)	TW := BCD_TO_INT(THUMBWHEEL) ;
		TW_ERROR := 0 ;
		CASE TW OF
		  1,5 : DISPLAY := OVEN_TEMP ;
		  2 : DISPLAY := MOTOR_SPEED ;
		  3 : DISPLAY := GROSS_TARE ;
		  4,6..10 : DISPLAY := STATUS (TW-4) ;
		ELSE DISPLAY := 0 ;
		  TW_ERROR := 1 ;
		END_CASE ;
		QW100 := INT_TO_BCD(DISPLAY) ;

6	FOR (3.3.2.4)	J := 101 ;
		FOR I := 1 TO 100 BY 2 DO
		  IF WORDS[I] = 'KEY' THEN
		    J := I ;
		    EXIT ;
		  END_IF ;
		END_FOR ;

7	WHILE (3.3.2.4)	J := 1 ;
		WHILE J <= 100 & WORDS[J] <> 'KEY' DO
		  J := J+2 ;
		END_WHILE ;

8	REPEAT (3.3.2.4)	J := -1 ;
		REPEAT
		  J := J+2 ;
		UNTIL J = 101 OR WORDS[J] = 'KEY'
		END_REPEAT ;

9	EXIT (3.3.2.4)	EXIT ;
10	Empty Statement	;

NOTE - If the EXIT statement (9) is supported, then it shall be supported for all of the iteration statements (FOR, WHILE, REPEAT) which are supported in the implementation.

Figure 2:6
Example of structured text [41]

2.2.3.4 Ladder Diagrams

The principle of ladder diagrams is as described in 2.2.2 although the diagrams are defined using characters. "A ladder diagram enables the programmable controller to test and modify the data by means of standardised graphic symbols. These symbols are laid out in networks in a manner similar to a 'rung' of a relay ladder logic diagram. LD are bound on the left and right by power rails." [41] Power flow is from left to right. A function block diagram can also be put on the rung of a ladder.

```
|                                      |
|      open              coil          |
+-------| |-----------------( )--------+
|                                      |
|      open                 coil       |
+-------| |--------+----------( )------+
|      closed      |                   |
+-------| / |------+                   |
|                                      |
```

Figure 2:7
A ladder diagram with power rails, links, coils and contacts

2.2.3.5 Function Block Diagrams

Signals flow from the output at the right hand side of a function to the input of the left hand side of the next function block. Each function is a box with inputs on the left hand side and one output on the right hand side and has no side effects. The functionality of the function is described within the block. Outputs can only be joined together via blocks i.e. they cannot be connected.

Ladder logic diagrams can be converted into function block diagrams. Figure 2:9 shows the equivalent function block diagram of the ladder logic diagram in Figure 2:8.

```
                                                              key
        auto           one       two      drive
+-------| |--------+--------|/|--------|/|----------( )------+         -| |- open
|       man        |                                        |         -|/|- close
+-------| |--------+                                        |         -( )- coil
```

Figure 2:8
Ladder logic diagram

```
                                                                  key
      +--------+
auto -| OR     |-------- |                                   +--------+
man  -|        |         |                           input- | name   |-- output
      + -------+         |     +---------+           input- |        |
                         |-----| AND     |   +---------+     + -------+
                               |         |---| =       |
                    one ----o|         |   |         |---drive
                    two ----o|         |   |         |        ------o   not
                               +---------+   +---------+
```

Figure 2:9
Function block diagram equivalent to Figure 2:8

IEC 1131-3 also defines the grammars of the language in its appendix, and examples of all the functions. [41] Appendix IV provides an example of the weigh function in each of the IEC 1131-3 languages, but not the formal semantics.

2.2.4 IEC 1131-3 and Safety Critical Code

When dealing with safety critical code it is felt by Maisey [42] that the language definition should be strict. After his study he felt that "none of the (IEC 1131) languages has been specifically designed for safety critical applications"[42]. He felt that all the languages were close to the application domain which is important. Much of the definition of the language though was by example many of which were "unclear, ambiguous, contradictory or lacking"[42]. IEC 1131 "concentrates on the syntax of the language, but is often less definitive about their semantics. This may lead to problems of ambiguities and implementation differences" [31].

Maisey's studies indicated that the languages had the following characteristics.

- They supported modularization but allowed side effects
- All apart from IL were relatively easy to understand (which is beneficial)
- They did not support traceability
- Checkability was lacking due to
    - lack of strong typing
    - lack of parameter checking
    - lack of boundary checking
- Analysability was lacking due to direct addressing of hardware.

[42]

"Another limitation of IEC 1131-3 is its basis in global variables .... Global variables are not the best way of providing for communications between functions or PLCs." [40]

2.3 TRANSLATORS

A **translator** is "a program which converts statements written in one language to the format of another programming language"[6].

2.3.1 Why Translate

Code is translated for a variety of reasons; the main reason though is to compile it. Compiling occurs when the code is translated from a high level language to machine readable format so that the code can be executed. Code can also be translated from one language to another language; so new compilers for different languages can be used that are more efficient without having to manually rewrite the source code to obtain the benefit of the new compiler. Code is translated so that tools that are available for one language can be used on the code. Building a translator can be more cost effective and beneficial than recreating a large analysis tool base that is language dependent. The translation technology is a well researched and developed technology.

2.3.2 How to Translate

Code is translated from the **source language** to the **target language**. A thorough understanding of the languages, syntax and semantics has to be gained before translation can commence.

Syntax of a language is the "structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs" [43]. **Semantics** of the language are the "relationships of symbols or groups of symbols to their meanings in a given language"[43].

The **grammar** of a language is defined by a set of rules identifying how a sentence (expression) can be built. Just as in English a sentence contains various types of words in a set order, so does a programming language. So the translator can automatically translate the code. One method of describing the grammar of a language is by using BNF (Backus Naur Form)(see Figure 2:10). The semantics can be described in terms of the target language.

Once the languages are defined a 'mapping document' can be written. A **mapping document** is a document that formally defines all the constructs of the source language and then the associated constructs of the target language. This is normally done using natural language, examples and a formal definition of the language. The mapping document (Appendix IV) uses BNF to define the syntax of language; BNF "is a syntactic metalanguage commonly used as notation for presenting language generation"[44].

The grammar defines legal expressions, and how they can be built from a top down approach.

e.g.

<assignment> ::= <variable> ':=' <expression>

An assignment (non terminal) is built up of an expression which is a non terminal followed by an ":=" then an expression which is also a non terminal.

If a non terminal can produce two or more legal expressions then they are divided by a '|' which means or.

Expression in [] brackets are optional

Expressions in { } brackets are optional and recursive i.e. there can be more than one of them.

Expression in <> are non terminals

Terminals are in ' ' marks

Figure 2:10
Definition of BNF syntax used

A non terminal is an expression which can be expanded further, whereas a terminal cannot be expanded any further i.e. it is there in that format.

Grammars that are used in automatic translators have to be 'unambiguous', they are sentences that can only be decomposed in one way when working from left to right along a production rule. "A sentence (or expression) is unambiguous if only one canonical parse exists for that sentence. Computer languages must be defined so that all sentences in the language are unambiguous"[44].

2.3.3 Build an Automatic Translator

Building a compiler is divided into two parts. The first is the front end which is responsible for the analysis of the structure and semantics. The second part is the back end which generates the target language and performs the optimisation see Figure 2:11. With a translator from one high level language to another an intermediate representation of the code such as a tree may not be required. In this case the front end and back end can be combined. A tree would be necessary if the structure and layout of the target language were significantly different from that of the source language.

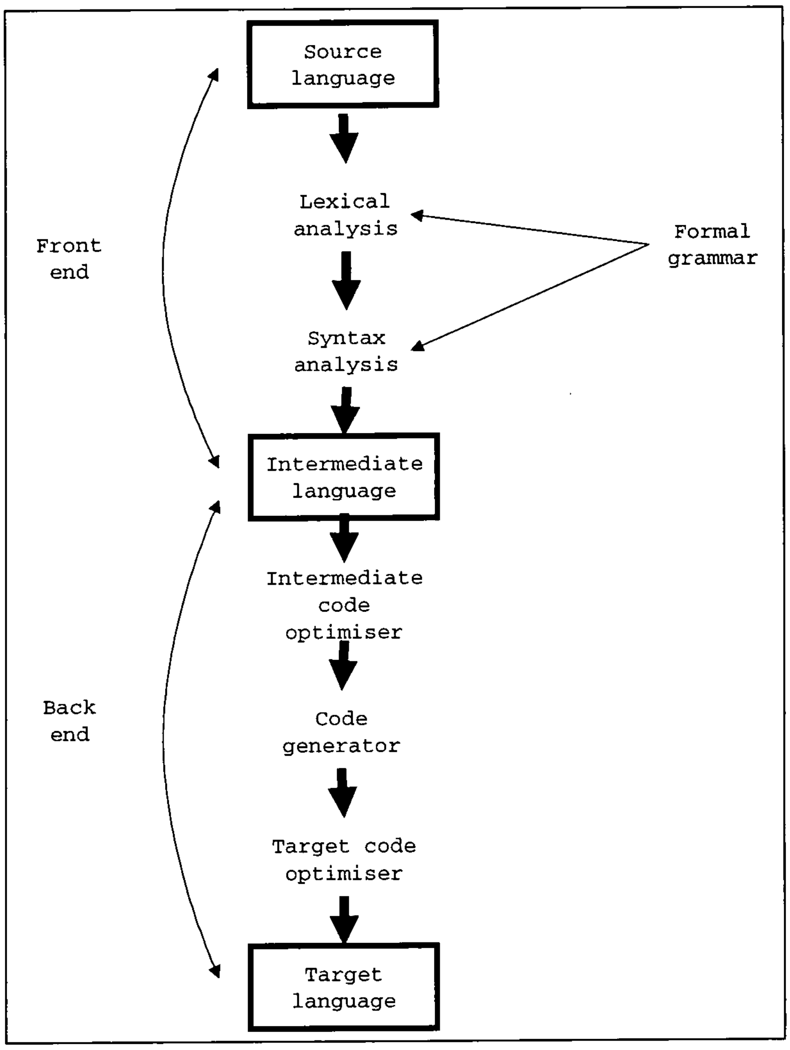


Figure 2:11
Diagram of a Compiler [45]

The lexical analysis identifies all the tokens sequentially in the code. A token is a keyword, a variable, or an operator. "A keyword is a reserved word and may not also be used as a programmer chosen identifier"[46] in most languages. The lexical analyser was a C program, that identified all the tokens. This was done by looking them up in a symbol table and then passing the syntax analyser the token, its type and any value associated with it. e.g. a keyword 'then' would only have to return type 'THEN' (which is assigned an integer value) a variable 'hello' would have a type and the name associated with it.

The syntax analysis identifies if the stream of tokens form a valid sentence including type checking of variables. Then the semantic analysis is performed to build the translated code. There are tools available such as 'YACC' (Yet Another Compiler Compiler) and BISON that allows the grammar of the language to be represented. Associated with each terminal and non terminal is the action to be taken to produce the target language. The compiler compiler then automatically generates the code that can be used to translate the source language.

A translator can parse a representation of the code one or more times. A one parse translator has a simple intermediate representation whereas a two or more parse translator will have a more complex representation. The greater the number of parses the longer translation tends to take but it should result in improved optimisation.

3. TRANSLATION

The translator was built with the aim of translating all the code automatically from the three PLC languages into WSL. This was to be done so the translation could be completed in considerably less time than would have been required for the operation to be performed manually. The translation is also easily repeatable if performed automatically. The process was divided into the following six stages:-

1. Define the source language
2. Define the target language
3. Write the mapping document defining how the source language maps to the target language.
4. Build the translator
5. Translate the code
6. Test the translation

The source languages were the Siemens languages used on the Siemens TI PLC using the APT (Application Productivity Tool). The grammar used by the BISON translator is located in Appendix I. The target language was WSL (Wide Spectrum Language), the grammar can be found in Appendix II. The formal semantics of the language can be found in [9]. The mapping document was written using natural language, formal grammars and examples and can be found in Appendix III. Translation of PLC languages into WSL had not been previously undertaken, so new research had to be carried out at this stage. The remainder of this chapter provides an introduction to the source and target languages, it also summarises the remainder of the six stages of translation.

3.1 SIEMENS LANGUAGES

The programs studied in this project were written using the Siemens TI PLC using the APT. The tool predates IEC 1131-3, so the languages are described below, paying attention only to the functionality used by the studied code.

The programs were written using a combination of the following three Siemens APT languages.

- Sequential function chart (SFC)
- Continuous function chart (CFC)
- Math language (Structured text)

SFCs and CFCs are both graphical languages while math language is a text based language. The math language can only be used within one of the other two languages. The code is developed to run on one PLC.

3.1.1 Modularity

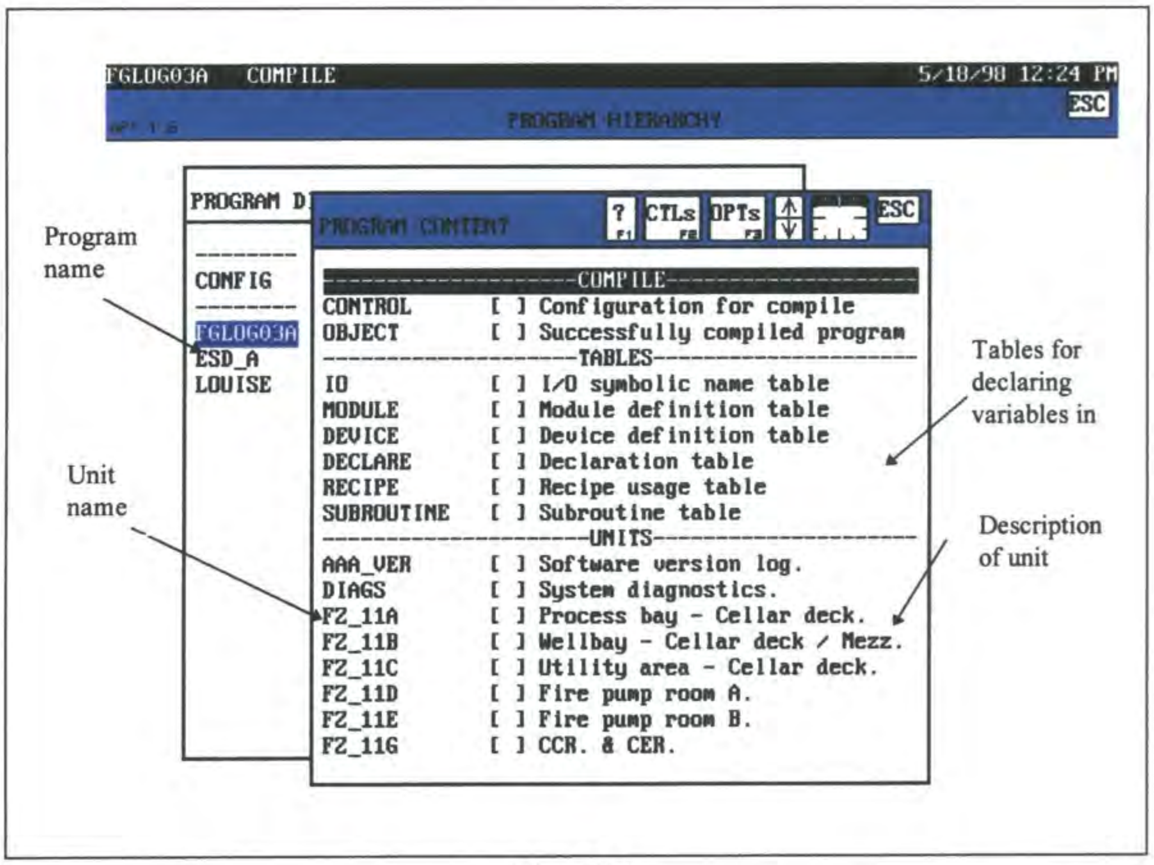
All the code written in each of the languages combines to make the program. The program can be divided into units, which are similar to modules. The SFCs and CFCs can be built in each of these units, the code is normally divided into units on the basis of the hardware functionality e.g. fire zones in the F&G system. In both samples of analysed code each unit contains at least one CFC, but only one unit contains a SFC.

The two graphical languages contain blocks into which the math language is coded. This in effect compartmentalises the code into modules and procedures.

3.1.2 Variables

Variables can be declared in one of two ways, within the math language text or externally to the programming languages within the environment. Variables declared in a math language block are defined in the declaration section at the beginning of the code. They are local to that math block and can only be of the following types: integer, real, timer, Boolean, or arrays of the above except timers.

The remainder of the variables are declared in tables provided by the environment. The variables are declared in tables within a unit and are global to all CFCs and SFCs within that unit or in tables outside all the units in which case they are global to the entire program. Variables declared in the tables can be given a hardware address so they can be accessed externally, ie via a data link. The types of variables that can be declared in the tables are: Analogue and Digital I/O, Word I/O, Valves, Integers, Reals, Flags, Booleans, Timers, Arrays (of a variety of types), Text, Recipes and other types that were not used in the analysed code.



```
FGLOG03A   COMPILE                                         5/18/98 12:24 PM
                              PROGRAM HIERARCHY                         ESC

PROGRAM D   PROGRAM CONTENT              ? CTLs OPTs            ESC
                                         F1   F2   F3
---------
CONFIG      -------------------COMPILE-------------------
---------   CONTROL       [ ] Configuration for compile
FGLOG03A    OBJECT        [ ] Successfully compiled program
ESD_A       ---------------------TABLES--------------------
LOUISE      IO            [ ] I/O symbolic name table
            MODULE        [ ] Module definition table
            DEVICE        [ ] Device definition table
            DECLARE       [ ] Declaration table
            RECIPE        [ ] Recipe usage table
            SUBROUTINE    [ ] Subroutine table
            ---------------------UNITS---------------------
            AAA_VER       [ ] Software version log.
            DIAGS         [ ] System diagnostics.
            FZ_11A        [ ] Process bay - Cellar deck.
            FZ_11B        [ ] Wellbay - Cellar deck / Mezz.
            FZ_11C        [ ] Utility area - Cellar deck.
            FZ_11D        [ ] Fire pump room A.
            FZ_11E        [ ] Fire pump room B.
            FZ_11G        [ ] CCR. & CER.
```

Figure 3:1
Screen print of the upper level of a program allowing variables and units to be declared

As can be seen from Figure 3:1 the variables are declared in one of the following tables.

Input / Output	both analogue and digital input and outputI/O is supported
Devices	a device is an object that uses a collection of I/O points to monitor and manipulate a field device
Declare	this is the table in which most variables are declared, integers, flags, arrays, timers etc.
Recipes	user defined structures (records) for associating numerous variables of possibly different types to a single variable.

The CFCs, SFCs and CFB (a block within a CFC) do not call each other directly. The only interaction between blocks and charts is via global variables i.e. parameter passing is not allowed. It is possible though to affect whether a SFC or CFB is on and hence able to execute on the current loop of the code, this is possible because each CFB or SFC is treated as a variable name that can either be on or off.

3.1.3 Sequential Function Charts

Pictures are easier to understand than text. The charts also provide an overview of how the code interacts. SFCs are used to specify a sequence of events during the control process. These charts are made up of steps and transitions. "Each step can contain one or more commands, each transition contains one conditional expression."[47] Parallel branches are steps that execute concurrently; as such more than a single branch can be followed at a given time (parallel branches were however not used in the analysed code). A selection branch allows a choice to be made as to which branch to follow. The first transition that is true indicates the path that should be followed. Transitions are tested from left to right. Figure 3:2 gives an example of a SFC.

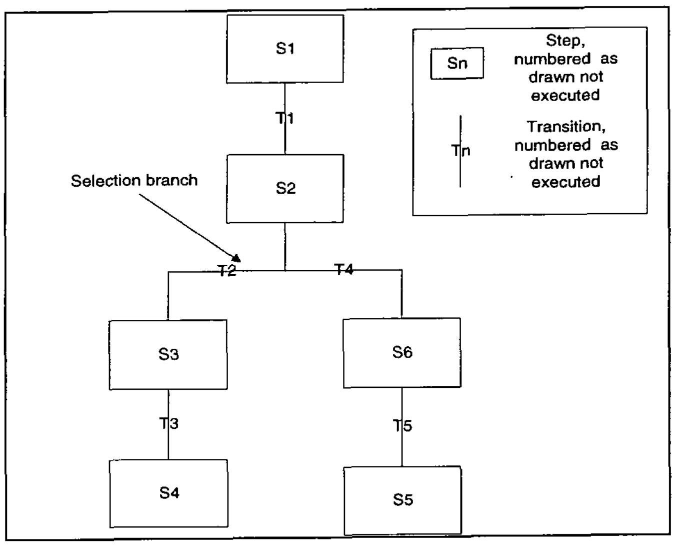

S1
T1
S2
Selection branch
T2
T4
S3
S6
T3
T5
S4
S5
Sn
Step, numbered as drawn not executed
Tn
Transition, numbered as drawn not executed

Figure 3:2

Sequential Function Chart

The charts are drawn using the picture icons supplied; each of the boxes are steps while a transition is the line between the steps. A step can contain a math language program or it can be empty. A transition contains a math language condition that if true allows the next step to be called. If all the transitions from a step are false the active step is re-executed. (The active step is the one that has control at any given time.)

Actions and steps are either predefined or written in the math language. Note that this is similar to the IEC 1131-3 sequential function chart and as such branching (i.e. decisions are made) and then convergence of the code are allowed which is in effect providing a conditional statement. Loops can be written by drawing a transition to end above a previously executed step.

The code in each step is divided into an initial part (optional) and a body part. The initial part of the code is always executed once, but the body of the code may not necessarily be executed. This is because transitions from the active step are tested after the initial section and then again after each execution of the body section.

The SFCs in the ESD and F&G code have a safe SFC associated with them. A safe SFC is associated with one SFC and it has a Boolean expression associated with it. If at any point during the execution of the main SFC the safe SFC has been activated and the expression becomes true, the execution of the main SFC is stopped and the safe SFC takes over execution. After the safe SFC, flow of control passes to a predefined position in the main SFC - often towards the end. Before the main SFC completes its final step the safe SFC has to be deactivated so that it cannot be called while the main SFC is not executing. This enables the code to be skipped which may be dependent on external interaction with the code such as a manual opening of a valve that the code within the SFC will check is shut. This would cause problems if the code was to be executed at the given time.

Figure 3:3 and Figure 3:4 are the main SFCs in the F&G and ESD code

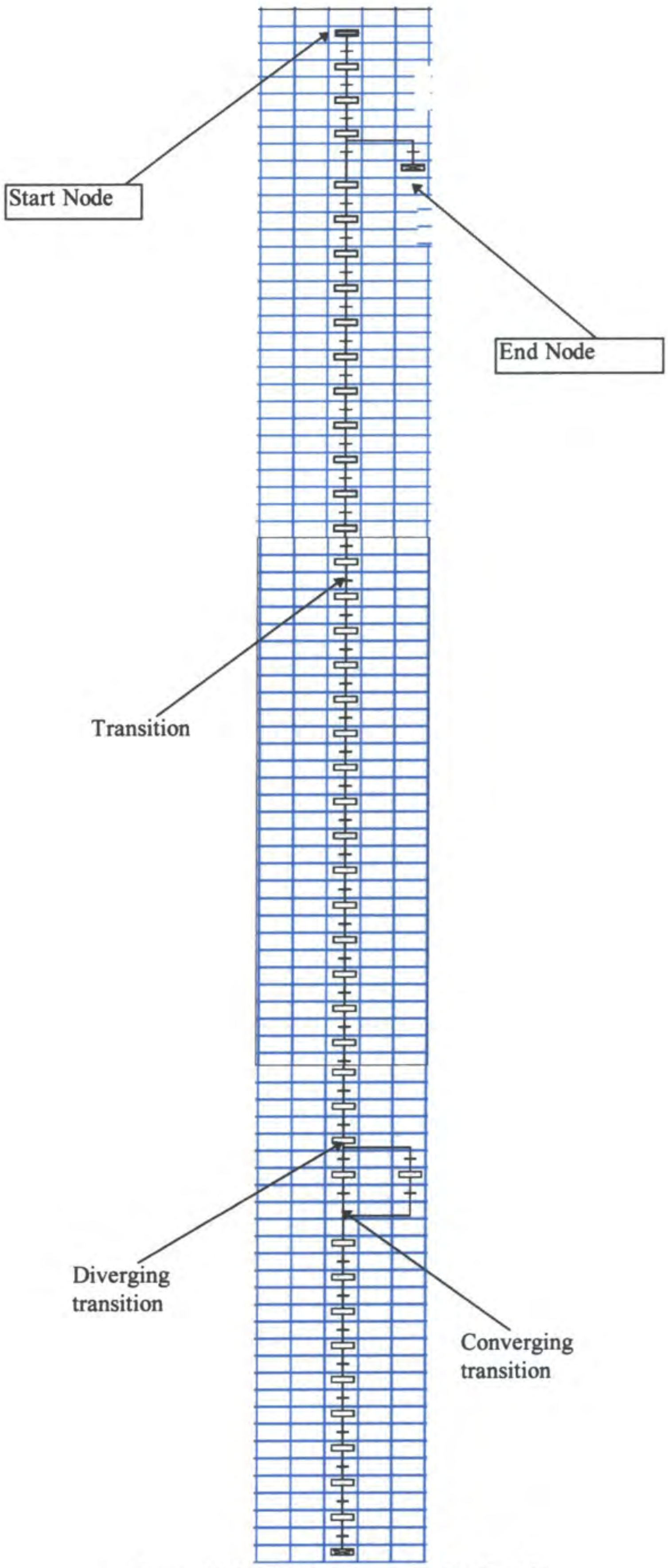


Figure 3:3 Main SFC from the F&G code

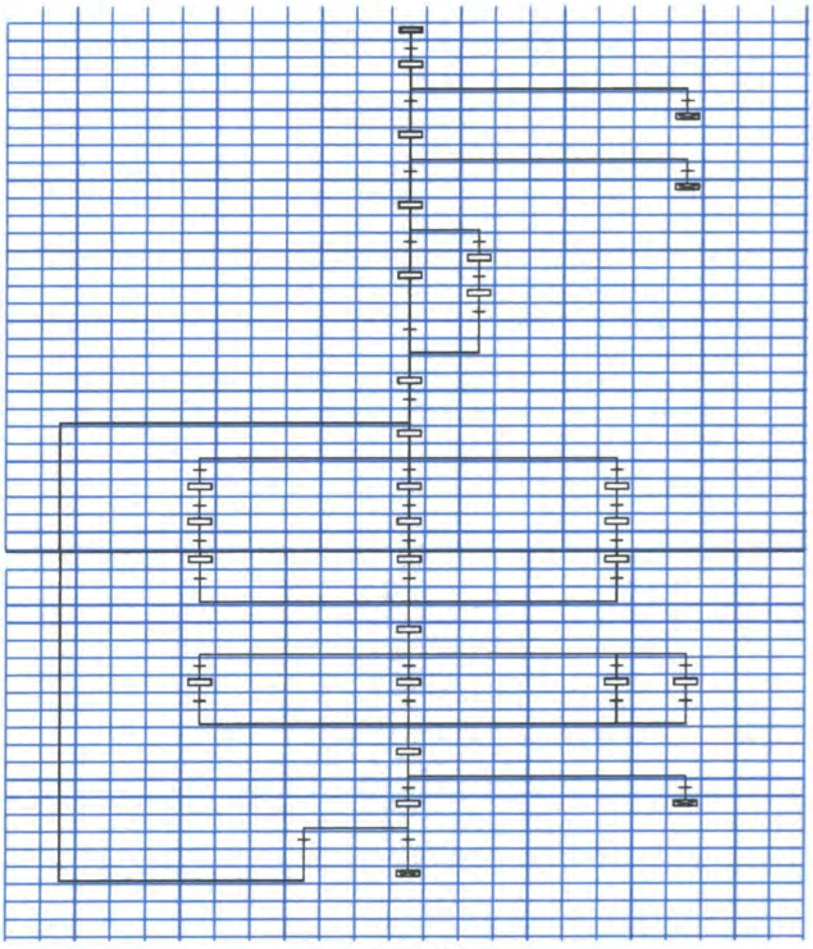


Figure 3:4
Main SFC from the ESD code

1.1.1 Continuous Function Chart

A CFC consists of CFBs; in the defined language they may have inputs and outputs and can be connected via lines drawn on the chart. The contents of 45 blocks are predefined and the meanings cannot be changed. The sub language that was used in the analysed code only had two types of blocks: mathblocks and interlocks, neither of which were predefined. The CFBs used were defined by the programmer using math language.

Figure 3:5 is an example of a CFC containing only interlock CFBs.

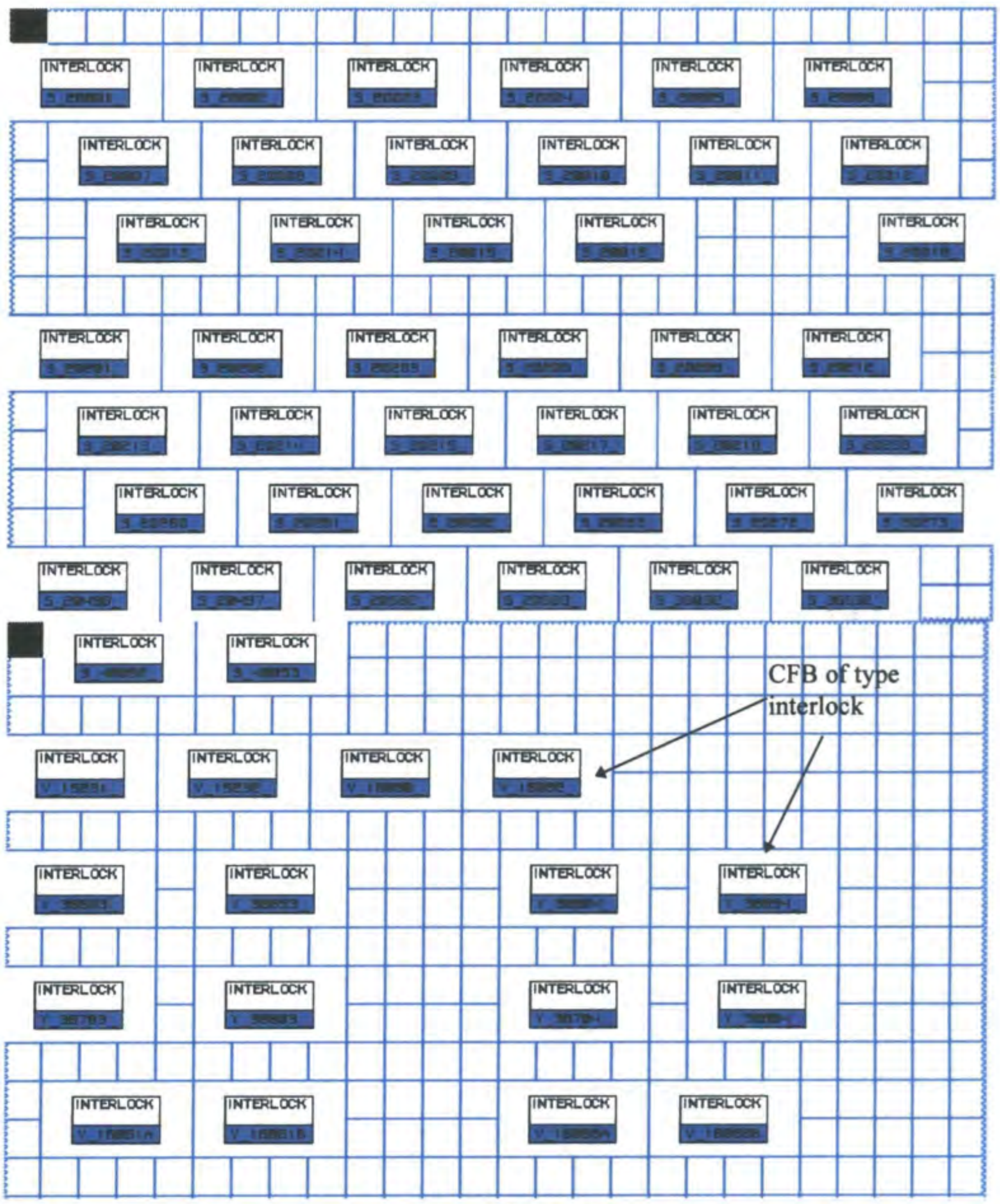


Figure 3:5
A CFC from the F&G code

1.1.1.1 Interlocks

Interlocks are used extensively in the F&G and ESD PLC logic. An interlock contains math language (see 3.1.6); it has no inputs or outputs. It cannot be turned on/off from within the code. It is continually on i.e. available to be executed. The interlock begins executing as soon as the controller is in run mode. Interlocks are given either high or low priority. The priority indicates execution order, all high priority CFBs are executed before low priority ones. An interlock can be compiled to either Relay Ladder Logic (RLL) or special function program (SFPGM); RLL executes faster. There is a subset of math language that cannot be compiled into the Relay Ladder Logic. The first time each interlock is run it can be initialised and then the initialising code is not executed on further iterations.

3.1.4.2 Math Blocks

A math block is similar to an interlock except that it can have inputs and outputs and many are predefined so cannot be coded directly. Some types of mathblocks can be turned off from within the code, although the mathblocks used in the analysed code cannot be. The mathblocks are connected via outputs of one block being the input of another. The mathblocks that were used did not have any inputs or outputs, so were stand alone, and were all of the same type with no predefined functionality, i.e. the functionality was defined by the programmer using math language.

3.1.5 Compilation Order

Unlike most programs the order that code is executed cannot be specified. All the code is executed once before any part is re-executed in the PLC loop. Code that is declared within a unit may not necessarily remain together at compile time. Even code that is declared within a CFC may not remain in a block after compilation. Once compiled though, the execution order remains the same. The ordering of the code is as follows:-

- high priority interlocks
- SFC
- CFB math blocks / low priority interlocks

3.1.6 Math Language

The Math Language is a text based language for performing mathematical calculations. It can be used in the step of a SFC, the transition of a SFC or in the interlock or mathblock of a CFC. As with the structured text of the IEC 1131-3 it is made up of expressions that are valid combinations of terms (constants or variables) and operators. The language is strongly typed.

The main statements used in the code are conditional statements and assignments. The assignments allow both mathematical expressions and /or Boolean algebra. A number of calls are made to the APT defined functions and procedures. There is only one while loop in the ESD code. There is no concept of pointers although specific hardware addresses can be written to and read from directly. Some of the predefined variable types have more than one variable attached to them e.g. analogue inputs and timers see mapping document Appendix III.

Each math language block can have an initial part. Within a SFC step the initial part is executed during the first run through the step every time the SFC is run. Within a CFB the initial part is executed only the first run through the block after the controller is switched on, it is never executed again.

3.2 WSL (WIDE SPECTRUM LANGUAGE)

WSL is a text based language and "is designed in such a way as to support the forward development of programs by stepwise refinement from a specification, or the reverse engineering of an executable program to a specification, within the same language. It contains both high level specification constructs and executable statements, and allows these to be combined within the same program." [48]

"The meaning of WSL programs is mathematically defined by formally specifying the semantics of WSL statements. Because of this formalism, transformations can be mathematically proven to preserve the meaning of programs" [48].

WSL was developed for transforming legacy systems. It contains both high level and low level constructs. It contains statements which include loops, assignments, conditional statements and procedure calls. There is no concept of a type being associated with any of the variables. Sub sections of code can be used in conjunction with the transformations. A more detailed language introduction can be found in Ward, [9].

Appendix II contains a copy of the grammar for WSL while a copy of the semantics can be found in [9].

3.3 MAPPING DOCUMENT

The mapping document can be found in Appendix III. The first task of writing the document was to identify from the manuals how the hardware and the programming environment worked. Then to identify if the translation would be possible. This involved being able to automatically identify all the individual blocks of the code, CFCs, CFBs, SFCs - steps and transitions. The variable's information including: name, type and scope. Extracting the variable information was difficult as it was all stored in binary files.

The next task was to decide how to layout the final WSL program as two graphical languages and one text based language had to be converted into a single text based language. Each of the blocks in the graphical languages were represented by procedural calls.

Every construct in the math language used had to be defined in BNF and then in the corresponding WSL form. This was often problematical as the math language was not formally defined and in some places it was ambiguously defined i.e. how the flags worked. Another problem was that not all of the desired constructs were available in WSL e.g. there is no XOR function in WSL. Variables such as timers were difficult to represent as WSL does not have any concept of time. The math language functions and procedures were often not supported within WSL so a decision had to be made as to whether a function/ procedure had to be written for each of them. A decision was taken to declare them as external functions/ procedures that had an unknown meaning within WSL.

Translating the internal workings of the SFC was then defined. This was not automatable as there was no obvious way automatically to understand the graph. Each of the individual steps and transitions could be automatically translated and then the ordering of the steps and transitions had to be supplied manually. The re-execution of steps and the decision making process at transitions was complex to map to text. This was performed by putting the SFC into an action system and re calling the step if the transitions failed.

The mapping document was beneficial as it identified all the constructs that had to be included in the parser. Once the mapping document was written developing the parser and mapping document became an iterative process. When the parser was tested it identified flaws in the understanding of both the math language and WSL. The hardest part of writing the mapping document was when there was no obvious equivalent construct in WSL to a PLC construct. The two major examples of this were timers and the SFC. The timers were problematical since WSL lacked time information and the ability to represent background processing. The SFC was difficult as the graphics represented information that was not possible to represent directly in WSL, so as close a representation as possible was used.

The easier part of the mapping document was that no intermediate representation of the code was required since the syntax of the math language and WSL were similar enough to provide a direct mapping. The math language also had a relatively small number of constructs to define in WSL. The number of different variable types was high and, many of them had elements and commands associated with them, all of which had to be defined in BNF and WSL. The syntax was not formally defined so in some instances definitions were identified from examples in the code. Mapping the CFCs was straight forward as only three types of CFBs were used and they were all distinct. The SFC was, as described above, not easy.

Mapping to WSL was reasonably difficult as many of the constructs were not typical of text languages such as the math language or Pascal.

Eg a := b or c had to be translated to a := if (b or c) then true else false

The lack of variable declaration and variable type in WSL was difficult to relate to, but it meant that it was possible to map all the variables into WSL which may not have been possible in other high level languages. The SFC was also possible to map into WSL by using the action system construct.

3.4 BUILD THE TRANSLATOR

The translation took place in two phases: the SFC was translated to an action system (see mapping document Appendix III) and the mathblocks were individually translated and then the whole code was assembled.

3.4.1 Math Language Translator

BISON was used to write the translator to translate the math language. A C program was written to perform the lexical analysis and identify the tokens. The tokens were identified from a look up table. As each new variable was identified it was added to the symbol table for use at a later date. The variables that were declared in tables were stored in files as there were too many to be stored in memory. The grammar of the language was entered into BISON to perform the syntax checking. Corresponding to each grammar rule a piece of code is written to describe how to generate the target language. BISON automatically generates a C program to perform the syntax checking and to generate the translated code. The size of some of the statements (conditional statements especially) meant that conventional grammar development in BISON was not always possible.

There were two major problems encountered when building the parser. The first was that the last statement of a block in WSL did not end with a ';'. The parser had to be defined having a first and other statements declared in the parser and if there was another statement in the block then the ';' had to be inserted. The second problem involved storing the long conditional statements and assignments in memory. Since some of the statements were too long they were all written to the file immediately on parsing; this was not possible for statements using XOR as XOR was not a primitive in WSL. The operands had to be passed to a locally declared function in WSL but this was not possible if the operand had already been printed to file. The solution was for the parser to fail to parse the XOR function and a different parser used to parse these mathblocks. The major problem during the translation process was memory shortage using both a PC with 16M of memory and the UNIX box.

3.4.1.1 SFC translation

The SFC in stored as a single file with each step beginning with a ^Sx where x represents a number. Each of the transactions start with a ^Tx. The first stage was to separate the file so each step is contained in a different file. Each of the files were then translated using the math language translator, the steps are entire math programs and the transaction are conditions. The next part of the translation involved building the action system this involved building the translated steps into an action system. The splitting up of the file and translating each of the files was controlled by a Perl script. The building of the action system was controlled by a C program. The order of the steps had to be supplied manually to the C program.

3.4.1.2 CFB translation

Each of the math blocks were translated using the translator and stored in a file of the math block name in the directory with the name of the CFC in a directory of the unit name. This was controlled by another Perl program that would call the translator and supply it with the mathblock name and the name of the output file.

3.4.1.3 Entire program

All information about variables, names of units, SFC, CFCs and CFBs was obtained. Then the SFC was translated, then each of the individual CFBs were translated before the entire program was assembled. Each of the individual mathblock procedures were inserted in turn as was the SFC block. This whole procedure was automated with a Perl script.

3.5 RESULTS

The translation was performed in its entirety. The translation identified some very long statements. It also identified an instance where a variable had the same name as a keyword. There was some trouble understanding how the PLC code should be written from the manual but this was sorted by using examples from the code.

The translation process collected a list of variable names and types. Each CFB was translated into WSL and stored in its' own file. A list of variables that were used incorrectly was stored as was the a list of CFBs that had failed to parse. If the block had failed to parse then either there was an error in the parser or the math language, both were identified using the parser, the first were fixed the second analysed, faults included assigning Booleans using the commands to assign flags.

The translation process was performed in about 7 stages. This was so stages could be debugged and developed individually and also so they could be performed on demand. The variable information for each of the systems was only extracted once from the binary files. It was necessary only once to identify the names of unit's, CFCs, CFBs and their types. The WSL could be rebuilt in a different manner without re-translating the code. Since the ESD and F&G code were long each task took hours rather than minutes e.g. translating all the math language would take a day. The other advantage of staged translation was that the parser could be changed to provide useful data, and the whole translation process was not necessary. Data that was collected included number of lines of code in each math block, maximum level of nesting of conditional statements, where variables of a specific type were used. Calculations performed during parsing of the number of if, else and elsif branches provided the number of test cases that would be required for analysis in 5.20. Changing the parser was also used to determine if and where various parts of the math language had been used.

The final WSL contained one action system and a high number of procedure calls. The ESD and F&G code looked quite similar in layout and structure.

4. THE CODE

The translation of the code generated much analysis material. The actual translation required identification of all the variables and their types. The number and length of each of the CFC and CFBs could be calculated. The translation also enabled much analysis to be performed on the code. This chapter identifies the general characterises of the ESD and F&G code on the offshore system.

The number of lines of WSL code (including blank lines) is given in Figure 4:1.

	ESD	F&G
With Comments	199,431	88,607
Without comments	99,757	51,171

Figure 4:1
Number of lines of WSL code

Both of the code samples were predominately CFCs situated in units. These CFCs contained CFBs most of which were interlocks although a few were of type mathblock. There is one main SFC and one safe SFC in each piece of code in the self test unit.

The code contained two types of global variables, those global to the entire program or those global to a unit. Locally declared variables in the math blocks can only be of the type integers, reals, Booleans, arrays or timers.

The F&G system has 3958 global variables and 35 variables that are global to the units. Of these variables 529 are input variables and 168 are output variables, hence 3296 are internal variables: although some of these have user defined hardware addressed so can be changed externally. The ESD system has 4413 globally variables and 5899 variables that are declared global to the units. Of these variables 1567 are input variables and 1047 are output variables, hence 9792 are internal variables. The quantity of each type of variable is detailed Figure 4:2 to Figure 4:5.

Fire and Gas system

Type	Number
analogue input	201
Boolean	1698
Boolean array	48
recipe	3
digital flag	96
digital input	325
digital output	67
recipe	58
DO10 array	41
fast timer	17
integer	1235
integer array	13
real	3
recipe	131
slow timer	14
word input	3
word output	5
single valve	29
dual valve	2

Figure 4:2
Number of F&G global variables

Type	Number
Boolean	35

Figure 4:4
Number of F&G variables global to one unit

ESD system

Type	Number
analogue input	229
Boolean	909
Boolean array	68
digital flag	364
digital input	1337
digital output	682
DO10 array	19
flag	478
integer	259
integer array	2
slow timer	4
text	59
text array	1
word input	1
word output	1

Figure 4:3
Number of ESD global variables

Type	Number
Boolean	3487
Boolean array	19
DO10 array	441
flag	663
integer	884
integer array	4
real	5
slow timer	51
text	100
text array	1
recipe	5
recipe	55
recipe	71
recipe	93
recipe	10
word input	6
word output	4

Figure 4:5
Number of ESD variables global to one unit

The ESD code has 38 units. 1 main SFC with 1 safe SFC connected to it. There are 184 CFCs containing 1990 CFBs of which 1253 are high priority interlocks, 734 low priority interlocks and 7 active mathblocks. The F&G code has 55 units. 1 main SFC with 1 safe SFC connected to it. There are 263 CFCs containing 1791 CFBs of which 1789 are high priority interlocks and 2 are active math blocks. Math language was used to provide the functionality of the CFBs and the steps in the SFC.

The math language statements that were used are:-

- Assignment statements
- Conditional statements
- While loop
- Procedural and function calls (APT defined procedures and functions)
- Comments

Although while loops were available they were only used once in the ESD system and not at all in the F&G system. This was unusual for a large application in a 'normal' domain. Even small pieces of code tend to have a high number of loops. Analysing code is theoretically easier the less loops that it contains.

Following are graphs giving CFC and unit information. Graphs Figure 4:6 and Figure 4:7 demonstrate the number of continuous function charts per unit. The graphs Figure 4:8 and Figure 4:9 show the number of CFBs per unit. Notice how the number of CFBs is dependent on the number of CFCs that are situated in each of the units. There is a high correlation between the number of CFBs and the number of lines of code in each of the units, this is demonstrated by graphs Figure 4:10 and Figure 4:11.

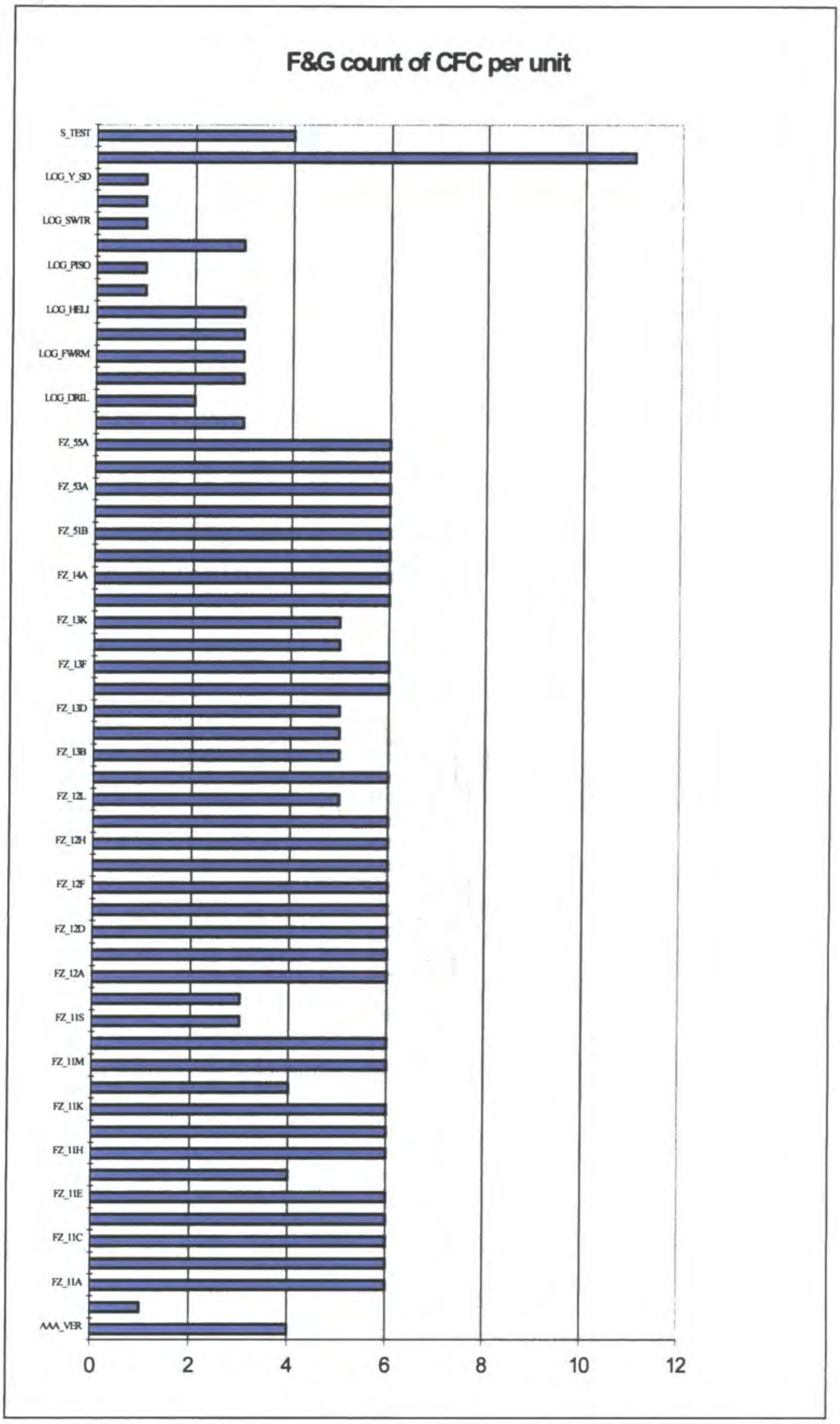


Figure 4:6
Graph showing number of CFC's in each unit of the F&G program

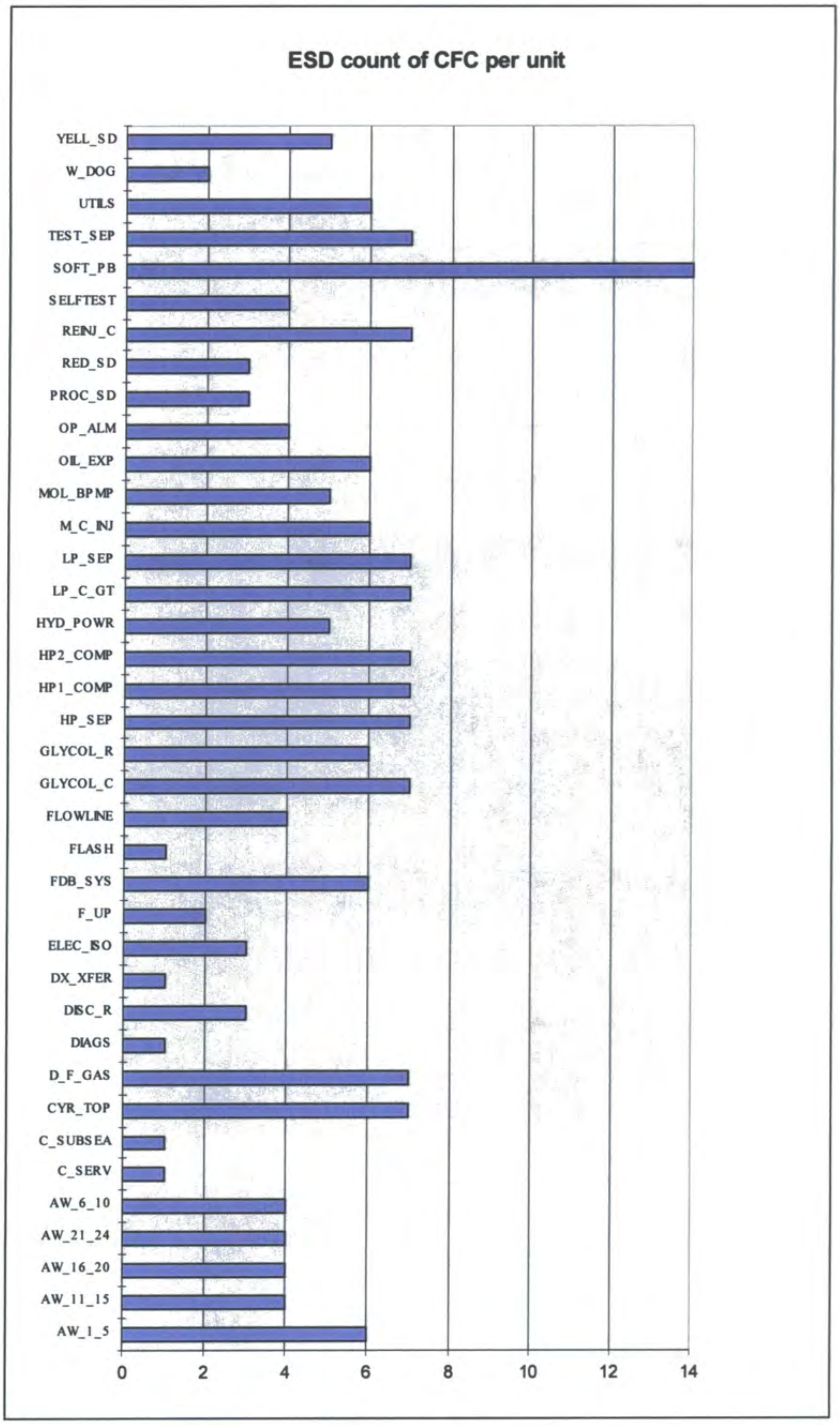


Figure 4:7
Graph showing number of CFC's in each unit of the ESD program

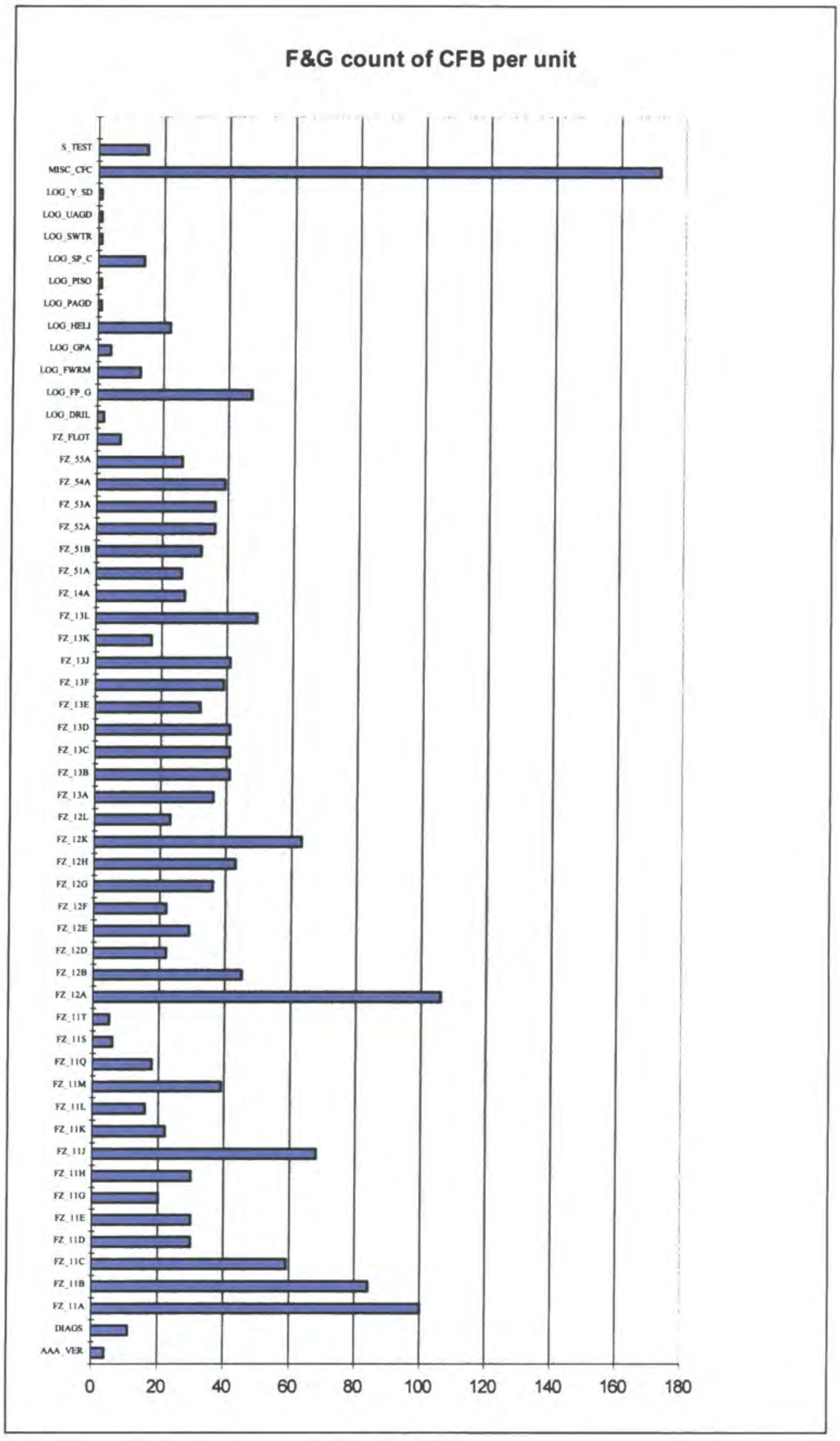


Figure4:8
Graph showing number of CFB's in each unit of the F&G program

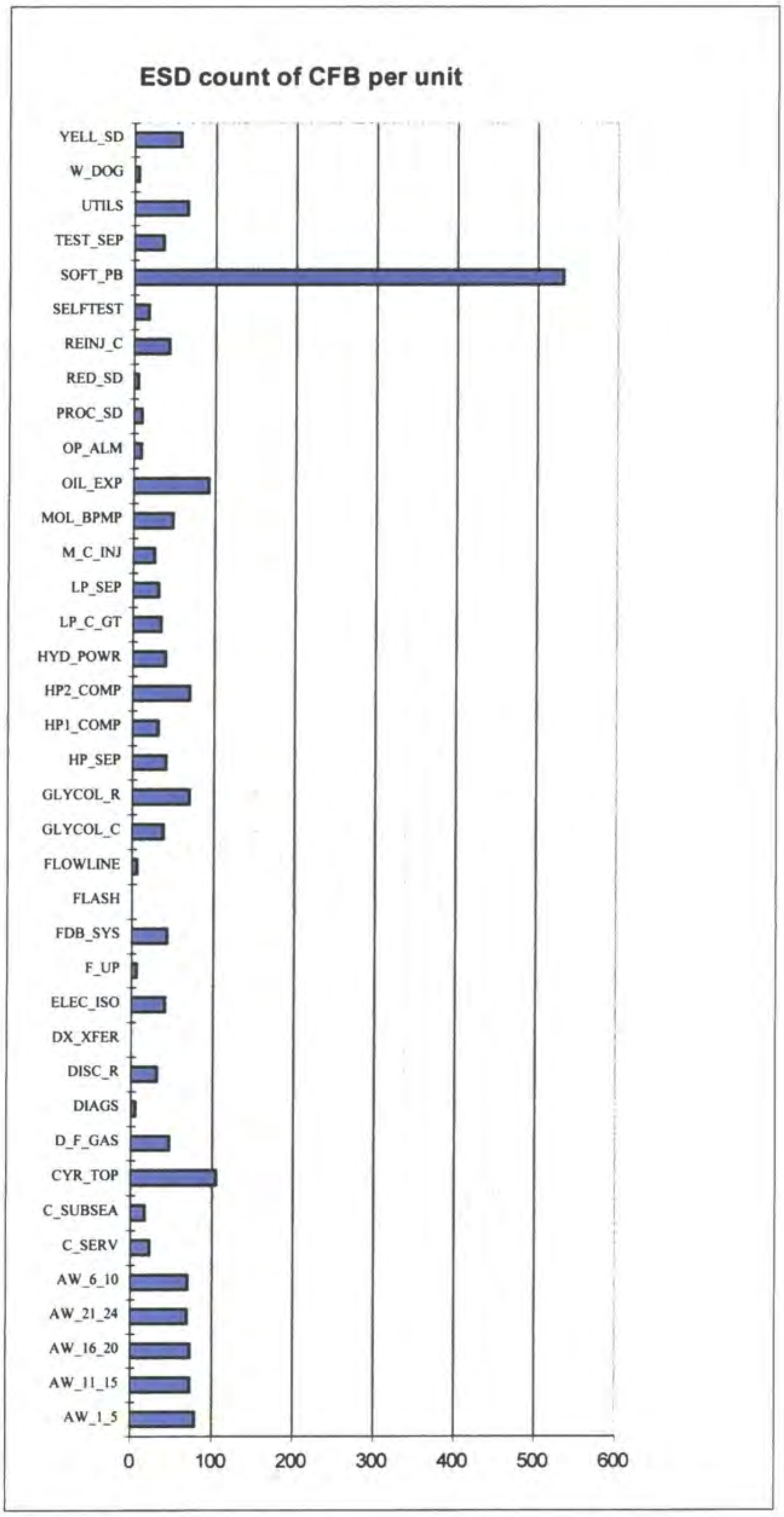


Figure 4:9
Graph showing number of CFB's in each unit of the ESD program

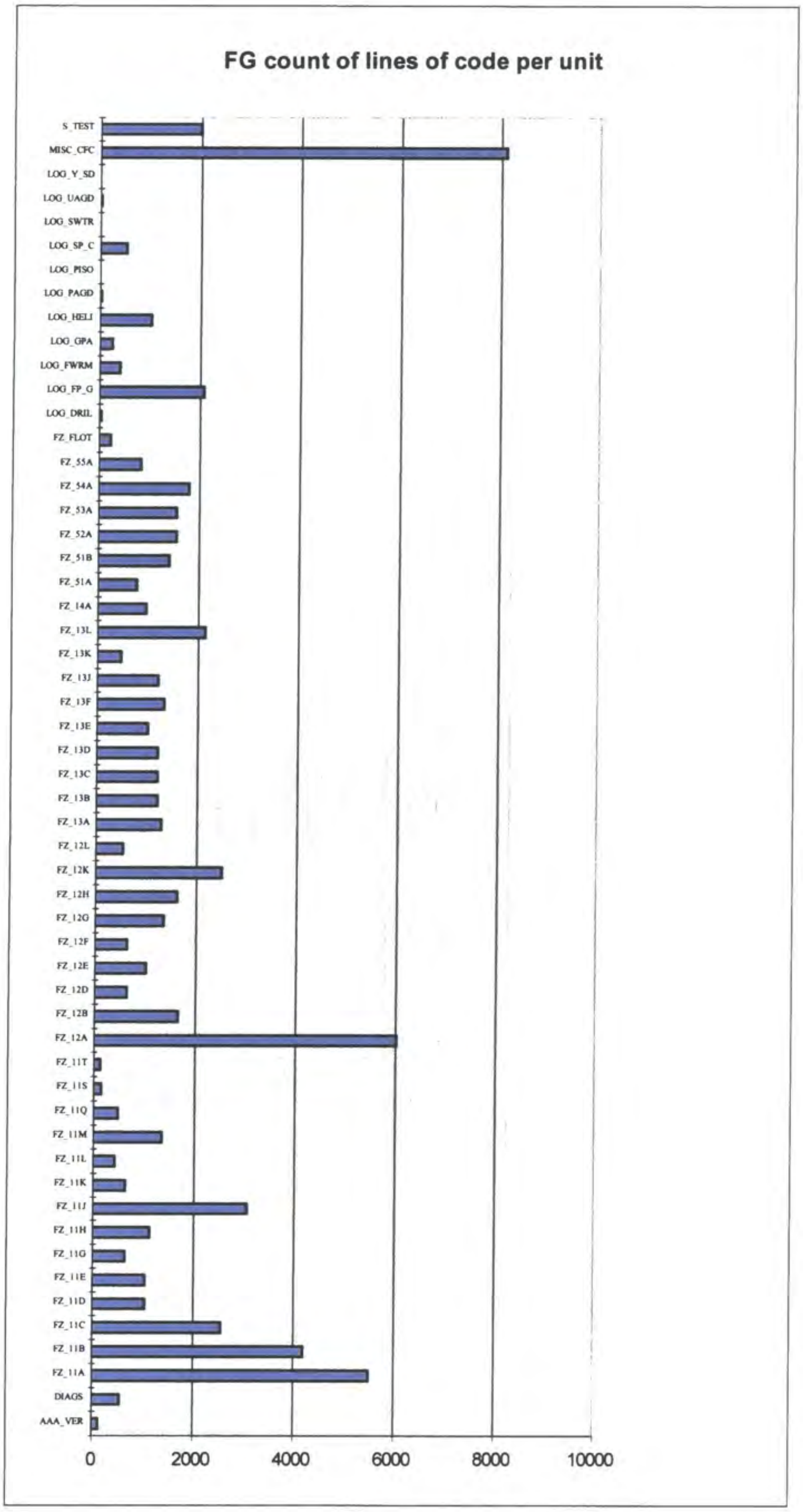


Figure 4:10
Graph showing the number of lines of code in each unit in the F&G program

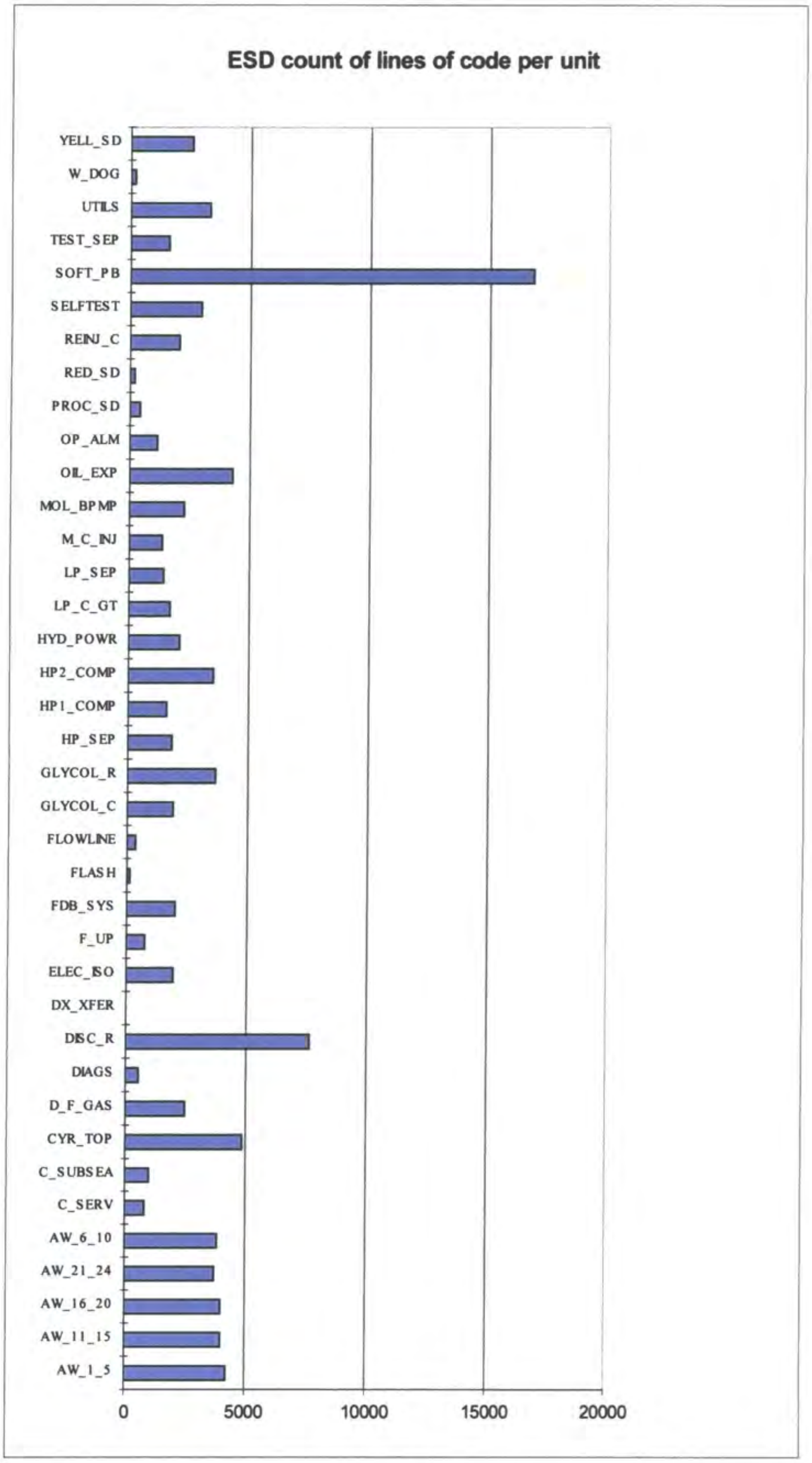


Figure 4:11
Graph showing the number of lines of code in each unit of the ESD program

Data about the number of lines of code in each of the CFB is given in Figure 4:12. This was interesting as much of the start code was taken up by comments. The majority of the code is less than 100 lines long which is in keeping with the suggestions in IEC 1508. [8]

	ESD	**F&G**
Maximum length of CFB	373	344
Minimum length of CFB	6	8
Average length of CFBs	49	41
no. CFBs 1→ 50 lines	59	732
no. CFBs 1 → 100 lines	1902	929
no. CFBs 100 → 200 lines	18	119
no. CFBs 200 → 300 lines	39	9
no. CFBs over 300 lines	2	2

Figure 4:12
Table showing the average number of lines of code in the CFBs

4.1 NESTED STATEMENTS

The code is mainly conditional statements many of which are nested. The maximum level of nested conditional statements is four in the F&G code and three in the ESD code (see Figure 4:13). A high proportion of the code has only one level of nesting although in the F&G code a third of the CFBs have up to four levels of nesting (see Figure 4:14).

Nest Level	**ESD**	**F&G**
0	43	51
1	1367	856
2	554	192
3	26	-
4	-	692

Figure 4:13
Table showing the level of nesting of conditional statements in the code

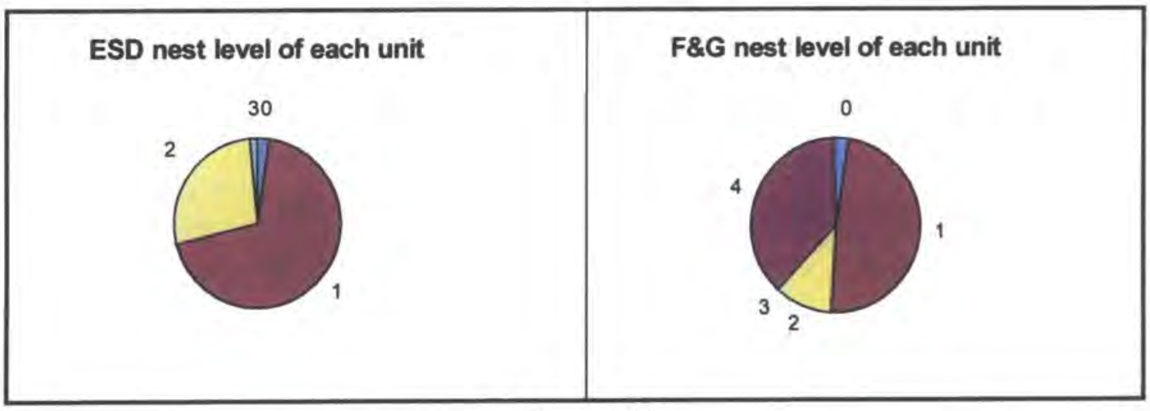


Figure 4:14
Chart showing the maximum level of nested conditional statements in each CFB

4.2 CODE MISUSE

It was noted that in some instances Booleans were assigned to by using the command defined for assigning to flags. This was identified as the translator had to be changed to succeed in parsing the code. In the user manual it is declared that a Digital flag can be used anywhere that a Digital output could be used. It would have to be assumed therefore that a Boolean can be used where a flag can be used.

4.3 VARIABLE USAGE

Most variables were declared as global variables in both samples of code. It was expected that all variables declared would be written to and read except inputs which should be read only and outputs should be write only. This was not the case, many types of variables were not read or not written and some were even not used. During translation it was not possible to identify variables that had been declared as constants so this accounts for some of the identified read only variables.

The graphs Figure 4:15 to 4:17 below demonstrate how the declared variables were used. They are divided by type and then the percentage that were not used, not read, not written, and all variables were represented graphically.

4.3.1 ESD Variables

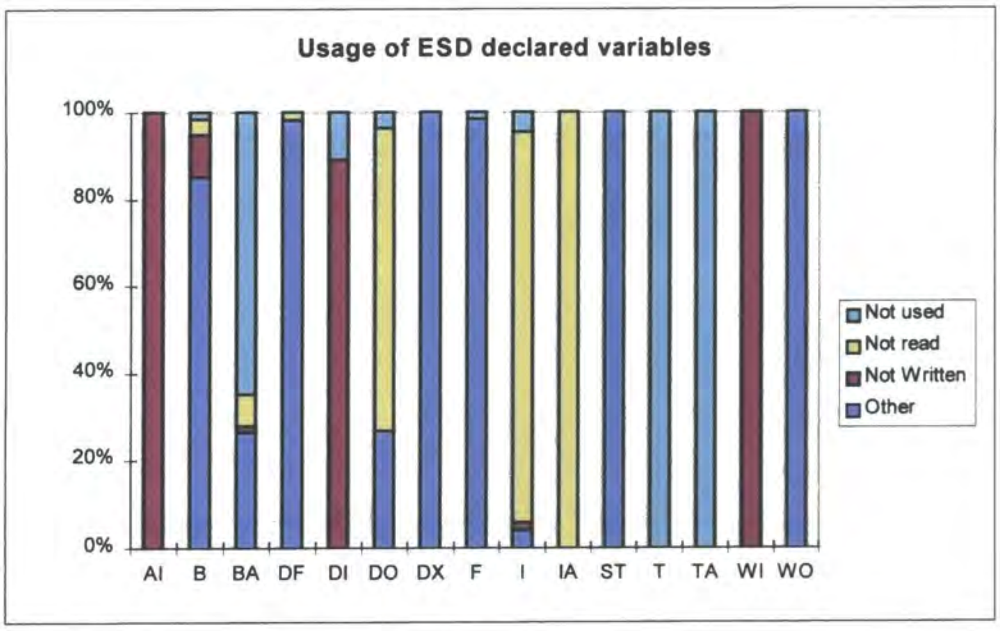


Figure 4:15
Graph showing variable usage for ESD global variables

In Figure 4:15 the inputs were all read only or they were not used. Many of the outputs were read and written to. None of the declared text and text array variables were used although text was indicated in the C&E charts.

A high proportion of the integers were not read, while over half of the BA(Boolean arrays) were not used. Some of the rest of the variables were read only, write only or not used.

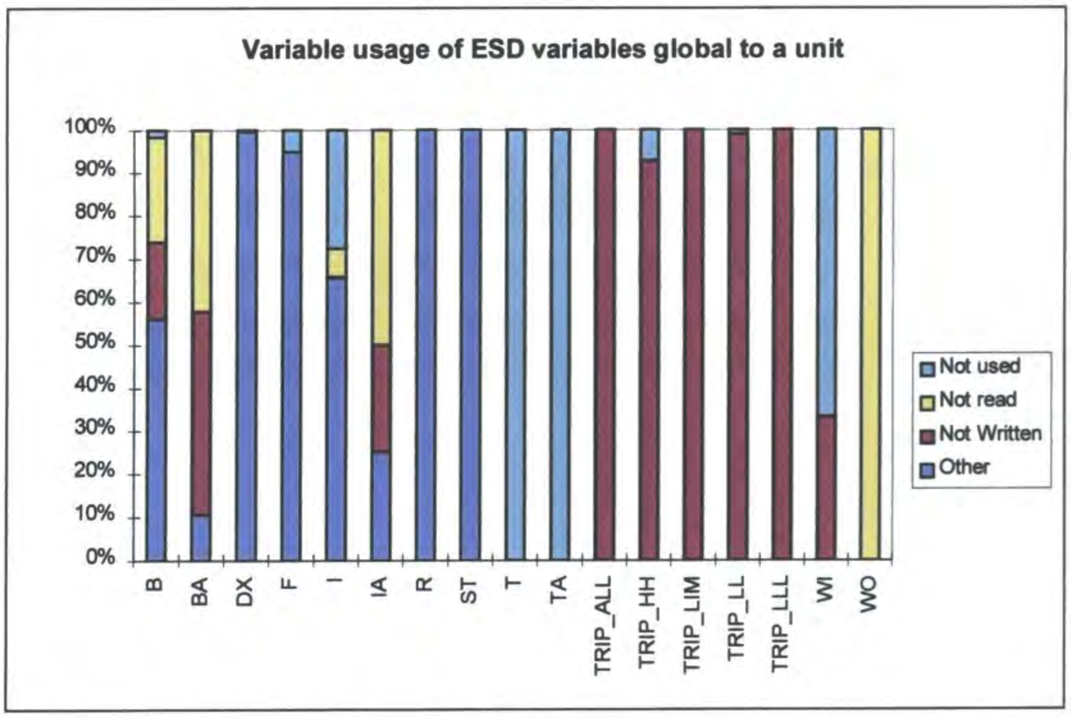


Figure 4:16
ESD usage of variables declared global to units

Figure demonstrates that most of the recipes were read only, this could either mean that they were used as constants or incorrectly. The word inputs were read only or not used. The word outputs were all write only. Again the text and text array variables were not used. The reals, slow timers and DX (DO10 Arrays) seemed to have been used as expected, while the rest were read only, write only or not used.

4.3.2 F&G Variables

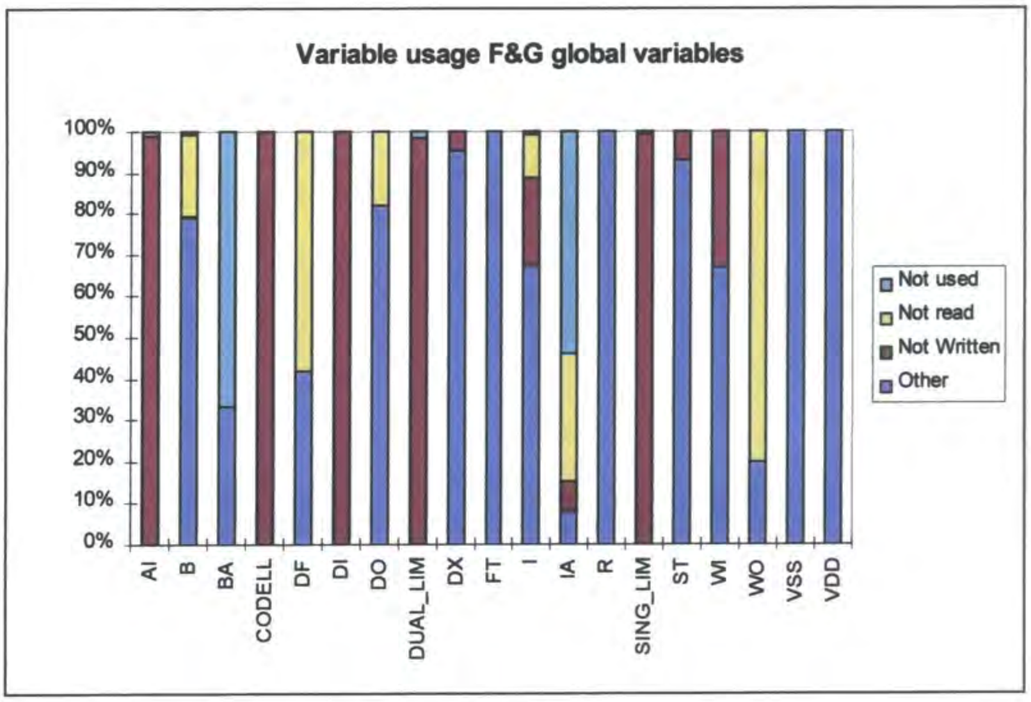


Figure 4:17
Graph showing variable usage for F&G global variables

Error! Reference source not found.4:12 demonstrates that the recipes were read only or not used which again suggests that they might have been used as constants. All of the inputs were read only, while a high proportion of the output variables were read and written. The rest of the variables were read only, write only, not used, and used as expected. There were 35 Booleans declared global to one unit, of these 3 were read only.

4.4 TIME

The time facility was used in both samples of code to obtain the hour of the day. Twice a day at a specific time the SFC was executed. The SFC contained the code for checking the hardware and software were still operational.

4.5 SFC

The SFC was used to perform hardware and software checking on the oil platform. The translation was complicated due to the options at the end of each step, which transition to follow or for the step to remain active. When the safe SFC became active the fact that the active step in the main SFC remained active could not be mapped in WSL. This was not an analysis problem as activation of the safe SFC was determined by a CFB and not an input or during SFC execution.

The translation of the SFC was into an action system which contained GOTO jumps within conditionals, which in turn were transformed into a nested do loop. (see Figure 4:18) The other option would have been for each step to be converted to an until loop with a break out to the safe SFC. This would however have then risked loosing the functionality of the APT SFC because control could not be returned to the main SFC in a different step. Results highlighted it was easier to understand the graphical layout of the SFC (Figure 3:3 and Figure 3:4) than the WSL code in either form of transformation.

```
do do
        if (%action = mi1)
        if      Action block
                set %action to block to call next
        fi      exit(1)
        elsif (%action = mi2)
        .
        .
        .
        .
        fi
od od
```

Figure 4:18
Action system translated into a nested do loop

The F&G SFC was checked to determine if an infinite loop could occur in one of the steps. This was not possible as the transition condition was set at the end of each step, either a Boolean to true, or a timer which after a set time would become true.

4.6 CONTROL FLOW OF THE CODE

The control flow of the code is relatively simple. The control flow diagrams of the F&G code are located in Figure 4:20 (the CFCs and CFBs) and Figure 4:21 (the SFC). Figure 4:19 is the key to Figure 4:20. The ESD had a slightly more complicated control flow diagram because it had to be divided into 3 parts, 'high interlocks', 'low interlocks' and 'active mathblocks'. The F&G code mainly consisted of 'high interlocks' and two CFBs that were of type 'active mathblocks'.

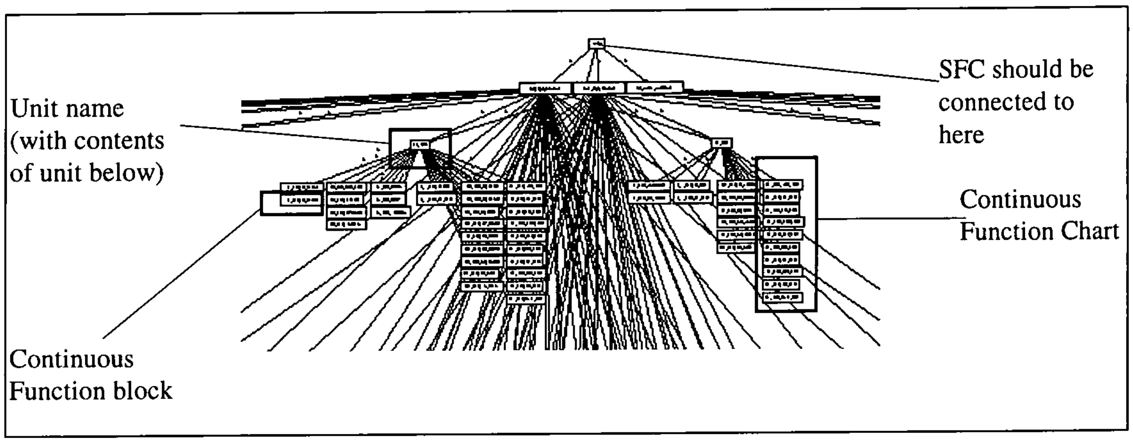


Figure 4:19
Key to (Figure 4:20) and the connection to Figure 4:21
(enlargement at the top at the centre of the picture)

Figure 4:20 gives the overview of the program and is too small to be able to read the detail. The main part is at the top and calls the high priority interlocks, then the SFC and then the two active mathblocks. The diagram is divided into groups of units, the unit name at the top and the columns underneath represent a CFC. Each of the blocks in the CFC represent a CFB. The unit that is considerably larger than the rest is the one that contains the two active mathblocks.

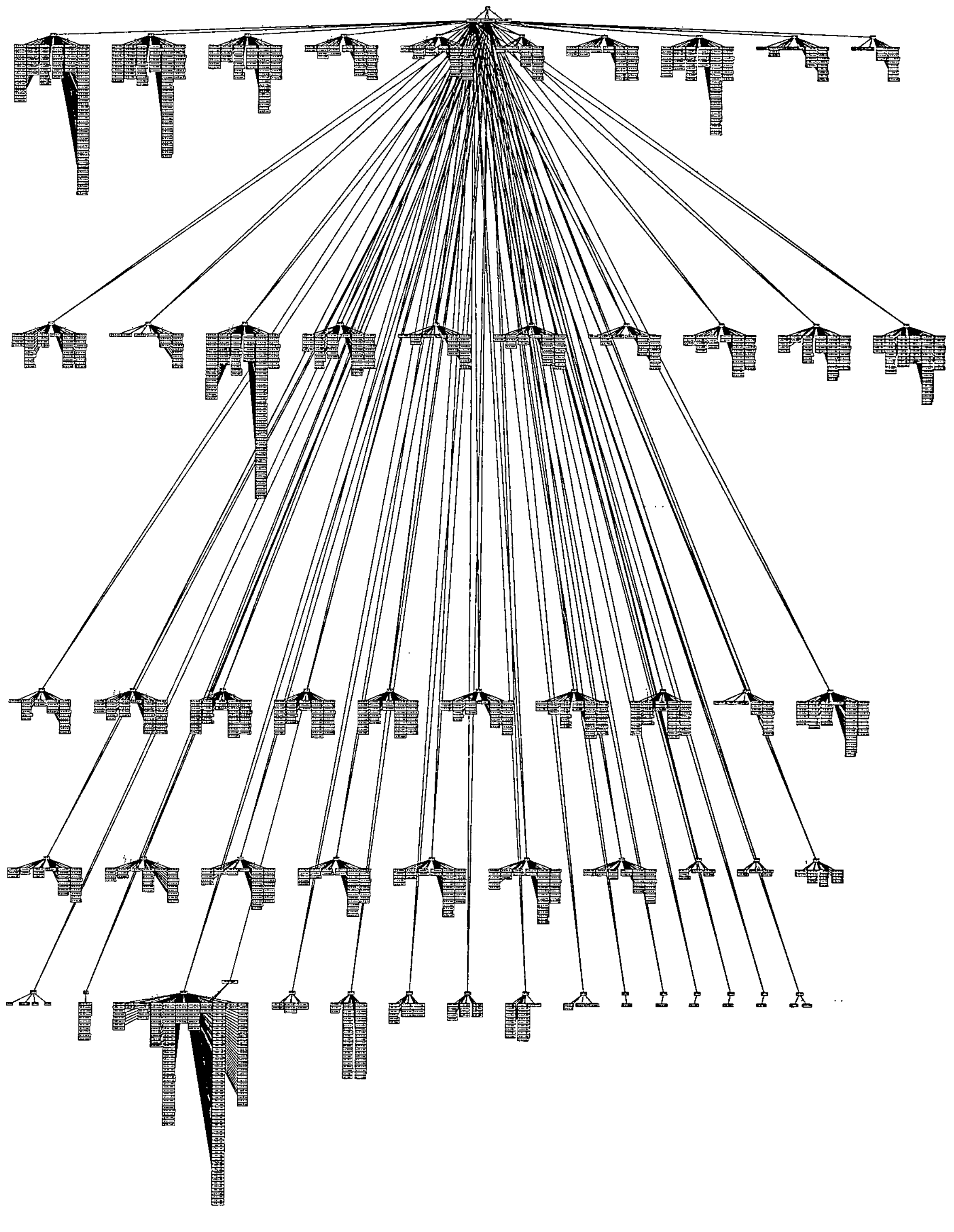


Figure 4:20
Control flow diagram of the F&G program

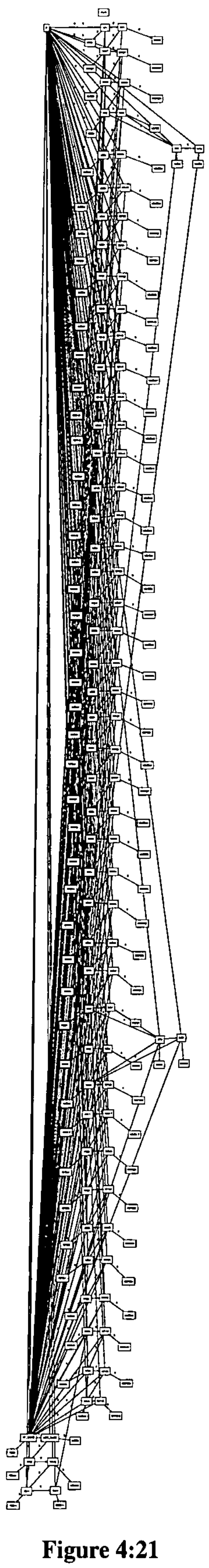


Figure 4:21
Control flow diagram of the F&G SFC

In Figure 4:21 each box is either the initial or the main part of a step, while the lines are all the possible transitions between the steps. Including the jumps to the safe SFC which is found in the bottom left hand corner. Notice how the return from the safe SFC is to a specific step in the main SFC.

4.7 SUMMARY

The ESD and F&G code consisted of units which were programmed mainly using CFCs with a main SFC and a safe SFC in the self testing unit. The code is predominately ' conditional statements' many of which are nested, assignments and procedure/ function calls. There is one while loop in the ESD code and none in the F&G code. The code consists of many variables of 17 different types (valves were not used in the ESD code). Most of the variables that are used are global, most of the ESD units contain variables that are global to only one unit. While only one F&G unit declares variables that are global to it.

5. ANALYSIS

The main reason for translating the PLC code as described in chapter 3 and identifying the characteristics of the code as described in chapter 4 was to analyse the code. IEC 1508 is a draft standard that will be used when building safety critical programmable electrical systems(PES). IEC 1508 identifies highly recommended (HR) techniques that could be used when developing software to each of the four SI levels. Techniques that will aid in the development of a specific type of PES for a specific task may not necessarily aid in the development of a different type of system. The analysis was performed on a subset of the HR techniques defined for SIL 1,2 or 3 to determine if they would be beneficial when developing code for ESD or F&G Offshore Oil Platform applications using PLC languages. Techniques that were highly recommended only for producing SIL 4 software were not analysed, as it is felt by the HSE that software systems should not be built to that reliability and that too much dependence would be put on them. A subset of the HR techniques was studied as it was not possible to analyse all of the techniques using only the data available. The criteria for choosing a technique was whether it was possible to analyse it using the data available. All the analysis was static as it was not feasible to perform dynamic analysis on the code.

Each technique was analysed individually in a similar manner. So the remainder of this chapter deals with each technique individually. A summary of the final results is given at the end of the chapter. The aim was (subjectively) to describe for each technique:-

- if it had been used
- if it could have been usefully used
- if it could not have been usefully used due to the programming environment
- if it could not have been usefully used due to the application

This information could then theoretically be used when deciding development techniques for the development of other ESD and F&G PLC systems on Offshore Oil Platforms. The analysis was performed using a GQM (Goal, Question, Metric) approach, identified by Basil and Rombach [49]. The theory was to identify a goal; the goal in each case was to use the technique for building the code sometimes with a reason if supplied by IEC 1508. Basil and Rombach then state that if the relevant questions are asked the metrics can be defined based on how to obtain the answers to the questions. The questions that were asked in each case were:-

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

In some cases it was not feasible to identify quantitative metrics so qualitative metrics were set. Each technique is discussed under the following headings:-

- Goal
- Definition of technique
- Questions
- Metrics (and the analysis of each metric)
- Conclusions

They were discussed in the following order:-

Those addressing coding standards

5.1 Coding Standards
5.2 Limit the use of pointers
5.3 Limit the Use of Recursion
5.4 No dynamic objects or variables
5.5 No unconditional jumps
5.6 Limit the use of interrupts
5.7 Limit the size of modules
5.8 Use information hiding / encapsulation
5.9 Use verified modules

Those addressing programming languages

5.10 Use a strongly typed programming language

5.11 Use a safe subset of the programming language

5.12 Different programming languages used

- 5.13 CFCs
- 5.14 SFCs
- 5.15 Math Language
- 5.16 APT tool

5.17 Design easily analysable programs

Those addressing analysis techniques

5.18 Use data flow analysis

5.19 Use control flow analysis

5.20 Use structured based testing

5.21 Use FMECA

5.22 Use software fault tree analysis

Definitions of the technique are from IEC 1508 unless otherwise stated. [8]

5.1 CODING STANDARDS

5.1.1 Goal

Use coding standards to ensure a uniform design of documents and code, to enforce egoless programming.

5.1.2 Definition of Technique

The minimum rules that should be adhered to should be defined. A definition of the coding standards should be applied which include modularisation and encapsulation (if using OO (object oriented) programming).

5.1.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.1.4 Metrics

5.1.4.1 What layout standards can be identified in the ESD and F&G code before translation?

Both programs have comments at the top of each block of code; the ESD comments tend to be longer. They both have comments on each line with assignments and other statements. Indentation was used in both programs, the F&G indentation was stricter and all indents are 5 spaces, the ESD indentation is less structured so the F&G code layout looks neater.

There is a variable naming convention that is followed by both programs that include the following and much more:-

- Recipes end with a _R
- Valves or digital inputs end with a _D
- Variables that are connected have the same beginning with a different last 2 letters.

5.1.4.2 What standards could be identified after translation of the code?

Much of the translated code looks very similar; there are pieces of code where the layout of many of the procedures is almost identical and only the names of the variables are different. This would suggest that the layout pattern, style, and coding had been reused, with variables that performed a similar function.

5.1.4.3 Were nested conditional statements allowed and if so to what level?

Nested conditional statements were used in both pieces of code. The level of nesting that was used in the ESD code was less than that of the F&G code. Figure 4:14 is a chart showing the level of if nesting. When transformations were performed on the nested conditional statements, they were difficult to remove, and it did not aid the understanding of the code. The nested conditional statements tended to be dependent on only one or two conditions. Lengthy conditions tended to be located in non nested statements.

5.1.4.4 What is the size of procedures?

The size of the various procedures is discussed in 5.7, but they tended to be of manageable length.

5.1.4.5 Is the timer usage consistent?

Timers are used consistently through out the code. In both the ESD and the F&G program timers are set by using the 'delay' command in the SFCs and set by changing their values in the CFCs.

5.1.5 Conclusions

It is obvious that coding standards have been used. The F&G level of nesting was immediately identifiable. The convention of having related variables with a similar name was beneficial during the understanding of the code. Being able to immediately identify the type of a variable and what it is connected to is also beneficial.

Standards can be used when developing ESD and F&G code on Offshore Platforms. The safety should be increased because it improves understandability and readability, especially during analysis or modification.

5.2 LIMIT THE USE OF POINTERS

5.2.1 Goal

Limit the use of pointers to allow ease of development, verification, assessment and maintenance.

5.2.2 Definition of Technique

"A pointer is a data item that specifies the location of another data item." [43] They can be used for example in linked lists.

5.2.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.2.4 Metrics

5.2.4.1 Where have pointers been used?

Using pointers is not allowed in the programming language. Hardware addresses can be read or written directly as desired.

5.2.4.2 Where has the hardware been directly referenced by address?

It is possible using the APT to address specific parts of the hardware; this is identified by a percentage sign followed by the type and then hardware address. In the F&G code this has been used in two CFBs, in one CFC they contained where identical individual lines of code have been used to write directly to 'status words'. The status word locations have not been given a variable name. All references to status words occur in just one unit. When this was commented upon, writers of the program declared they were reading from the hardware address.

The ESD has the same two blocks of code in the same named unit, CFC and CFB. The ESD code though also has other references to 'status words' in a CFB in a different unit. One CFB also has approximately 280 assignments to hardware values that are of type Boolean and they are being assigned the value of a 'flag'. The hardware locations could not be correlated to the hardware values of any of the variables. There were also instances identified where variables with different names, scope and in some cases type were assigned the same hardware address. This technique is known as aliasing and is considered an unsafe practice.

5.2.5 Conclusions

Pointers cannot be used with the APT system which should have improved the ease of development, verification, assessment and maintenance. Direct addressing of hardware is still allowed although it should be possible to remove it and declare the variables with their hardware address. By declaring variables instead of writing directly to the hardware valuable information may be lost i.e. the maintainer need not necessarily realise the address of the variable.

The absence of pointers is a well-accepted safety feature as it removes instances of multi referencing, dynamic memory allocation. Pointers also tend to lead to confusion as to what is assigned what value. Variables though have been allowed to alias the same hardware address which will introduce a similar confusion.

5.3 LIMIT THE USE OF RECURSION

5.3.1 Goal

Limit the use of recursion to allow ease of development, verification, assessment and maintenance.

5.3.2 Definition of Technique

Recursion is defined as either "1) A process in a software module calls itself or 2) the process of defining or generating a process or data structure in terms of itself."[43] e.g.:-

```
proc A
        {
                call proc A
        }
```

Mutual recursion is defined as A calls B and B calls A.

5.3.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.3.4 Metrics

5.3.4.1 Where has recursion been used and how could it have been better used?

Recursion is not supported by the APT system.

5.3.5 Conclusions

The APT system did not use recursion which implies that the ESD and F&G system can be developed without it. It has been suggested that lack of recursion improves safety as it removes an area of memory management, and space allocation that tends to be error prone. It can cause timing difficulties as during analysis the length of a cycle cannot be calculated.

5.4 NO DYNAMIC OBJECTS OR VARIABLES

5.4.1 Goal

Use coding standards to prevent the use of certain language constructs. Do not use Dynamic objects or Dynamic Variables.

5.4.2 Definition of Technique

Dynamic is "pertaining to an event or process that occurs during computer program execution; for example dynamic analysis, dynamic binding." [43]

A dynamic object allows the type of the parameters to be passed to a functions to be declared at run time. A dynamic variable are variables whose type can be declared at run time.

5.4.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.4.4 Metrics

5.4.4.1 Are dynamic objects used?

Dynamic objects were not supported by the APT system.

5.4.4.2 Are dynamic variables used?

Dynamic variables were not supported by the APT system.

5.4.4.3 How does not using them improve safety features?

By not having dynamic variables and objects the memory management is easier for the compiler and programmer and so less likely to introduce faults. Implicit type casting at run time can also lead to programmer confusion, but this is only used when flag commands were used to assign to Booleans.

5.4.5 Conclusions

Dynamic variables and objects are not supported by the APT and this implies that ESD and F&G code can be written without the use of dynamic objects and variables. Dynamic objects are a well known source of subtle and difficult to find errors which may occur long after initial commissioning. The fact that they are not used should increase safety by improving memory management resulting in a reduction in the number of errors in the executing code.

5.5 NO UNCONDITIONAL JUMPS

5.5.1 Goal

Use no unconditional Jumps to allow an ease of development, verification, assessment and maintenance.

5.5.2 Definition of Technique

An unconditional jump is a "jump that takes place regardless of execution conditions" [43] e.g. GOTO jumps.

5.5.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.5.4 Metrics

5.5.4.1 How and where are GOTO jumps used?

GOTOs were not part of the math language definition.

The SFC graphical language allows transitions from one step to another, which would be a 'GOTO' but not an unconditional jump because the jump was dependent on the transition condition being true.

There is no defined movement between the CFBs in the CFCs. At compile time they are put into a sequential order by the compiler so although there is no ordering defined by the user there is no jumping either since it is a compiler defined order.

5.5.5 Conclusion

GOTOs are not a necessary part of ESD and F&G programming and the APT does not support them. Although the SFC allows GOTOs they are not unstructured jumps. By preventing GOTOs omissions and commissions should be prevented. The code should have a clearer defined structure, which appears to have occurred in the analysed code.

5.6 LIMIT THE USE OF INTERRUPTS

5.6.1 Goal

Limit the use of interrupts to allow ease of development, verification, assessment and maintenance.

5.6.2 Definition of Technique

An interrupt is - "1) the suspension of a process to handle an event external to the process 2) to cause the suspension of a process 3) Loosely an interrupt request." [43]

The aim is to consider the interrupts that are explicitly introduced by the software being programmed and not if there are any interrupts in the operating system, since the operating system is not part of the analysis of this study.

5.6.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

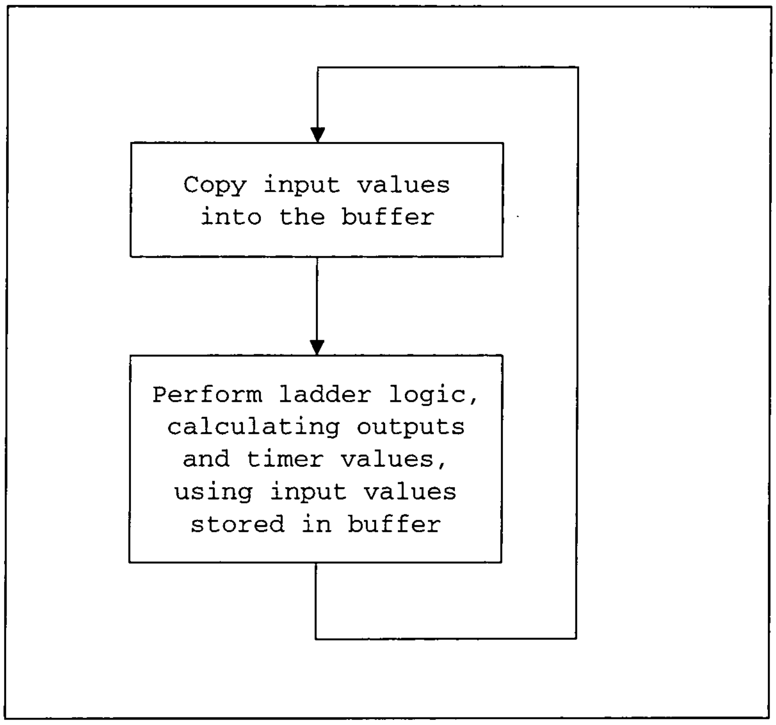
5.6.4 Metrics

5.6.4.1 How are interrupts used?

Hardware interrupts are not used in the code. Software interrupts can be used in the code and are within a defined structure. Execution can move from the main SFC to the safe SFC if the safe transition condition is true. Both the ESD and F&G code were written so the transition condition was set in a CFB and not in the SFC. If the transition condition is false at the beginning of the SFC it is false for the whole SFC. During the end steps of the SFC the safe transition condition is set to false so in future iterations the safe SFC would only be called if the variable was reset to true. This removed all forms of interrupts.

5.6.4.2 Do values change during an execution?

The value of all the inputs is read into the buffer at the beginning of each loop. Even if the input values change this is not made known until the next loop through the code. This is a hardware design implemented by the controller that prevents inconsistent input readings throughout the code. Note that all the internal variables change when written to including the values of the timers. Timers are background processes and can time out during one iteration of the code. By reading inputs into a buffer they need not interrupt the code when they change. At set points during execution there are checks that a variable value has changed before execution continues; especially in the SFCs.



Copy input values into the buffer
Perform ladder logic, calculating outputs and timer values, using input values stored in buffer

Figure 5:22
Diagram demonstrating how the inputs are used by the PLC

5.6.4.3 How does not using interrupts improve verification, assessment and maintenance?

The absence of interrupts means that the code executes sequentially. There is consistency through the code and all the code will be executed at least once before any part is executed a second time. The code when it is compiled does not remain in the same order. This means that the order cannot be tested but by removing interrupts it is known that every block of code will be executed and not starved.

5.6.5 Conclusions

ESD and F&G code do not need interrupts as demonstrated by the sample code although the facility is available in the SFC. By keeping the ordering consistent the tasks should run on time as no task is going to be starved. Consistent input readings will mean all the code reacts to the same values. It also forces the designers to a conservative style of real time design. This illustrates a benefit of PLC code; scheduling real time constraints can be programmed without the use of interrupts.

5.7 LIMIT THE SIZE OF MODULES

5.7.1 Goal

Use limited size modules in order to minimise the complexity of a system by:-

- limiting the parameter number
- only having one exit and entry point
- having a fully defined interface

5.7.2 Definition of Technique

A module is "2) A logically separate part of a program" [43]. A module should have a well defined task or function to fulfil. This provides coherence within the module (coherence in a module should be strong). The connections between the modules should be limited and strictly defined.

Modules should communicate via interfaces; the global variables that are used should be well structured and their access controlled and their use justified. All interfaces should be well documented. An interface should contain the minimum number of parameters for the necessary functions; this is typically five.

Sub programs should be built providing several levels of modules, of which the sizes should be restricted to two to four screens. Each module should have a single entry and exit point and the modules should hide information from the enviroment.

5.7.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.7.4 Metrics

5.7.4.1 Is the function of the modules well defined?

If a module is taken to be a unit then it is defined to be a page of a C&E chart; each of which relates either to a function (e.g. Red Shut down) in the ESD code or a platform area (e.g. fire gas zone 11A) in the F&G code. The CFB in the CFCs generally relate to the manipulation of one or more variables. The CFB will then be the name of the variable that is to be set. To this effect the function of the modules in the CFCs is well defined. The function of the SFC is to perform the self checking routines and to sequentially check that they have been carried out.

5.7.4.2 What is the communication between modules?

The communication between units is via globally declared variables which are known as intertrips. The majority of the variables that are declared are global. There is also no communication within units as the CFBs and SFCs are not directly called. None of the tested variables were used in all of the units. (See 5.8 for the analysis of when, and where and how the variables were used.) The SFC calls one step to be executed then the next; they are always declared in the specified order and do not return to the calling step but move down the diagram.

There is no interface between user defined functions and procedures, so no parameters passed. There are APT defined functions and procedures which were used.

5.7.4.3 What is the size of the modules (equated to math blocks)?

The size of the units vary, see Figure 4:6 to Figure 4:11 for the number of CFB and CFCs located in each of the units. The number of lines in each of the CFBs was calculated including comments. There are 50 lines of code within four screens.

In the ESD program only 53 CFBs were shorter than 50 lines. In the F&G program there were 732 CFBs that were shorter than 50 lines of code. The ESD code had 2020 CFBs while in the F&G code there were 1791 CFBs. An interesting fact though was that the average length (lines of code including comments) was:-

ESD	F&G
49	41

Figure 5:1
Table showing the average number of lines of code

This demonstrates that the average length of a CFB was less than the recommended four screen lengths. Although most of the CFBs, as demonstrated in Figure 4:12 were greater in length than the IEC 1508 recommended four screens. This indicates that a large number of the CFBs must have neen very short hence less than the recommended two screens length.

The SFC diagram in the F&G code covers 3 pages while the ESD diagram is on two pages. This is within the recommended length and since it was pictorial it was easier to understand than four screens of code. (See Figure 3:3 and Figure 3:4) Each step contained textural code, so all the information was not provided by the picture.

5.7.4.4 Does every module have only one entrance and exit point?

The CFCs do not have a single entry and exit point because the CFBs are compiled into an arbitrary order; it is not even necessary that the CFBs within the chart will remain together. After compile time there will be one exit and entry point for the entire piece of code written within a loop.

The CFBs have two entry and two exit points; on the first iteration of the code the entry point is the declaration section, the 'INIT' part of the code is then executed and the exit point is then before the 'BODY' part of the code. On future iterations the entry point is at the 'BODY' and the exit point is at the end of the block.

The SFC has only one entry point into the main SFC and the safe SFC. There can be many exit points from the main SFC; in the F&G program there are two while in the ESD program there are four. The safe SFC is called from any of the steps in the SFC and it returns to a specific step in the main SFC which is not necessarily the calling step. This means that after execution of a safe SFC the re-entry point to the main SFC is known, and as such allows SFC steps to be skipped.

5.7.5 Conclusions

The APT supports the code being divided into modules, of a relatively small size that map to the environmental conditions. This technique has been successfully used and as such should aid safety as the code is not a large single document but one of manageable sized chunks for analysis.

5.8 USE INFORMATION HIDING / ENCAPSULATION

5.8.1 Goal

Use information hiding / encapsulation to increase the readability and maintainability of the software.

5.8.2 Definition of Technique

Information hiding is "a software development technique in which each module's interfaces reveal as little as possible about the module's inter workings and other modules are prevented from using information about the module that is not in the module's interface specification." [43]

Encapsulation is "a software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module." [43]

Data that can be globally accessed can be incorrectly changed, and all the code would then have to be revalidated. Information hiding reduces these difficulties if one module has to be changed then only one module has to be re validated.

5.8.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.8.4 Metrics

5.8.4.1 Is information hiding supported by APT?

Information hiding is not supported in the conventional sense by the APT. It allows the code to be divided into CFCs and CFBs but there is no formal interface between them. All information passed around the program is via global variables.

5.8.4.2 How many units are global variables normally used in?

One in ten variables were analysed to provide the following data. The results listed below show the number of units that each variable was used, read and written in. The ESD sample variables demonstrated more clearly that the variables were read in one unit and written in another.

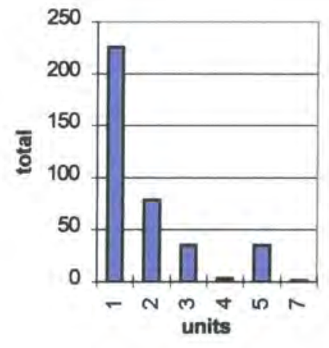
It should be noted that the F&G system does not have unit defined global variables. The variables sampled in the F&G code were used in a maximum of 3 units (see Figure 5:5), while in the ESD code the variables sampled were used in up to 7 units (see Figure 5:2).

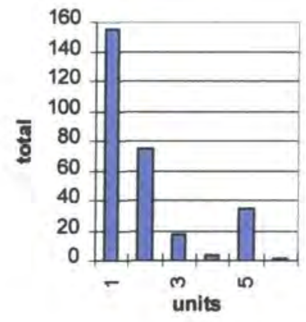
It was interesting to note that some units contained local variables of the same name. Some of the local variables also had the same name as global variables.

5.8.4.3 Are global variables read and written in multiple units?

Of the variables sampled there were instances of variables being both read and written to in more than one unit. The summary of the number of units that global variables were read and written in is provided in Figure 5:2 to Figure 5:7. The data is slightly inaccurate as it was taken from the translated code and flags in the translated code were assigned to in every block that they were used in, as that was seen as the best method of mapping the On function.

ESD code





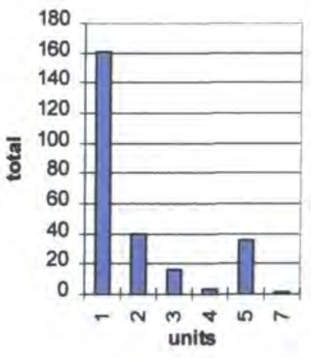
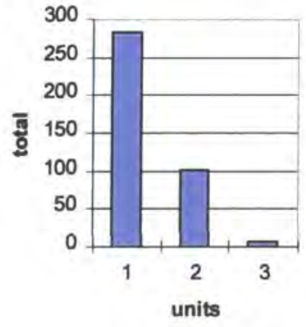


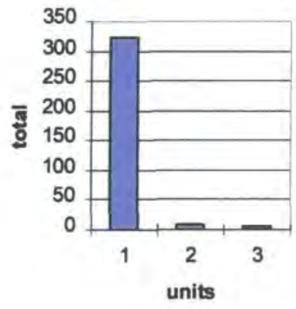
Figure 5:2
ESD number of units each variable was read or written in

Figure 5:3
ESD number of units each variable was read in

Figure 5:4
ESD number of units each variable was written in

F&G code





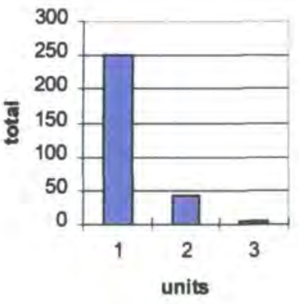


Figure 5:5
F&G number of units each variable was read or written in

Figure 5:6
F&G number of units each variable was read in

Figure 5:7
F&G number of units each variable was written in

The ESD code demonstrated a number of instances where variables were read in one unit and written to in a different unit. This is demonstrated in Figure 5:2 to Figure 5:4. The data also demonstrated that variables tended to be written to in one or two units, often read in more and in many cases different units. This information has not been supported by software fault tree analysis which often indicates internal variables (not outputs) only being written to in one block.

The F&G code demonstrated a large number of variables that were read in one unit and written to in two units. The unit in which the variable was read was normally one of the ones it was written to. There are also a number of instances where the variables are read in one unit and written to in a different unit. It was not analysed how many of the CFBs within a unit the variables were used in. The F&G variables sampled were used in a maximum of 3 units. See Figure 5:5 to Figure 5:7.

5.8.5 Conclusions

Information hiding and encapsulation was not an option available with the APT. A greater level of programming discipline was therefore required to not use variables in all of the units. Of the 392 F&G variables tested 283 were used in only one unit and the rest in a maximum of 3. In ESD, 377 global variables were sampled and 225 of them were used in only 1 unit but the rest were used in up to 7 units.

Encapsulation aids safety as it reduces the chances of read write conflicts between units but it was not available in the APT system.

5.9 USE VERIFIED MODULES

5.9.1 Goal

Use a library of trusted or verified modules to avoid continually having to revalidate code

5.9.2 Definition of Technique

Well defined PES (Programmable Electrical Systems) consist of hardware and software components and modules that are distinct and interact in a clearly defined way. In many PES there are parts of the code that can be reused, which requires less revalidation.

5.9.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.9.4 Metrics

5.9.4.1 Was code reused?

It is believed that all the code was written from scratch, as it is the first time that the PLC software was used. The SFC layout, structure and design were similar in the ESD and F&G code. They both had the same sort of language constructs and they both mainly used interlocks and similar sorts of functions. They both used SFCs in the self test units.

5.9.4.2 Was code written so it could be reused?

The code was written in a style that could potentially be reused. There was a self test unit in both pieces of code developed in a similar style. The units were divided partly in terms of plant layout and partly in terms of function. Other platforms may have similar designs but each of them is unique. Also within the CFBs much of the code looked identical with just 1 or 2 variables replaced. This suggests the existence of informal or formal coding standards.

5.9.5 Conclusion

It is not easy to develop for reuse, because there is no method of encapsulating code to perform just one function. Theoretically reuse is possible. It is hindered by the dependence on global variables but there were definite similarities between the two samples of code. Reuse of previously verified code reduces the need of re-verification but increases the problem of identifying if 2 functions have the same functionality..

5.10 USE A STRONGLY TYPED PROGRAMMING LANGUAGE

5.10.1 Goal

Use a strongly typed programming language which will permit a high level of checking by the compiler to reduce the probability of faults.

5.10.2 Definition of Technique

User defined types can be formed from basic programming types, and strict checks are enforced at compile time to ensure that the correct type is used.

5.10.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.10.4 Metrics

5.10.4.1 Is it a strongly typed language that was used?

The manual[50] identifies the language as being a strongly typed language but it allowed Booleans and flags to be interchanged. It also did not insist on the recommendations that were made within the user manual. (See 5.11.4.2)

5.10.4.2 Were there any instances of code that would not have compiled using a strongly typed language?

Variables that were declared to be of type Booleans were assigned using the command to assign a flag throughout the F&G code. This should not have caused a problem because flags have values of true and false, but they are assigned differently from Booleans.

Unit	Boolean
FZ_11B	WSL_XS_20201
FZ_11C	WSL_XS_20215
FZ_11J	WSL_XS_20217
FZ_11J	WSL_XS_20218
FZ_11J	WSL_XS_20263
FZ_11M	WSL_XS_20202
FZ_12G	WSL_XS_20213
FZ_12H	WSL_XS_20260
FZ_13F	WSL_XS_20214
FZ_13L	WSL_XS_20206
FZ_13L	WSL_XS_20259
FZ_51A	WSL_XS_20212
LOG_PAGD	WSL_XS_20261
LOG_PISO	WSL_XS_20209
LOG_UAGD	WSL_XS_20262
LOG_Y_SD	WSL_XS_20203

Figure 5:8
Table showing the Booleans that are assigned as flags

Had the language been a strongly typed language the above Booleans would not have been able to be assigned the value of a flag.

5.10.4.3 Would a strongly typed language have aided maintenance by preventing erroneous code being included?

In the math language timers can be started using the DELAY command or by setting the values of the timer. Both options could be used on the same timer but this is not recommended by the manual[51] - only one should be used. Two methods would make it more difficult during maintenance to identify the method used with individual timers. With the code supplied the SFCs used the DELAY command and the CFBs did not use the DELAY command. Turning an individual timer on by two different methods was not used.

5.10.4.4 Were variables declared and not used?

Some global variables were declared and not used. A language that identified these may have proved useful as it may have identified code that had not been written. In the case of the ESD code all the text variables that were declared were not used but the C&E charts still identify textural output. The text variables had the same hardware address as an integer array; it is believed that the array was used to copy the text values into another integer array. Even digital inputs and outputs had been declared and not used; so were pieces of hardware made redundant or was the code omitted? The F&G system had 2 out of the 3 Word inputs that were declared but not used.

ESD		F&G	
Type	**Quantity**	**Type**	**Quantity**
B	14	AI	2
BA	44	B	12
DI	148	BA	32
DO	25	DUAL_LIM	1
F	7	I	11
I	12	IA	7
T	59	SING_LIM	1
TA	1	VSS	1
		WI	2

Figure 5:9
Table showing types and quantities of variables not used

The F&G system had less variables declared globally to units, this might have been the reason that all of the variables declared at unit level were used. The ESD system though did have variables declared at the unit level that were not used. All the text variables declared at the unit level were also not used see Figure 5:10.

Type	Quantity
B	57
F	34
I	243
T	1000
IA	1
TRIP_HH	4
TRIP_LL	1
WI	4

Figure 5:10
Table showing types and quantities of ESD unit variables that were not used

5.10.5 Conclusions

Strongly typed languages may not have allowed any of the above to occur; the Booleans would have to have been declared as flags. Allowing variables of different types to have the same hardware address seems to mean that the language was not as strongly typed as it could have been.

Not allowing the above would remove the possibility of ambiguity and omissions but decreases flexibility. Code could potentially be written using a stronger typed language which could help to prevent omissions, detectable and undetectable errors. Enforcing the correct use of timers aids maintenance. A strongly typed programming language which the APT seemed to supply reduced the chances of careless mistakes with similar variable names being allowed unchecked.

5.11 USE A SAFE SUBSET OF THE PROGRAMMING LANGUAGE

5.11.1 Goal

Use a subset of the language to reduce the probability of introducing programming faults and increase the probability of detecting faults.

5.11.2 Definition of Technique

Examine the language to determine the constructs that are error prone or difficult to analyse, especially using static analysis methods. A subset of the language should be defined to exclude these techniques.

5.11.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.11.4 Metrics

5.11.4.1 What is the subset of the language and how do the removed techniques improve safety?

Overview

The top level of the program did not have any SFCs or CFCs; all of the code was in units. The main and safe SFC that was included in both programs was put in the 'SELFTEST' ('S_TEST') unit. It was CFCs that were predominately used in both the ESD and the F&G code.

CFCs

Within each of the CFCs only the interlocks and math blocks were used. All of the math blocks were active so they could not be turned off, and interlocks cannot be turned off. The math blocks were not given any inputs (although this is allowed) and interlocks cannot be given any inputs. Hence none of the connection possibilities between the CFBs were used nor were any of the other CFB types which include:-

- Pieces of hardware types
- Maths functions e.g. divider, summer, subtractor

All of the coding that required interaction of variables was written in the math language.

SFCs

Most of the SFC functionality was used. Parallel execution was not used which is beneficial since it is easy to write invalid programs. Only the local safe SFCs were used and not either of the other two types. Using any safe SFCs would have made the exit conditions more confusing. The F&G SFC did not use any loops although the ESD SFC had one loop. (see Figure 3:3 and Figure 3:4) Also the condition to call the safe SFC was set outside the SFC, within a CFB.

SSABORT command was not used. This stops the execution of all the relevant SFCs. It would mean that it may not have been known where in the cycle the program had reached.

Math language

All of the math language was used apart from the printing. There was only one while loop used. Some of the APT defined procedures and functions were used. No piece of code that could only be compiled to SFPGM was used.

5.11.4.2 Which error prone parts would the author have removed if possible?

In 5.18 the example is cited of a function that according to the manual can only be compiled to RLL and a function that can only be compiled to SFPGM used in the same CFB. Since most of the code was compiled to RLL then the author would probably have excluded all functions and procedures that were compile only to SFPGM to prevent this sort of fault from occurring. This was reported to Siemens and the APT development team issued the following statement:

"The APT Engineering organization has reviewed the manuals and has confirmed that the definition of BITS_TO_INT function is described incorrectly. Page 11-16 of the APT Programming Reference manual states in the first sentence that this function is "available only in SFPGM for Series 505 controllers." This limitation is incorrect. The function is available in both SFPGM and RLL. We have entered DT7265A into our APT Configuration Management and Problem Tracking system to document the incorrect statement in the manual. This Development Task (DT) will be incorporated during the next manual update cycle."

5.11.4.3 Was it possible to tell which were error prone parts?

In some parts of the user manual it was difficult to comprehend what was the meaning of the definitions. These included:-

- When the on command on a flag turned off.
- What parallel execution was - SFC predominately.
- How often the input values were read by the program
- What control errors are
- Whether the parameter types were checked
- Whether there is boundary checking on arrays

5.11.5 Conclusions

It is possible to use a subset of the APT language. The ESD and F&G code did not require all of the available features. Defining a subset should simplify static analysis performed on the code, which in turn should lead to safer code.

Using only a subset of the code improves the understandability of the constructs. The safety that it should provide is the removal of omission and commission, and the analysis should be more likely to identify detectable and undetectable errors.

5.12 DIFFERENT PROGRAMMING LANGUAGES USED

5.12.1 Goal

Use a programming language with a defined subset of language to produce easily verifiable code with the minimum of effort.

5.12.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.12.4 Metrics

This goal is used against the three programming languages used by the APT, CFCs, SFCs and the math language, then the APT system as a whole is studied. The results are in 5.13 to 5.16 respectively.

5.13 CFCS

5.13.1.1 Is the language fully and unambiguously defined?

There is confusion about the order the CFBs are compiled into and whether they remain within the CFCs at compile time. The semantics are also never formally defined.

5.13.1.2 Is the language problem oriented?

The language is problem oriented in as much as it allows a different chart to be built for each piece of hardware, or each function that is required to be performed on the platform.

5.13.1.3 Does the language provide:-

- **Block structure**

  The entire programming language is made up of blocks. The CFC is divided into interconnecting CFB (Continuous Function Blocks); the sub-language used only had distinct CFBs.

- **Run time checking and array boundary checking**

  Arrays are not stored or used directly by the CFCs.
- **Parameter checking**

  No parameters are passed so there is no requirement for parameter checking.

5.13.1.4 Does the language encourage:-

- **Use of small and manageable modules**

  It is possible to have many CFCs in a program, each of which can be divided into CFBs. It is easier to maintain control of the CFC if it is all displayed on one screen. The F&G and ESD code's largest CFC corresponded to 2 screens, which was 50 CFBs.
- **Restriction of access to data in defined modules**

  Restriction of data is not possible since variables cannot be declared as local to an entire chart. Variables can be declared as local to the block which means that information cannot be accessed elsewhere otherwise they are global to a unit or all the code.
- **Definition of variable sub ranges**

  Variables are not declared within the CFC. They are declared as part of one of the other programs.
- **Any other type of error limiting construction**

  The subset of blocks used did not take inputs and outputs and the math blocks were set so that they could be turned off.

5.13.1.5 Features which make verification difficult should be avoided including:-

- **Unconditional jumps (excluding sub routine calls)**

  Each of the blocks are distinct entities and there is no connection between them, apart from the ordering which is decided at compile time.

- **Recursion**

  This is not possible. Each block is executed once before any block is repeated.

- **Pointers, heaps or any type of dynamic variable or object**

  These are not possible.

- **Interrupt handling at source code level**

  This is not supported.

- **Multiple entries or exits of loops, blocks or sub programs**

  The blocks in the chart are distinct and can only be entered at one point, but the blocks of the charts are ordered at compile time and the ordering can be changed each time, so the entry point to the chart effectively changes.

- **Variant records or equivalence**

  There are no records stored in the CFCs

- **Procedural parameters**

  There are no procedures, so procedures cannot be passed as parameters.

5.13.2 Conclusions

Many of the characteristics required for an IEC 1508 language were found in the subset of CFC language that was used with the ESD and F&G code. The possibility of only using a much smaller subset was available, which enabled less confusion and greater rigidity over the characteristics of the code.

5.14 SFCS

5.14.1.1 Is the language fully and unambiguously defined?

It was unclear how parallel execution within the steps worked; also none of the language was formally defined.

5.14.1.2 Is the language problem oriented?

A different chart could be created for different functions or pieces of hardware.

5.14.1.3 Does the language provide:-

- **Block structure**

  The entire programming language is made up of blocks (steps). The chart is made up of steps that are joined together by transactions. This means that the code in each of the blocks can be quite small. In each of the steps there is math language text.
- **Run time checking and array boundary checking**

  Arrays are not stored or used directly by the SFCs.
- **Parameter checking**

  No parameters are passed so there is no requirement for parameter checking.

5.14.1.4 Does the language encourage:-

- **Use of small and manageable modules**

  There can be any number of steps to an upper limit of 500 combined together and even any number of SFCs if necessary. The SFC in the ESD code fitted onto two screens while the F&G SFC fitted onto three screens. Since all the steps are joined by transactions which can be the 'true' transaction the steps can be as small as desired.
- **Restriction of access to data in defined modules**

  Variables cannot be declared as local to an entire chart, variables can be declared as local to the step which means that information cannot be accessed elsewhere.
- **Definition of variable sub ranges**

  Variables are not declared within the SFC they are declared as part of the math language in the step or as part of the programming environment of the APT. So variable sub ranges are not relevant to SFCs.

- **Any other type of error limiting construction**

  Although the language supports parallel processing of steps the sub language did not use parallel processing. The ESD SFC had one loop back to previous steps, but the F&G did not. The sub language did not use SSABORT of the safe step which meant the whole SFC could not be stopped in an unknown state.

  The language encourages code that is divided into sequential blocks and it allows implicit delays where the code loops until the exit transition becomes true. Care has to be taken that the exit transition does become true, as this could potentially be a danger of the language.

5.14.1.5 Features which make verification difficult should be avoided including:-

- **Unconditional jumps (excluding sub routine calls)**

  Jumps are made from one step to the next but they are dependent on the transition condition being true so they are not unconditional jumps.
- **Recursion**

  This is not possible.
- **Pointers, heaps or any type of dynamic variable or object**

  This is not possible.
- **Interrupt handling at source code level**

  This is not supported
- **Multiple entries or exits of loops, blocks or sub programs**

  There is only one entry step in an SFC. There are no multiple entrances to the steps in the SFC. The exit depends on which step is to become active next and hence which transaction is to be followed. The transition to the safe SFC can become true at any time so the active step is postponed. This was programmed so if the safe SFC was going to become active in a pass it occurred as soon as the safe SFC was turned on or not at all. When the safe SFC has finished the active step is stopped and the return step becomes active. There is more than one final step in the SFC. The ESD SFC has four while the F&G SFC only has two.

- **Variant records or equivalence**

  There are no records stored in the SFCs

- **Procedural parameters**

  There are no procedures, so procedures cannot be passed as parameters.

5.14.2 Conclusions

Many of the characteristics required for an IEC 1508 language were found in the subset of the SFC language that was used with the ESD and F&G code. The program proved that a subset of the language could be used. It was possible to create loops in steps, and loops in the SFC that did not exit.

5.15 MATH LANGUAGE

5.15.1.1 Is the language fully and unambiguously defined?

There is ambiguity defining turning an 'on' flag automatically off. The language is not formally defined but defined in natural language with examples in the manual.

5.15.1.2 Is the language problem oriented?

It is similar to Pascal so it is oriented round assignments and conditions.

5.15.1.3 Does the language provide:-

- **Block structure**

  The language does not support a block structure, but the language is expected to be within a block of another language.

- **Run time checking and array boundary checking**

  There is run time checking.

- **Parameter checking**

  Parameters are passed to the math language defined procedures.

5.15.1.4 Does the language encourage:-

- **Use of small and manageable modules**

  The size of a block of math language is dependent on the amount of information that is to be processed in that block and that is dependent on the design of the CFC and the SFC so the actual block size is programmer and designer dependent.

- **Restriction of access to data in defined modules**

  Variables can be declared within the block which means that these variables can only be accessed in the blocks. Most of the variables are declared globally outside the math block in which case the data is accessible to any of the blocks. Only Booleans, integers, reals, timers or arrays can be declared within the block.

- **Definition of variable sub ranges**

  Variables declared within the block cannot be given a sub range although they can be given an initial value.

- **Any other type of error limiting construction**

  The manual does not encourage the use of while loops, and there is only one while loop in the ESD code and none in the F&G code.

The compiler has allowed Booleans to be used as flags in more than one instance in different blocks of code (see 5.10).

5.15.1.5 Features which make verification difficult should be avoided including:-

- **Unconditional jumps (excluding sub routine calls)**

  No unconditional jumps are allowed in the math language code.
- **Recursion**

  This is not possible.
- **Pointers, heaps or any type of dynamic variable or object**

  These are not possible.
- **Interrupt handling at source code level**

  This is not supported
- **Multiple entries or exits of loops, blocks or sub programs**

  Each block is entered at the start and either the initial code or the main body of the code is executed depending on whether it is the first loop of the execution or later ones. The exit is when the last statement of code in either the initial part or the main part is executed.
- **Variant records or equivalence**

  There are no records stored in the CFCs
- **Procedural parameters**

  There are no procedures, so procedures cannot be passed as parameters.

5.15.2 Conclusions

Many of the characteristics required for an IEC 1508 language were found in the subset of Math language that was used with the ESD and F&G code. It was possible to use a subset of the language and avoid all the problem areas identified in IEC1508.

5.16 APT TOOL

5.16.1.1 Is the language fully and unambiguously defined?

All the languages and their interactions are defined apart from the parts cited 5.13 to 5.15. The ordering of the of the entire system is not clearly defined.

5.16.1.2 Is the language problem oriented?

It allows the code to be divided into units with respect to the functional units of the platform.

5.16.1.3 Does the language provide:-

- **Block structure**

  Code can be divided into units which then allows the use of one or more programming language, which themselves are divided into blocks.
- **Parameter checking**

  Parameters are not passed between the various parts of the programs.

5.16.1.4 Does the language encourage:-

- **Use of small and manageable modules**

  The facility is provided to divide the code into units and then to divide each of the units into one or more programs written as either CFCs or SFCs.
- **Restriction of access to data in defined modules**

  Most of the variables declared within the APT are global; in the F&G code the majority are global to the entire program, while with the ESD about half are global to the entire program while the other half are declared as global to the entire contents of one unit.

- **Definition of variable sub ranges**

  Variables cannot be declared to have sub ranges.
- **Any other type of error limiting construction**

5.16.1.5 Features which make verification difficult should be avoided including:-

- **Unconditional jumps (excluding sub routine calls)**

  These are not allowed in any of the languages supported by the APT.
- **Recursion**

  This is not allowed in any of the languages supported by the APT.
- **Interrupt handling at source code level**

  This is not supported
- **Multiple entries or exits of loops, blocks or sub programs**

  The SFC starts at the beginning and finishes at one of the end steps,(see 5.14) The CFCs are reordered at compile time and each CFB is executed in an order in a loop.
- **Variant records or equivalence**

  Recipes are records in which every part has to have a declared type.
- **Procedural parameters**

  User defined procedures (CFCs, CFBs, SFCs) could not be called.

5.16.2 Conclusions

Many of the characteristics required for an IEC 1508 language were found in the subset of languages that were used with the ESD and F&G code. The system allowed safer subsets to be used. There is the potential problem of not knowing the ordering of the system. It is possible to build SFCs that do not terminate. On the whole most of the accepted problem areas are avoided either by the APT or the subset of the language.

5.17 DESIGN EASILY ANALYSABLE PROGRAMS

5.17.1 Goal

Design programs so analysis is easy and feasible and the program is fully testable.

5.17.2 Definition of Technique

Programs should be designed so that they are easy to analyse using static analysis techniques. To do this structured programming methods should be followed - these include:-

- Module control flow should be composed of small structured components
- Modules should be small
- Number of possible paths through a program should be small
- Program parts should be decoupled
- Complex calculations should not be the basis of branching / looping decisions.
- Branch and loop decisions should be related to the module input parameters
- Boundaries between different types of meanings should be simple

5.17.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.17.4 Metrics

5.17.4.1 What is the size of modules and are they small?

If a module is taken to be a unit then size can be calculated by looking at the number of lines of code in the unit, the number of CFCs or the number of CFBs in each unit. The number of lines of code can vary depending on the length of statements, most of which were no longer than a screen width or were moved onto the next line. See graphs Figure 4:6 to Figure 4:11.

The ESD code is divided into 38 units, each containing between 1 and 14 CFCs. Most of them have between 4 and 7 CFCs which could be considered small. The lines of code per unit and the number of CFBs relates directly to the number of CFCs per unit. Only two of the units have over 5000 lines of code and one of them has 17000 lines which is much higher than the others. This same unit has a high number of CFBs and the highest number of CFCs. So this unit could not be called comparatively small.

The F&G code is divided into more units and there are fewer lines of code compared with the ESD code. This mostly suggests that the units are smaller than those in the ESD system. There are 55 units each having at least 1 CFC and at most 11; many of them have 7 CFCs. The number of lines of code in each unit relates directly to the number of CFBs per unit and not strongly correlated the number of CFCs per unit. The maximum lines of code in a unit is 8000 although many of them have under the 2000.

The F&G program has smaller units than the ESD program; both programs have one unit that is considerably larger than the others. Most of the units though are relatively small, without being so small that there would be hundreds of units; which is thought to increase complexity.

5.17.4.2 The number of paths through a program is dependent on branching and loops

The ESD code contains only one while loop which either does not execute, or terminates in 450 cycles or less see Figure 5:28 for the code. The number of paths through a conditional statement is dependent on the level of nesting which is a maximum of 4. The number of 'if', 'else' and 'else if' branches is analysed in 5.20.

The actual number of paths through the entire piece of code is a factorial of the number of mathblocks in the code, since the ordering of the CFBs is unknown. This is because it is determined at compile time and so could be different each time it is compiled; although it probably isn't as there is a system for determining execution order at compile time.

5.17.4.3 Are parts of the program decoupled?

The CFBs and CFCs are not decoupled because they are all dependent on global variables that are dependent on other parts of the program.

5.17.4.4 Where are complex calculations used as the basis of branching and could this be changed?

There are no complex calculations although in some instances the branching is dependent on long combinations of and's and or's. Conditional statements that are multi lines long could be regarded as complex logical expressions. There was limited functionality within a branch if the conditional had been complex. This indicates a value that was dependent on many others, often tracing back to inputs.

5.17.4.5 Branching based on input parameters?

There are no input parameters in the code studied. Therefore the branching conditions cannot be based on input parameters. Some of the math blocks can take inputs but in the code studied they have not been used. Code is easier to analyse if parameter passing is not allowed because the origination of variables does not need to be tracked, or where variables were passed from. Global variables though, which were used within the system, can be just as hard to analyse.

5.17.4.6 Do math blocks use predominately global or local variables?

In the F&G system only one unit has its own set of global variables declared. All the rest use global variables and the variables declared within the CFBs. The F&G code has 3989 global variables while the ESD code has 4413 global variables and 5899 variables that are declared global to only one unit. The ESD math blocks use a mixture of global and unit global variables in the code. Many of the CFBs declare their own local variables.

5.17.5 Conclusions

The code is written in relatively small modules which allow a modest number of paths through each block. There are a large number of paths through the program since the CFB order is determined by the compiler. Variables that are used are declared globally either to a unit or the whole program. The declaration of variables is external to the code within the APT tool.

Input parameters have not been used. The major difficulty of the static analysis would therefore be to prove that the ordering of the CFBs is immaterial. The code that was written had CFC names that were often similar or contained the unit name so most of the CFC could be joined to a unit by their name. This was a great help during static analysis.

5.18 USE DATA FLOW ANALYSIS

5.18.1 Goal

Use data flow analysis to detect poor and potentially incorrect data structures

5.18.2 Definition of Technique

Data flow: " the sequence in which data transfer, use, and transformations are performed during the execution of a computer program". [43]

Combines information from control flow analysis with information about which variables are read or written in different parts of the code. Variables that are important to identify are those that are:-

- written more than once without being read - omitted code
- written but never read - redundant code

5.18.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.18.4 Metrics

5.18.4.1 Were all variables written to and read from?

The code could not be checked for variables read and then written and vice versa since the ordering of the CFBs is not predefined. Variables that were written to but never read could be an indication of redundant code. Variables that were read and never written to could be an indication of omitted code, or, alternatively,variables that have been used as constants. The variables listed below do not include variables that were not used only the variables that have not been written to but have been read and vice versa. It would be expected for inputs to be read and not written to and outputs to be written to but not read, but this was not always demonstrated.

Not Written		Not Read	
Type	Quantity	Type	Quantity
DI	1189	DF	6
AI	229	DO	474
WI	1	BA	5
BA	1	B	33
DF	1	I	232
B	89	IA	2
I	4		

Figure 5:11
ESD code variables not written and read

Of the variables declared globally to the ESD system all the inputs that had been used were read only. The digital flag (DF) is treated as an output so it is interesting that it was read but not written. Most of the digital flags that were used were written to and read - 378 while only 6 of them were just written to. 183 of the digital outputs were both read and written to. While only 11 of the 259 declared integers were read and written to.

Not Written		Not Read	
Type	**Quantity**	**Type**	**Quantity**
WI	2	WO	4
B	625	B	851
BA	9	BA	8
I	3	DX	2
IA	1	I	58
TRIP_ALL	5	IA	2
TRIP_HH	51		
TRIP_LIM	71		
TRIP_LL	92		
TRIP_LLL	10		

Figure 5:12
ESD variables declared in units that were not read or written

Of the variables declared as global to one unit in the ESD system, the inputs and outputs were either used as expected or not used at all. 17 of the 19 BA (Boolean arrays) that were used were not read and written to, this can be seen in Figure 5:12 which demonstrates that 9 were not written to and 8 were not read from. The ESD recipes that were not written to could have been used as constants, since they did not contain any inputs only integers and reals.

Not Written		Not Read	
Type	**Quantity**	**Type**	**Quantity**
DI	325	DF	56
AI	199	WO	4
WI	1	DO	12
CODELL	3	I	128
B	11	IA	4
DUAL_LIM	57	B	337
DX	2		
I	267		
IA	1		
SING_LIM	130		
ST	1		

Figure 5:13
F&G code not written and read

Of the global variables in the F&G system, all the inputs used were read and not written to. The recipes consist only of integers and reals (not inputs) so it could be that they were used as constants. Of the integer arrays only 1 was read and written to. The number of outputs used was not equal to the number write only so some of them must have been read.

There were 35 Booleans declared local to a unit and three of them were not written to. The three not written to were declared as false.

This analysis has identified the possibility of large amounts of redundant or omitted code.

5.18.4.2 Some internal procedures should not be used in the same CFB

The analysis identified procedures that should not be used in the same CFB. This has been resolved and they can in fact be used together since the manual was out of date. (see 5.11.4.2)

5.18.5 Conclusions

Data flow analysis proved to be useful. It demonstrated an overview of the whole program which would aid understanding if the workings of the entire program are not understood. It demonstrated where the functions / procedures may have been used incorrectly but in fact were not.

The information about variables was very interesting; the study revealed that of the 4413 global variables declared in the ESD system many were not used as expected by analysis or not at all. There were 3958 global variables declared in the F&G system many of which were not used as expected. This analysis of the code indicated that there could be redundant and omitted code in both of the programs. On further investigation it was demonstrated that many of the variables not written to were used as constants, or a connection to a data link. It did identify three variables that had been used incorrectly which in itself proved the benefit of the technique.

Since the data analysis has proved useful on the ESD and F&G code it would be assumed that it could be used beneficially on other ESD and F&G systems.

5.19 USE CONTROL FLOW ANALYSIS

5.19.1 Goal

Use Control flow analysis to detect poor and potentially incorrect program structures.

5.19.2 Definition of Technique

Control flow :- "The sequence in which operations are performed during the execution of a computer program" [43]. Control flow should identify code which does not follow good programming techniques. A directed graph can then be created and analysed. It should identify:-

- inaccessible code e.g. due to unconditional loops
- knotted code - poorly structured programs can only be reduced to a knot composed of several nodes.

5.19.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.19.4 Metrics

5.19.4.1 What does the procedural control flow look like?

Figure 4:20 and Figure 4:21are control flow diagrams of the F&G code.

The procedural control flow is as expected from the APT top level layout. Figure 4:20

- the main program calls the init parts of all the CFBs
- then the main parts for the high interlocks,
- then the SFC (which is shown on a separate diagram Figure 4:21)
- then the two mathblocks.

This is quite a straight forward control flow graph that does not show complicated flow of control in the code. The graphs were drawn from the translated code.

5.19.4.2 What is the control flow within the code?

A graph was not drawn of the internal structures of the code, although this is possible. The code consisted of GOTO type jumps in the SFC and from a hand performed analysis of the F&G SFC there was no code that was inaccessible. Within the main body of the code there were conditional statements that were nested up to four levels. (See Figure 4:13) The only while loop in the ESD code can be located in one of the CFBs. It was studied to ensure that it would terminate.

5.19.4.3 Does the SFC control flow program look like the SFC diagram?

The SFC chart demonstrates identical control flow to the control flow diagram of the SFC part of the translated code. They both show the same information; this would suggest that the SFC programming language shows a higher level of abstraction than WSL and so should prevent incorrect programming structures.

5.19.5 Conclusions

Control flow diagrams can be drawn of both the top level and the lower levels of the code. The top level diagram did not increase the information about the program structure. It gave information about the size and numbers of units, and number of CFBs at a glance for maintenance etc.

The control flow diagrams of the individual pieces of code may have indicated more information about where complicated and inaccessible code could be located. Since the code consisted mainly of conditional statements a study of the values of the conditionals has greater likelihood of locating the omitted and complicated code than a control flow diagram.

Control flow analysis of the overall layout of procedures and units is not necessary with graphical languages.

5.20 USE STRUCTURED BASED TESTING

5.20.1 Goal

Use structured based testing to apply tests which exercise certain subsets of the program structure.

5.20.2 Definition of Technique

Testing is based on the analysis of the program, where a large fraction of selected program elements are executed. Below is the main selection of subsets that are of differing vigour:-

- **Statement testing**

  Each statement is tested once. In the case of a conditional statement then only one of the paths will be tested. Loops have to be entered at least once.

- **Branch testing**

  All branches of every condition should be tested, i.e. the then, else if, and the else branches should all be executed at least once.

- **Compound condition testing**

  Every condition in the conditional i.e. those parts linked by 'and' and 'or' should be tested as true and false.

- **LCSAJ - (linear code sequence and jump testing)**

  Testing all sequences of code between jumps, or between the start of the program and the start of a jump, or the end of a jump and the end of the program.

- **Data flow testing**

  The test paths are selected by their data usage, i.e. a path where a variable is both written to and read.

- **Call graph testing**

  The test path is so that every function that is used is executed

- **Entire path testing**

  Test all possible paths through the code.

- **FAT's (factory acceptance testing)**

  The tests are performed by checking that each line of the C&E chart is executed correctly.

5.20.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.20.4 Metrics

5.20.4.1 Statement testing

Most of the code is in 'conditional statements'. The level of nesting of the code can be up to four levels if there is an outermost 'else' or 'else if' then none of the inner levels of the 'if' branch are necessarily tested.

	F&G code	**ESD code**
if	8102	6742
else	4190	2353
else if	12	1128

Figure 5:14
Table showing the approximate number of times each of the key words were used in the programs

If the number of conditional statements are taken as a maximum approximation of the number of statement tests and the number of conditional statements minus the maximum number of nest levels in each math block is taken as a minimum number of statement test cases then Figure 5:15 gives an approximate number.

	minimum	maximum
F&G	4057	8102
ESD	6742	3753

Figure 5:15
Table showing the approximate number of statement tests

5.20.4.2 Branch testing

This increases the number of test cases in the code in comparison to statement testing. It is believed that all the test cases should be accessible.

Branch Testing	
F&G	12,304
ESD	10,245

Figure 5:16
Table showing the approximate number of branch tests

5.20.4.3 Compound condition testing

Compound testing represents a large increase in the number of test cases but could prove very useful since many of the conditions are based on more than one conditional. Some of the condition statements are based on a long string of combined conditional. Testing that they work correctly would be beneficial since they are the parts of the code that are most likely to lead to confusion because of there length.

Compound Testing	
F&G	15,694
ESD	21,586

Figure 5:17
Table showing the approximate number of compound tests

5.20.4.4 LCSAJ - (linear code sequence and jump testing)

The ESD code only has one while loop and the F&G code has no while loops, so unless the SFCs and CFBs are included as jumps there would be very little benefit to this form of testing.

Each of the SFC steps should be considered to start and end with a jump, but some of them will need to be tested twice as the first loop may contain different code to the future loops. So executing each step the once would mean that some of the code is not executed.

Using this method each of the CFBs could be treated as individual identities and so the ordering of them would not have to be fixed. Since the ordering is fixed at compile time the start and the end of each of the CFBs could be considered as a jump.

LCSAJ Testing	
F&G	12,304
ESD	10,245

Figure 5:18
Table showing the approximate number of LCSAJ tests

Note all the block of code between the conditional statements would have to be added to the value in Figure 5:18 and the number of CFBs and Steps in the program..

This testing would seem to fit in with this sort of program although there is only one explicit loop in the code.

5.20.4.5 Data flow testing

Data flow testing would be similar to taking a slice through the code and follow a variable being written to and read from. The study in 5.8 indicated that each of the variables were used in a small number of units. This would mean that a high number of the variables would have to be traced through the code to gain a reasonable coverage. It has also been identified that there is a high degree of interaction between variables. It would also be relatively difficult to identify variables that do not overlap and to obtain distinct blocks of code e.g. for the F&G code most slices seem to go into the SFC.

5.20.4.6 Call graph testing

Enforcing that all of the function and procedure calls are carried out would require a level of testing between that of branch testing and compound condition testing. This is due to the fact that in many of the condition statements contain function calls. This is probably easier to obtain than the compound condition testing and it provides more test cases than the branch testing.

5.20.4.7 Entire path testing

Entire path testing would be impossible for these systems. It is even more difficult than with a normal large program because the ordering of the CFBs is undefined. This implies that every possible combination of ordering of the CFBs would have to be tested along with every combination of every path through each of the CFBs and the SFC.

5.20.5 FAT (Factory Acceptance Tests)

Many PLC systems are tested presently by using FATs. These are performed by relating directly to the C&E chart. For each input the outputs are all checked for correctness. This sort of testing covers all the cases for which the program is specified, but it does not necessarily check any of the redundant parts of the code, or self checking parts.

	ESD	F&G
Number of inputs	327	426
Test cases for inputs	868	489
Test cases for intertrips	162	320
Number of outputs	486	376
Number of intertrips	161	265

Figure 5:19
Table demonstrating number of FAT tests

The F&G C&E charts specified quite a high amount of voting whereas the ESD C&E charts did not specify any voting. In some cases in the ESD C&E charts the on and the off value of an input were considered rather than just the on value.

5.20.6 Conclusions

There seem to be considerably less test cases for the FAT than other forms of testing. But the FAT test should perform at least the statement testing in each test since every CFB is executed in every block. The FAT may not test the SFC as that can be turned off.

The more testing that is performed the greater the level of safety. Testing does not identify all of the software faults especially as it is not possible to perform entire path testing. The compound condition testing will force the greatest number of test cases. Branch testing should probably be the minimum test coverage considered and not statement testing.

LCSAJ testing seems to be ideal for this sort of code since it allows each step in a SFC and each CFB in a CFC to be tested individually. Although interface testing between the CFBs and the SFC would also have to be performed.

5.21 USE FMECA

5.21.1 Goal

Use FMECA (Failure Mode and Effects Criticality Analysis) to rank the criticality of component failure which could result in injury, damage or system degradation through single point failures.

5.21.2 Definition of Technique

FMECA's are similar to FMEA's (Failure Mode and Effects Analysis) with more information requiring identification.

FMEA is performed on the hardware normally after the detailed design stage once the hardware and interactions between them have been defined. The method is to identify for each piece of hardware, in the system, the reasons why it might fail, and then identify their failure modes, i.e. the frequency of this specific type of failure. This is normally calculated by the manufacturer. (Care should be taken though since this could be the average and the actual value may be much lower. Also the value obtained by the manufacturer will be under a specific set of test conditions.) The seriousness of a failure should then be assessed e.g. critical.

The FMECA part then identifies what can be done to prevent the seriousness of the failure, i.e. the redundancy and the fail safe design requirements that are needed. It also requires making a note of what has been done to prevent the seriousness of the failings. [13]

5.21.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.21.4 Metrics

5.21.4.1 Use FMECA with the PLC code?

FMECA can be used with these programs, except the ESD and F&G code both have a high proportion of inputs and outputs which would make the analysis long and tedious. It is not known from the code and the C&E charts whether they use the same type of hardware which would reduce the amount of work to be performed. The information about the number of inputs, outputs, valves (and hence hardware devices) is from the code which means that the redundancy identified in this sort of analysis has already been added.

It would be assumed that all hardware parts used would have their failure rate frequency calculated. The F&G system has a total of 728 known hardware devices while the ESD has a total of 2260. Each of these would require a table to be created for them identifying how they can fail, the frequency of the failure, the seriousness of the failure, what can be done to prevent the failure. If it were to say take an hour to perform the analysis on each piece of hardware it would take 3,000 hours. Working a 35 hours week would give 85 weeks worth of work. The original estimation could be inaccurate and it may take more or less time and it will take less time with experience.

5.21.4.2 What information can be obtained?

The information that can be obtained is the location and type of failures that are likely to occur and how to guard against them. If the mean time of failure is very low then either the piece of hardware can be changed, or less dependency put on it, i.e. redundant hardware and spare parts should be easily available during maintenance of the system.

FMECA will identify, on average, how frequently an individual piece of hardware will fail, and not evaluate it in conjunction with any other items of hardware. It will identify where the safety problems are likely to arise and how they can be fixed, but not how hardware failures are likely to interact.

5.21.5 Conclusions

Although the amount of hardware that is required for this system is large the analysis identifies important information. It gives information about where the failures are likely to be found and how to improve the safety. If this information is then studied in conjunction with the layout of the hardware and where and how it is used rather than just the single point of failures it would provide very useful information about how to obtain the safest hardware configuration. This is because "it can be used to identify the redundancy and fail safe design requirements, single point failure modes, and inspection points and spare part requirements." [13] Analysis can be used with hardware in the system and should provide hardware system safety.

5.22 USE SOFTWARE FAULT TREE ANALYSIS

5.22.1 Goal

Use software fault tree analysis (SFTA) to perform an analysis of events that will lead to a hazard or a serious consequence.

5.22.2 Definition of Technique

A starting event is the immediate cause of a hazard; the analysis is performed from branch to root along a tree path, using logical 'and' and 'or' operators between nodes if more than one follow a node. Fault tree analysis was originally developed for identifying problems with hardware but is now being developed for software.

Fault trees can be applied to software but "the analysis is used for verification, as the code must have already been written to generate the trees"[13]. The other benefit of software fault trees is that theoretically much of the process can be automated.

Software fault trees can be used with the pre written software for two methods. The first is to check the internal values of the code, i.e. an array is never out of range, or a variable cannot reach a dangerous value. The other method is to relate the code to a hazard to prove that the hazard cannot occur i.e. an output is not assigned false when it should be true (normally dependent on input). Building a software fault tree is labour intensive but it looks at high level faults and not at specification.

The technique is similar to that used in hardware fault trees; a hazardous event is identified and is made the top (root) event. For example a hazard could be variable x being equal to 10, and then the code is followed down the tree until it is proved that x can not be made equal to 10. The tree is built directly from the code and as such is a representation of it. As branches on the tree are developed that can not possibly occur they are no longer considered. With SFTA it is the loops that cause problems; each iteration of the loop can be studied, which may lead to very large or infinite trees. The other option is to prove the tree by induction in which case the following have to be proved:-

- hazard cannot occur if there is no iteration
- hazard cannot occur if there is 1 iteration
- hazard cannot occur if there is n +1 iterations

If the second two statements can be drawn as identical trees apart from n then the tree can be shown to not cause the desired hazard by induction.[52] The software fault tree analysis condition of a hazard is believed to be the weakest pre condition of that hazard. A software fault tree is preferable though because it gives more detailed information. Nancy Leveson defined the templates of trees found below.

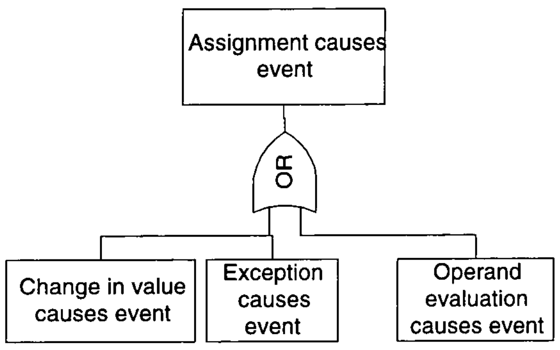


Figure 5:20
Template of an assignment software fault tree

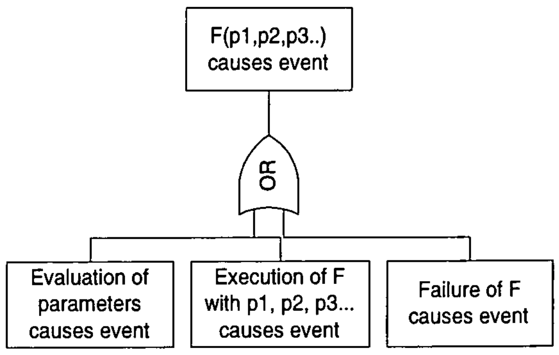

F(p1,p2,p3..) causes event
OR
Evaluation of parameters causes event
Execution of F with p1, p2, p3... causes event
Failure of F causes event

Figure 5:21
Template of a function software fault tree

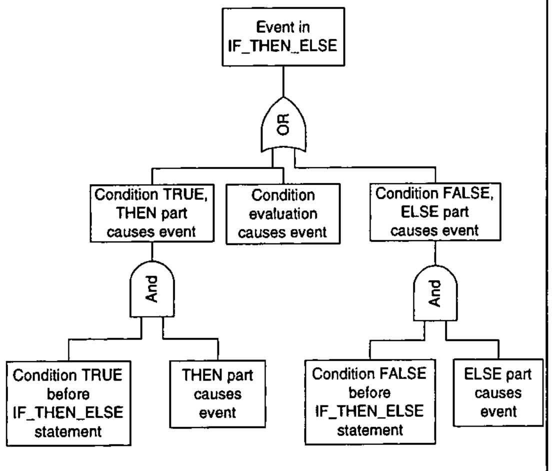

Event in IF_THEN_ELSE
OR
Condition TRUE, THEN part causes event
Condition evaluation causes event
Condition FALSE, ELSE part causes event
And
And
Condition TRUE before IF_THEN_ELSE statement
THEN part causes event
Condition FALSE before IF_THEN_ELSE statement
ELSE part causes event

Figure 5:22
Template of an 'if-then-else' software fault tree

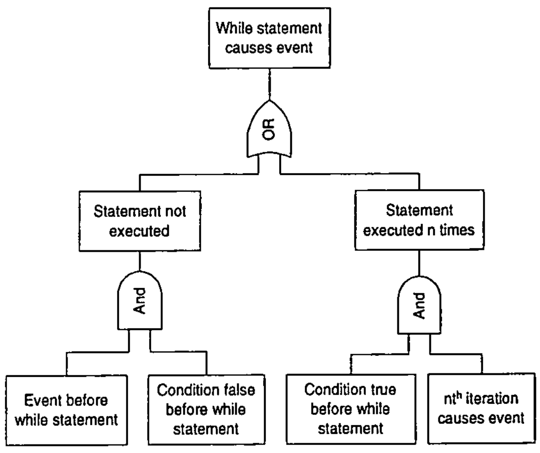

While statement causes event
OR
Statement not executed
Statement executed n times
And
And
Event before while statement
Condition false before while statement
Condition true before while statement
nth iteration causes event

Figure 5:23
Template of a while loop software fault tree

[13]

5.22.3 Questions

1. Is this technique possible to use with the provided PLC code?
2. Does it give information about any safety features?

5.22.4 Metrics

5.22.4.1 Can a software fault tree be built for a math block?

A software fault tree can only be built for any math block if it is possible to build a sub fault tree for all of the statement types found within the code. A template has been built for all the statement types apart from:-

- Edge function (defined in the math language)-due to it requiring previous knowledge
- Timers - as the trees do not hold time information
- While loop - too much iteration to template.

The edge function has to become an end node within a tree. The timers can sometimes be analysed and the while loop can be analysed. Two examples of analysed math code have been included in the code the first one is to determine if the digital output value PAHH_14320IS = true (Figure 5:27) and the second is to see if an array in the while loop can go out of range (Figure 5:29). Before the trees could be built a template for each type of statement has to be built a selection are included give in Figure 5:24 to Figure 5:26.

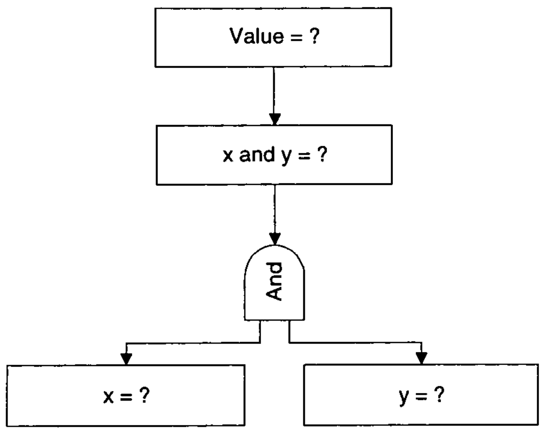
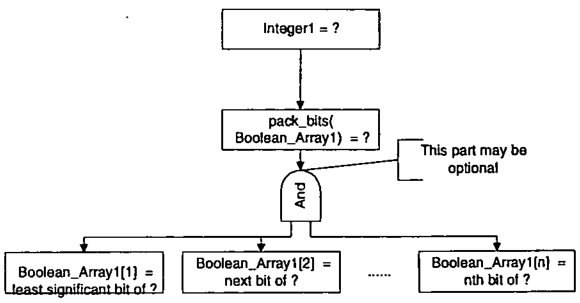


Figure 5:24
An assignment statement



Integer1 = ?
pack_bits(Boolean_Array1) = ?
This part may be optional
And
Boolean_Array1[1] = least significant bit of ?
Boolean_Array1[2] = next bit of ?
......
Boolean_Array1[n] = nth bit of ?

Figure 5:25
Pack_bits function

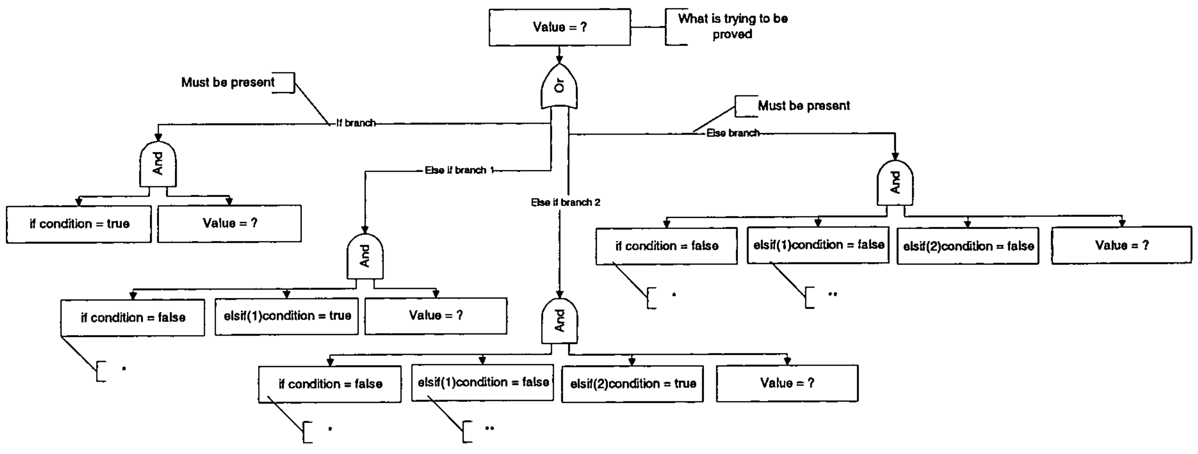


Figure 5:26
An if statement

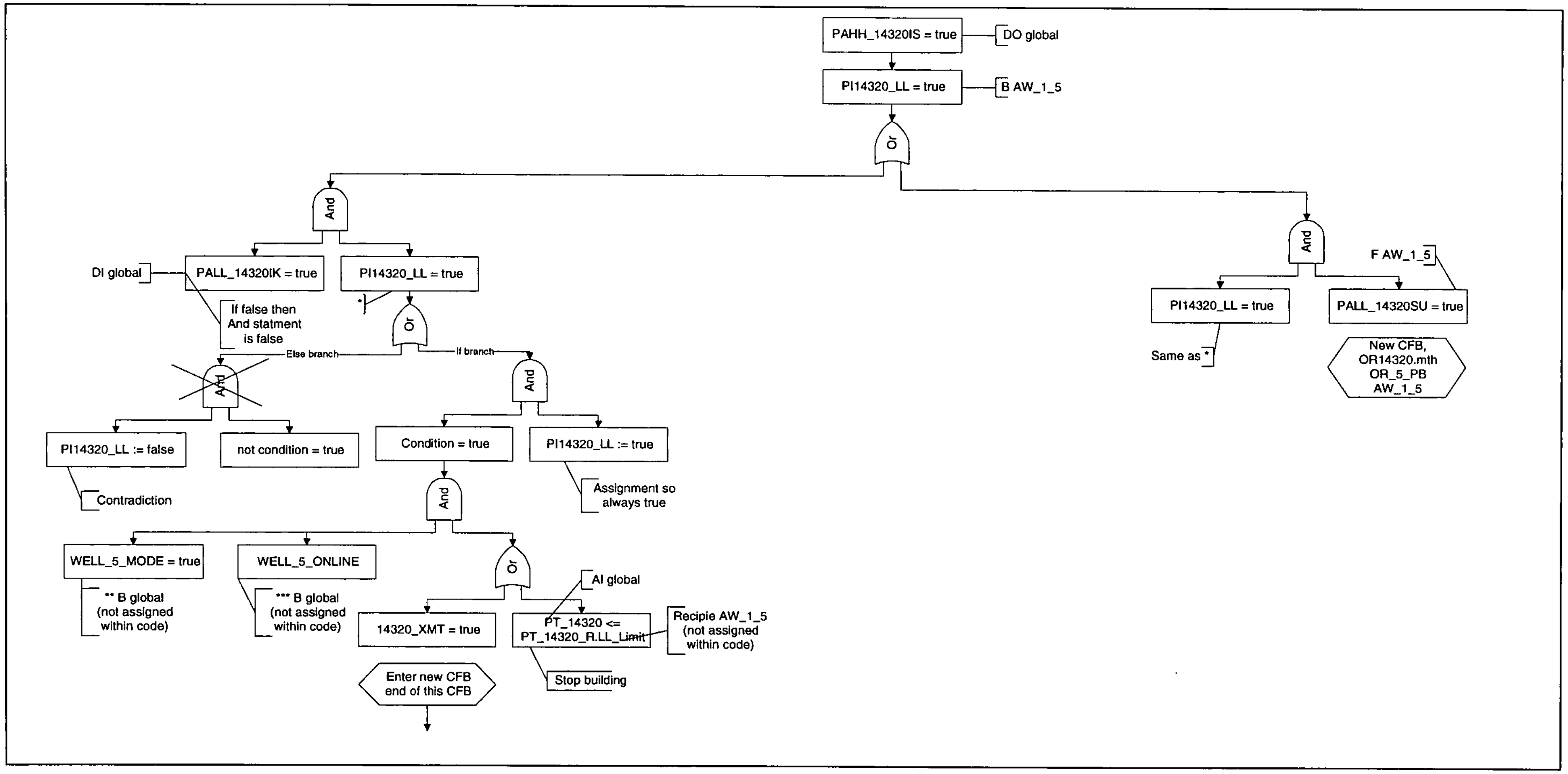


Figure 5:27
A software fault tree for the digital output = true as the hazard only entering one math block

Building the tree for the CFB that contained the while loop was more difficult than for the one containing the digital output. Building trees with functions, conditional statements and assignments is a case of filling information into a template.

The math block corresponding to the fault tree below (without comments)

```
Integer: x;
Integer: y;

Begin
Init
I := 1;                     { new_array index }
counter := 1;                 { Counts the number of alarms detected }

Body
x := 1;                     { Number of tags in FIRST_UP_LST }
y := 1;                     { Increment to array_text location }

WHILE ((x <= 450) AND (counter < 50)) LOOP   { Check all ID_TAG locations }
                              { until 50 alarms detected   }
   IF (ID_TAG[x]) THEN                { If tag is in alarm }
      MARK_TAG[x] := true;            { This tag already accounted for}
      counter := counter + 1;       { Increment alarm counter }
      time_fup[I]    := array_time[1];   { Pick up the current time }
      time_fup[I+1]  := array_time[2];   { Pick up the current time }
      time_fup[I+2]  := array_time[3];   { Pick up the current time }
      time_fup[I+3]  := array_time[4];   { Pick up the current time }
      time_fup[I+4]  := array_time[5];   { Pick up the current time }
      time_fup[I+5]  := array_time[6];   { Pick up the current time }
      time_fup[I+6]  := array_time[7];   { Pick up the current time }
      time_fup[I+7]  := array_time[8];   { Pick up the current time }
```

```
        new_array[I]    := array_text[y];    { Copy the text by copying the  }
        new_array[I+1]  := array_text[y+1];  { integer value                 }
        new_array[I+2]  := array_text[y+2];
        new_array[I+3]  := array_text[y+3];
        new_array[I+4]  := array_text[y+4];
        new_array[I+5]  := array_text[y+5];
        new_array[I+6]  := array_text[y+6];
        new_array[I+7]  := array_text[y+7];
        new_array[I+8]  := array_text[y+8];
        new_array[I+9]  := array_text[y+9];
        new_array[I+10] := array_text[y+10];
        new_array[I+11] := array_text[y+11];
        new_array[I+12] := array_text[y+12];
        new_array[I+13] := array_text[y+13];
        new_array[I+14] := array_text[y+14];

        IF (counter < 50) THEN
           I := I + 15;              { Increment the new_array index }
                                   { ready to write the next tag   }
        ELSE
           I := 1;                  { Reset new_array index         }
        ENDIF;
    ENDIF;
    x := x + 1;                      { Increment the ID_TAG index    }
    y := (((x - 1) * 15 ) + 1 );        { Increment the array_text index}

END LOOP;
```

Figure 5:28
Code of the ESD While loop

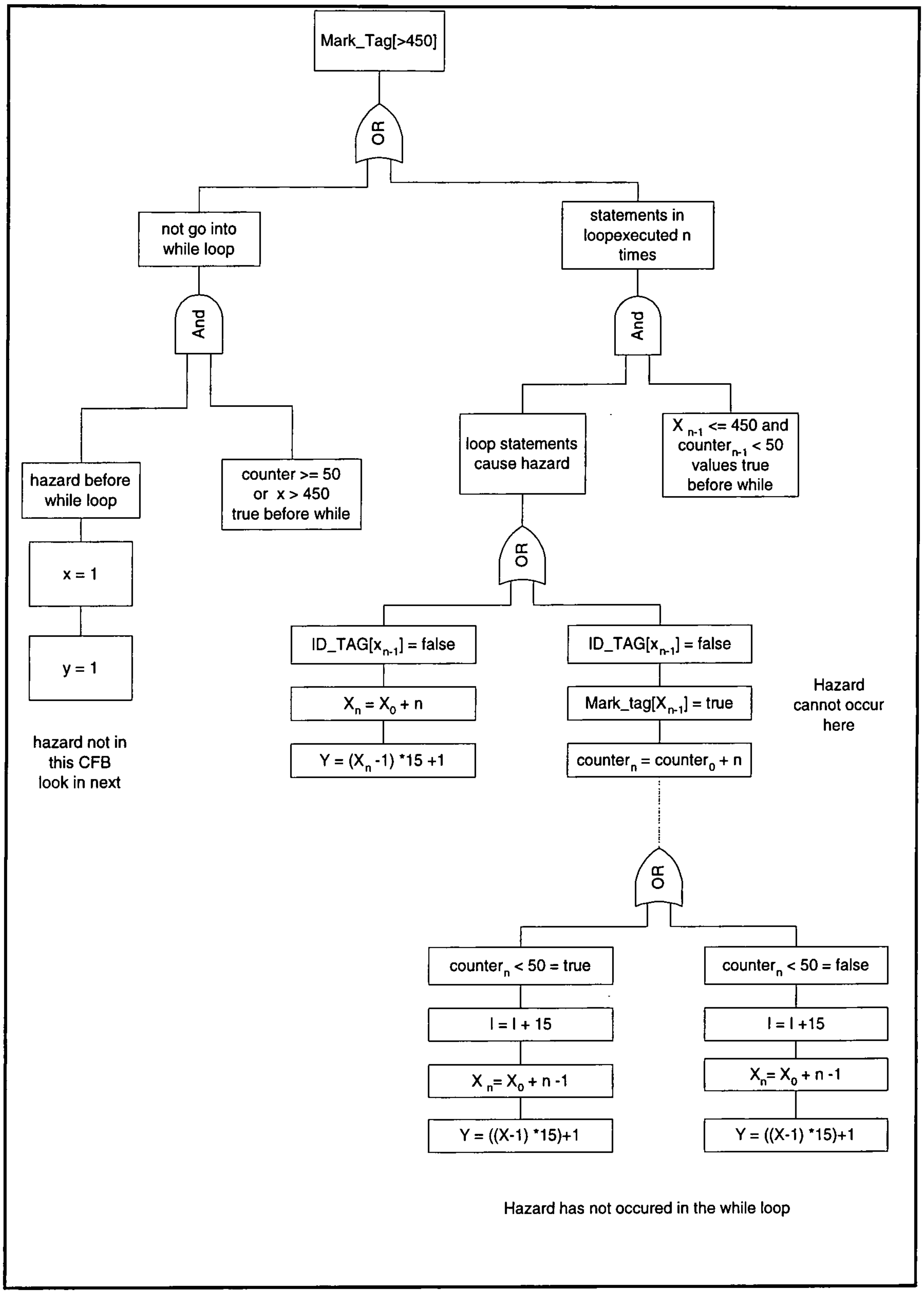


Figure 5:29
Hazard in while loop

It can be seen from the above examples that although it is possible to perform the SFTA on an individual CFB the variables in blocks are dependent on variables defined elsewhere. Building trees over more than one block is considered in section 5.22.4.3. Each case of a conditional statement has to be considered so if a tree goes through a 'multi' branched 'conditional statement' with a high number of conditions then the tree will be large.

5.22.4.2 Can a software fault tree be built for a math block?

The code in each step of the SFC is math language and so is treated as above. The only problem with the SFC is that after each step has been analysed the transitions can cause looping problems. This is because each transition into the step and each transition out of the step has to be analysed to determine if the step could be on its nth re-execution. This leads to relatively large and complex trees but it was possible to build the software fault tree for both the ESD and F&G SFC. There are 10 templates that are associated with building the tree for the SFC, and the actual trees covered three pages of A3.

In the F&G code most of the slices through the code demonstrate some sort of dependency on the SFC. For the trees to be used they would ideally be automated and it is not possible to automate the transitions of the SFC, the advantage though is that the tree for the transitions only need be built once as it can be reused.

5.22.4.3 Can the whole program be combined?

The examples above have demonstrated that SFTA on a single CFB or SFC is not sufficient, the remainder of the code has to be considered. The method of joining blocks would be to treat them as sequential pieces of code since only the piece of code relevant to the node are used the tree can just continue. If a variable in a node at the end of a block is assigned to in more than one block then a tree has to be built for each of the blocks that the variable has a value assigned to it. Analysis so far has not demonstrated any instances of this occurring.

5.22.5 Conclusions

SFTA can be performed on the code supplied. Loops are generated in the tree through numerous executions of the code. Since the ESD code has only one specific while loop and the F&G none, the SFTA should be easy to perform; it is not that simple though due to the SFC, timers and edge function. Another issue is also the fact that the whole system is developed in a loop. This sort of code would be ideal for SFTA had all the code been written using CFCs. Although it is not too difficult to build a tree for the SFC it just has to be performed manually.

The trees provided much information about the code when they were drawn, such as hardware addresses that were assigned to more than one variable and potential looping of code.

The advantage of this technique is that it does not compare the code to the specification which could be erroneous but to the hazardous events and whether they can be performed. Although this technique in many parts can be automated more information about the code can be obtained by doing it manually using the templates. A template has to be designed for each type of statement supported by the language.

5.23 A SUMMARY OF THE RESULTS

5.23.1 Insisted On By The Compiler

- No pointers
- No recursion
- No dynamic variables
- No dynamic objects
- No unconditional jumps
- No programmed interrupts

5.23.2 Not Insisted On By The Compiler But Were Used

- Coding standards
- limiting module size (although some CFBs were still relatively long)

5.23.3 Techniques Supported By The Language And Used

- Strongly typed programming languages
- A defined subset of CFCs
- A defined subset of Math language (Structured text)
- A defined subset of SFCs *(none of the languages used had all the recommended properties.)*
- Designing easily analysable code.

5.23.4 Techniques Supported By The Translated Code

- Control flow analysis
- Data flow analysis

5.23.5 Techniques Supported By The APT Code And Extra Information

- Software fault tree analysis

5.23.6 Techniques Not Supported By The APT Tool

- Reuse of verified modules
- Information hiding and encapsulation

5.23.7 Techniques That Could Be Used With Difficulty

- FMECA - due to the high number of hardware devices
- Structure based testing - entire path testing is not possible, although statement path and branch testing may have proved possible.

6. CONCLUSION

The ESD and F&G code were successfully translated into WSL. Analysis was then performed on the code against a selection of IEC 1508 HR development techniques.

The code that was translated was the FAT code for an Offshore Oil Platform. This code has been operational for at least 2 years. It was written using Siemens TI high level languages; CFCs, SFCs and math language. The languages were automatically translated into WSL using a number of programs. Perl scripts were used to determine variable types, names of CFBs and the ordering of the translation and building of the code. A C program was used to build the SFC equivalent in WSL and a YACC parser was built to translate each block of math language. Prior to building the parser every construct in the SFC, CFC and math language languages had to be identified, defined as precisely as possible and then defined in WSL (see mapping document Appendix III).

The languages were translated into the following constructs:-

- SFC → An action system
- CFC → A procedure containing only procedure calls to each CFB, which in turn were procedures.
- Math language → The equivalent text based structure in WSL.

During development of the mapping document and parser every construct had to be defined in WSL. This was performed by identifying the representation that closely mapped the PLC languages in WSL. The judgement was based on the knowledge of the languages that was available, since no formal semantics of the PLC languages were avaliable. During the analysis phase of the thesis, some minor problems with the WSL representation were identified, although the syntax and semantics were technically correct. Analysis would have been easier if a different representation had been used for a number of the constructs. These included:-

- There was an 'init' procedure in the WSL representation of the SFC but none of the analysed code actually made use of this facility. Also the fact that the active step in the main SFC remains active during execution of the safe SFC until control returns to the main SFC was not represented in WSL. This fact though was recognised implicitly for use during the analysis.
- There was an extra procedure at the end of each block in which flags had been used to check if the flags had been turned 'on'. Only the 'clear' and 'latch' commands were used, not the 'on' command. Therefore the data collected regarding which unit's variables were read and written to was slightly inaccurate.
- The time representation within the timers was difficult to analyse both from the PLC and WSL code as only static and not dynamic analysis was performed.

The key characteristics of the ESD and F&G code were identified and are summarised below:-

- The number of lines of code (including blank lines but not comments) was 199,431 for the ESD system and 88,607 for the F&G system.
- The ESD system had 38 units while the F&G code had 55 units.
- Most of the variables declared in the systems are global variables. Since there was no parameter passing, all data transfer was via global variables. The analysis of a subset of the global variables identified that they were predominately used in one or two units to a maximum of seven in the ESD code and three in the F&G code.
- Variables were not used exactly as expected but most when analysed by the company supplying the code had been used correctly.
- The CFCs consisted of two types of distinct CFBs, the active math blocks and interlocks.
- The SFC consisted of steps and transitions. The transitions demonstrated convergence and divergence but no parallel execution.
- The math language constructs that were used were: assignment, conditional statements, while loop, APT defined procedures and functions, comments.

- The conditional statements were nested to a maximum of four levels. A high proportion of the blocks had four levels of nesting in the F&G code. The majority of the blocks in the ESD code had a maximum of one or two levels of nesting.
- There was only one while loop in the analysed code and that was in a CFB in the ESD code.

The analysis of the code was performed against 22 of the HR development techniques for producing SIL 1,2 or 3 software. The techniques were analysed to determine whether:-

- They had been used
- They could have been used
- They could not have been used due to the applications
- They could not have been used due to the programming environment

Many of the analysed techniques could have been used or had been used. There were some techniques that could not have been used due to either the applications or the programming environment used.

6.1 CRITERIA FOR SUCCESS

The ESD and F&G code were automatically translated and analysed; but was the project successful? This can only be determined by analysing the criteria for success (section 1.2).

The top priorities of the thesis were:-

1. To identify key highly recommended techniques from SIL 1,2 or 3 that can be analysed using the data available.

The techniques that were chosen were dependent on the data available for analysis. The data available was the ESD and F&G code, the C&E charts and the APT programming environment. The analysed techniques addressed the following:-

- Coding techniques (e.g. limit the use of pointers, use information hiding)
- Programming languages (e.g. use a strongly typed language, use a safe subset of the PLC language)
- Analysis techniques (e.g. use structured based testing, use software fault tree analysis)

2. To analyse the code to asses the feasibility of using the technique with the specific safety critical PLC code.

Each of the designated techniques of IEC 1508 were analysed with respect to the code. Each technique was analysed individually although some of the data could be used for multiple techniques. The techniques were divided into the following categories after analysis:-

- techniques insisted on by the compiler
- techniques not insisted on by the compiler but used
- techniques supported by the translated code
- techniques supported by the APT tool and extra information
- techniques supported by the APT tool
- techniques that could be used but only with difficulty

The techniques that were insisted on by the compiler were development techniques that analysis easily identified could be used. Those that could only be used with difficulty were harder to perform the analysis on, to determine if they could be used. All the techniques that were analysed were performed or calculations performed to determine how they could be used, as with the different number of test cases.

The GQM approach was a useful basis for analysis as it meant that each technique was analysed using a similar structure. The method also allowed data transferral between the techniques being analysed.

3. To identify the general characteristics of the ESD and F&G PLC code on an offshore platform.

These were discussed in detail in chapter 4 and a summary is given above.

The secondary priorities of the thesis were to determine:-

1. If a single language could be used to replace the three PLC languages.

The three languages were translated into the one text based language -WSL. Two of the original languages were graphical languages, while the third was a textural language. The source and target languages were all high level languages. The randomness of the compilation order of the CFBs was lost during translation.

2. If any language deficiencies were identified in the PLC languages.

None of the PLC languages were formally defined hence they had to be formalised before translation could commence. There were sections of the PLC language definition that were confusing. These included:-

- Execution order
- Parallel execution within steps of an SFC
- When an on flag was automatically set to false
- How frequently input variables were read

The languages allowed the same hardware address to be assigned to more than one variable - a form of aliasing. The languages also enabled Boolean arrays to be converted to integers and back within blocks of code. Both these can cause maintainability and understandability problems in safety critical code.

3. If it is helpful to perform analysis in this way and what the benefits and problems were.

The analysis identified much data about the code and the techniques that were analysed. This data could then be used in association with other similar applications. Many of the problems that occurred during analysis of the code were due to the sheer size and quantity of data that was generated and manipulated. The size of the ESD and F&G programs were very memory intensive both during translation and analysis. Also WSL was not ideally suited to being the target language. WSL does not have any concept of time or of variable type. Some of the primitive programming constructs of the math language were not available in WSL; this probably would have been true for any target language. An example would be:-

```
A := B or C     was converted to       A := if (B or C) then true else false
```

The analysis was beneficial to the company that wrote the code as it identified variables that had been used but not read, and variables that had been read but not written. This according to IEC 1508[8] implied omitted or redundant code. When the variables were analysed by the company three of the variables had been used incorrectly. The analysis was also beneficial to Siemens as it identified an error in the documentation of the math language specification, which is in the process of being remedied.

It therefore can be concluded that all the criteria for success have been met.

6.2 FURTHER WORK

If further work in the field was to be performed, it would be interesting to analyse some of the techniques in more detail, especially software fault tree analysis. It would also be interesting to be able to analyse more of the techniques identified in IEC 1508 for developing safety critical code.

Another avenue of further work would be to analyse another similar safety critical application. This would identify the characteristics specific to the analysed system and those general to similar applications.

6.3 SUMMARY

In summary all the criteria for success have been met. The ESD and F&G PLC code were successfully translated and analysed against IEC 1508. The key characteristics of the code were identified, and tools were developed to aid the analysis.

ACRONYMS

AI	Analogue Input
ALARP	As Low As Reasonably Possible
APT	Application Productivity Tool
B	Boolean
BA	Boolean Array
BCD	Binary Coded Decimal
BNF	Backus Naur Form
C&E	Cause and Effect Charts
CFB	Continuous Function Block
CFC	Continuous Function Chart
CODELL	Recipe
CPU	Central Processing Unit
DF	Digital Flag
DI	Digital Input
DO	Digital Output
DUAL_LIM	Recipe
DX	DO10 Array
EEPROM	Electrical Erasable Programmable Read Only Memory
ESD	Emergency Shut Down
F&G	Fire and Gas
FAT	Factory Acceptance Test
FBD	Function Block Diagrams
FT	Fast Timer
GQM	Goal Question Metric
HAZOP	Hazards and Operability Analysis
HR	Highly Recommended
HSE	Health and Safety Executive
I	Integer
I/O	Input / Output
IA	Integer Array

IEC 1131-3	Programmable controllers Part 3: Programming languages
IEC 1508	Functional safety of electrical/ electronic/ programmable electronic safety related systems
IL	Instruction Lists
LD	Ladder Diagrams
MA	Maintainers Assistant
OO	Object Oriented
PC	Personal Computer
PES	Programmable Electrical System
PID	Proportional Integral Derivative
PLC	Programmable Logic Controller
R	Real
RAM	Random Access Memory
RLL	Relay Ladder Logic
ROM	Read Only Memory
SCS	Safety Critical System
SFC	Sequential Function Chart
SFPGM	Special Function Program
SFT	Software Fault Tree
SFTA	Software Fault Tree Analysis
SI	Safety Integrity
SIL	Safety Integrity Level
SING_LIM	Recipe
ST	Structured Text
ST	Slow Timer
VDD	Dual Valve
VDM	Vienna Development Method - a formal method
VSS	Single Valve
WI	Word Input
WO	Word Output
WSL	Wide Spectrum Language
YACC	Yet Another Compiler Compiler

REFERENCES

[1] J. McDermid, "Introduction and Overview to Part II," in *Software Engineer's Reference Book*, J. McDermid, Ed.: Butterworth Heinmann, 1991.

[2] R. Mortimer, "Data Re-Engineering Using Formal Transformations," in *Computer Science*. Durham: University of Durham, 1998.

[3] A. T. Bertztiss, "Safety Critical Software - A Research Agenda," *International Journal of Sotfware Engineering and Knowledge Engineering*, vol. 4, pp. 165-181, 1994.

[4] "Systems Challenge -- Microcontrollers," . http://www.industrialtechnology.co.uk/micro2.htm, 1996.

[5] K. Clements-Jewery and W. Jeffcoat, *The PLC Workbook Programmable Logic Controllers Made Easy*: Prentice Hall, 1996.

[6] A. Chandor, J. Graham, and R. Williamson, *The Penguin Dictionary of Computers*.

[7] IEC, "IEC 16708 Analysis Techniques for Dependability - Reliability Block Diagram Method," .

[8] IEC, "Draft IEC 1508 - Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety Related Systems.," 28/9/96.

[9] M. P. Ward and K. H. Bennett, "Syntax and Semantics of the Wide Spectrum Language MetaWSL," : submitted to IEEE transactions on Software Engineering, 1998.

[10] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Prgramming Languages*, vol. 3, pp. 121-189, 1995 September.

[11] "The Safety Critical Systems Club," : http://www.cs.ncl/research/csr/clubs/scsc.html, 1998.

[12] J. McDermid, "Issues in the Development of Safety Critical Systems," in *Safety Critical Systems, Current Issues Techniques and Standards*, F. Redmill and T. Anderson, Eds.: Chapman and Hall, 1993, pp. 16 -42.

[13] N. Leveson, *SafeWare : System Safety and Computers*. Reading, Mass.: Addison-Wesley, 1995.

[14] M. Wilikens, M. Masera, and D. Vallero, "Integration of Safety Requirements in the Initial Phase of the Project Lifecycle of Hardware/ Software Systems An Experience Report Based on the Application of IEC 1508," in *SAFECOMP 97*, P. Daniel, Ed. University of York: Springer, 1997, pp. 83-108.

[15] J. Penny, "DO178B - Time for a Change?," *Safety Systems the Safety Critical Sytems Club Newsletter*, vol. 7, pp. 4-6, 1998.

[16] P. G. Bishop, "Dependability of Critical Computer Systems 3," , vol. 3. London: Elsevier Applied Science, 1990.

[17] I. Sommerville, *Software Engineering*, Fourth ed: Addison-Wesley, 1992.

[18] M. o. D. (U.K), "Safety Management Requirements for Defence Systems Containing Programmable Electronics.," in *Second Draft Defence Standard 00-56*, August 1996.

[19] D. o. D. (U.S), "System Safety Program Requirement. Military Standard MIL-STD 882C," , January 1993.

[20] M. I. S. R. A. (U.K), "Development Guidelines for Vehicle Based Software," , November 1994.

[21] NATO, "Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems. Standardization Agreement STANAG 4404," .

[22] P. A. Lindsay, "A Systematic Approach to Software Safety Integrity Levels," in *SAFECOMP 97*, P. Daniel, Ed. York: Springer, 1997, pp. 70 -82.

[23] B. B. C. Education, "Disaster Piper Alpha," . http://www.bbc.co.uk/education/disaster/piper.htm, 1998.

[24] R. A. Delemos and T. Anderson, "Analysis of Timeliness Requirements in Safety Critical Systems," *Lecture Notes in Computer Science*, vol. 571, pp. 171-192, 1991.

[25] N. H. Vaidya and D. K. Pradhan, "Fault-Tolerant Design Strategies for High Reliability and Safety," *IEEE Transactions on Computers*, vol. 42, October 1993.

[26] J. Bowden and V. Stavridou, "Safety Critical Systems, Formal Methods and Standards," *Software Engineering Journal*, vol. July 1993, pp. 189-205, 1993.

[27] J. Martin, *"Design of Real-Time Computer Systems*", Prentice Hall, New Jersey 1967.

[28] H. Thane, "Safe and Reliable Computer Control Systems: an Overview," in *SAFECOMP 97*, P. Daniel, Ed. York: Spriner, 97, pp. 25 - 36.

[29] S. P. Wilson, J. A. McDermid, P. M. Kirkham, C. H. Pygott, and D. J. Tombs, "Computer Based Support for Standards and Processes in Safety Critical Systems," in *SAFECOMP 97*, P. Daniel, Ed. York: Springer, 1997, pp. 197 - 209.

[30] T. P. Kelly and J. A. McDermid, "Safety Case Construction and Reuse Using Patterns," in *SAFECOMP 97*, P. Daniel, Ed. York: Springer, 1997, pp. 55 - 69.

[31] N. Storey, *Safety Critical Computer Systems*: Addison-Wesley Longman, 1996.

[32] J. C. Knight and L. G. Nakano, "Software Test Techniques for System Fault-Tree Analysis," in *SAFECOMP 97*, P. Daniel, Ed. York: Springer, 1997, pp. 369 - 380.

[33] T. Anderson and P. A. Lee, *Fault Tolerant Principles and Practice*. London: Prentice Hall Int, 1981.

[34] B. Littlewood and L. Singinc, "The Risk of Software," *Scientific America*, vol. 1267, pp. 38-43, November 1992.

[35] B. Malcolm, "The JFIT Safety Critical Systems Research Programme, Origins and Intentions," in *Safety Critical Systems, Current Issues, Techniques and Standards*, F. Redmill and T. Anderson, Eds.: Chapman & Hall, 1993, pp. 41-63.

[36] A. M. Dearden and M. D. Harrison, "Using Executable Interactor Specifications to explore the Impact of Operator Interaction Errors," in *SAFECOMP 97*, P. Daniel, Ed. York: Spriner, 97, pp. 138-147.

[37] B. B. C. Education, "Disaster Our Technology Just Can't Go Wrong," . http://www.bbc.co.uk/education/disaster/essay.htm, 1998.

[38] K. H. Bennett and L. M. Williamson, "Understanding PLC code," *submitted to IEEE nt. Workshop on Program Comprehension, 1999*, 1998.

[39] D. Hedley and R. G. Kirsopp, "The Testing of Ladder Logic Programs for PLCs (Programmable Logic Controllers)," *EuroSTAR'93*, pp. 25-28, 1993 October.

[40] B. Tinham, "Open Control will IEC 1131 Provide an Answer," *C&I*, vol. 1996, pp. 31-32, 1996 October.

[41] IEC, "IEC 1131_3 Programmable Controllers - Part 3: Programming languages," .

[42] D. J. Maisey, "How Suitable are the IEC 1131_3 Languages for Safety Critical Software," *High Integrity Systems*, vol. 1, pp. 351-357, 1995.

[43] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *http://hexham.dur.ac.uk/ieee/610_12.html*, 1983.

[44] C. S. French, *Computer Science*, 4 ed. London: DP Publications Ltd.

[45] J. P. Bennett, *Introduction to Compiling Techniques A First Course Using ANSI C, LEX and YACC*: The McGraw-Hill International, 1990.

[46] Terrance W. Pratt and M. V. Zelkowitz, *Programming Language Design and Implementation*, 3rd ed.

[47] Siemens, "SIMATIC TI505 Programming Reference Manual (4.x)," .

[48] E. Younger, "BYLANDS Program Transformations," . http://www.dur.ac.uk/~dcs1ejy/Bylands/.

[49] V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, vol. 14, pp. 758-773, 1988 June.

[50] Siemens, "SIMATIC APT Programming Manual (4.2.1 and 4.3)," .

[51] Siemens, "SIMATIC APT Programming Reference (graph/math) Manual (4.2.1 and 4.3)," .

[52] J. R. Taylor, *Fault Tree and Cause Consequence Analysis for Control Software Validation*: Riso National Laboratory, DK-4000 Rosilde, Denmark, 1982.

APPENDIX I

BNF FOR WSL

NON-TERMINALS

```
program                     ::=  ( statement_sequence )? <EOF>
statement_sequence          ::=  statement ( ";" ( statement )? )*
statement                   ::=  ( a_or_x_proc_call | assert_statement | assignment |
                                 comment_statement | action_system | call_statement
                                 | if_statement | loop_statement | skip_statement |
                                 var_statement | where_statement | while_statement |
                                 procedure_call )
a_or_x_proc_call            ::=  proc_call_type
proc_call_type              ::=  ( "!" | "!p" ) a_proc_call
                            |    "!xp" x_proc_call
a_proc_call                 ::=  name actual_parameters_with_var
x_proc_call                 ::=  name actual_parameters_with_var
actual_parameters           ::=  "(" ( expression ( "," expression )* ")" | ")" )
actual_parameters_with_     ::=  "(" ( expression ( "," expression )* )? "var" ( lval (
var                              "," lval )* )? ")"
actual_parameters_with_     ::=  "(" ( expression ( "," expression )+ )? ( "var" lval ( (
optional_var                     "," lval )+ )? )? ")"
assert_statement            ::=  "{" condition "}"
assignment                  ::=  "<" assign ( ( "," assign )+ )? ">"
                            |    assign ( "," assign )*
assignment_sequence         ::=  assignment_option ( ( "," assignment_option )+ )?
Assignment_option           ::=  ( assignment )?
Action_system               ::=  "actions" ":" name ":" action_sequence
                                 "end_actions"
action_sequence             ::=  action_option ( ( "." action_option )+ )?
Action_option               ::=  ( action )?
Call_statement              ::=  "call" name
comment_statement           ::=  ( "comment" | "#" ) ":" <STRING>
if_statement                ::=  "if" condition ( "then" | "->" ) statement_sequence (
                                 "elsf" condition ( "then" | "->" ) statement_sequence
```

		)* ("else" statement_sequence)? "fi"
loop_statement	::=	"do" statement_sequence "od"
procedure_call	::=	name actual_parameters_with_var
skip_statement	::=	"skip"
var_statement	::=	"var" assignment ":" statement_sequence "end"
where_statement	::=	"begin" statement_sequence "where" (definition)* "end"
while_statement	::=	"while" condition loop_statement
proc_formal_parameters	::=	"(" (lval ("," lval)*)? "var" (lval ("," lval)*)? ")"
formal_parameters	::=	"(" (lval ("," lval)*)? ")"
formal_parameters_with_ optional_var	::=	"(" (lval ("," lval)*)? "var" (lval ("," lval)*)? ")"
expression	::=	expression_or_condition
expression_or_condition	::=	logical_or_condition
logical_or_condition	::=	logical_and_condition ("or" logical_and_condition)*
logical_and_condition	::=	relational_condition ("and" relational_condition)*
relational_condition	::=	set_in_condition (relational_op set_in_condition)*
relational_op	::=	"="
	|	"<"
	|	">"
	|	"<="
	|	">="
	|	"<>"
	|	"xor_bit"
	|	"or_bit"
	|	"and_bit"
set_in_condition	::=	set_union ("in" set_union)?
Set_union	::=	set_intersection (("\\" | "\/") set_intersection)*
set_intersection	::=	concat_expression ("/\" concat_expression)*
concat_expression	::=	adding_expression ("++" adding_expression)*

```
adding_expression            ::=  multiplying_expression ( add_op
                                  multiplying_expression )*
add_op                       ::=  "+"
                             |    "-"
multiplying_expression       ::=  power_expression ( mult_op power_expression )*
mult_op                      ::=  "*"
                             |    "/"
                             |    "mod"
                             |    "div"
power_expression             ::=  funct_call_slice_expression ( "**"
                                  funct_call_slice_expression )?
Funct_call_slice_expressi    ::=  primary ( actual_parameters | slice_values )?
on
Slice_values                 ::=  "[" slice_range "]"
slice_range                  ::=  expression
segment_specifier            ::=  ( ".." | "," ) expression
primary                      ::=  ( name | ( "[" expression "]" )+ | "(" expression ")" |
                                  mw_or_x_function_call "(" expression ( ","
                                  expression )* ")" | "if" condition "then" expression
                                  "else" expression "fi" | "abs" "(" expression ")" |
                                  "frac" "(" expression ")" | "int" "(" expression ")" |
                                  "sgn" "(" expression ")" | "max" "(" expression ( ","
                                  expression )* ")" | "min" "(" expression ( ","
                                  expression )* ")" | "powerset" "(" expression ")" |
                                  "{" expression "|" condition "}" | "array" "("
                                  expression "," expression ")" | "head" "(" expression
                                  ")" | "tail" "(" expression ")" | "last" "(" expression
                                  ")" | "butlast" "(" expression ")" | "length" "("
                                  expression ")" | "reverse" "(" expression ")" |
                                  number | "-" primary | <STRING> | "not" primary |
                                  "true" | "false" | "integer" "?" "(" expression ")" |
                                  "even" "?" "(" expression ")" | "odd" "?" "("
                                 expression ")" | "MyVect" "?" "(" expression ")" |
                                 "set" "?" "(" expression ")" | "name" "?" "("
                                 expression ")" | "empty" "?" "(" expression ")" |
                                 "subset" "?" "(" expression "," expression ")" |
                                 "member" "?" "(" expression "," expression ")" |
                                 "sequence" "?" "(" expression ")" )
mw_or_x_function_call      ::=   ( mw_function_call | x_function_call )
mw_function_call           ::=   "@" name ( "?" )?
x_function_call            ::=   ( "!xf" | "!f" | "!xc" ) name
number                     ::=   <integer>
condition                  ::=   expression_or_condition
lval                       ::=   name ( slice_values )?
Name                       ::=   ( "%" )? <IDENTIFIER>
definition                 ::=   proc_definition
                           |     funct_definition
proc_definition            ::=   "proc" name proc_formal_parameters "=="
                                 statement_sequence "."
funct_definition           ::=   "funct" name formal_parameters "==" (
                                 statement_sequence ":" )? expression "."
assign                     ::=   lval ":=" expression
action                     ::=   name "==" statement_sequence
```

APPENDIX II

GRAMMAR FOR THE MATH LANGUAGE

The grammar rules that were used in the BISON parser for the math language translator follows. The word sin capitals are tokens, the other words can be expanded further. /* ? */ is a comment.

```
mathblock: /* empty */ | declarations BEGIN main | declarations BEGIN main
        | MATH PRAGMA '(' '"' RLL '"' ')' ';' declarations BEGIN main
        |MATH declarations BEGIN main ;
main:INIT initialisation body |statements ;

declarations: /* empty */ | declarations declaration ';' ; declaration: /* empety */
        | BOOLEAN ':' b_variables constant | INTEGER ':' i_variables constant
        | REAL ':' r_variables constant
        | BOOLEAN RETENTIVE ':' b_variables constant
        | TIMER FAST ':' f_variables t_constant
        | TIMER SLOW ':' s_variables t_constant
        | ARRAY '(' INT '.''.' INT ')' OF a_type ;

a_type: BOOLEAN ':' ba_variables constant | INTEGER ':' ia_variables constant
        | REAL ':' ra_variables constant
        | BOOLEAN RETENTIVE ':' ba_variables constant ;

var : I | R | DI | DO | B | WI | WO | BA | IA | T | DX | RA | TA | F | AI | FT | ST | DF ;

b_variables: var |b_variables ',' var ;
i_variables: var |i_variables ',' var ;
r_variables: var |r_variables ',' var ;
ba_variables: var |ba_variables ',' var ;
ia_variables: var |ia_variables ',' var ;
ra_variables: var |ra_variables ',' var ;
f_variables: var |f_variables ',' var ;
s_variables: var |s_variables ',' var ;
```

```
constant: /*empty */ | ':' '=' val;

t_constant:/* empty */ | ':' '=' NUM ',' NUM ',' bool ',' bool ; val: NUM | bool;

initialisations: /*empty */ | INIT initialisation ;

initialisation: statement ';' init ;

init: /* empty */ | init statement ';' ;

body: /* empty */ | BODY statements ;


statements: /* empty */ |first_statement ';' statement_list ; first_statement: assignment

        | if_statement | while_loop ;


statement_list: /* empty */ | statement_list statement ';' ; statement: assignment

        | if_statement | while_loop ;


assignment: value ':"=' p_exp | f_value | t_value | t_value ':"=' p_exp

        | recipe ':"=' recipe | r_value ':"=' p_exp | r_value | AI ':"=' p_exp

        | AI '.' ai_value ':"=' p_exp | v_value | v_value ':"=' p_exp | sfc_value

        | sfc_value ':"=' p_exp | procedure ;


sfc_value: unit '.' ABORT | unit '.' ENABL ;

unit: S_TEST | SELFTEST ;

time: PROGRAM '.' IHOUR ;


r_value: recipe '.' RTU |recipe '.' INUSE |recipe '.' DSTBL |recipe '.' DRDY

        |recipe '.' STATUS |recipe '.' L_LIMIT |recipe '.' H_LIMIT |recipe '.' LL_LIMIT

        |recipe '.' HH_LIMIT |recipe '.' BAD_XMT_LIM |recipe '.' HI_LIM

        |recipe '.' XMT_LOW |recipe '.' XMT_HIGH |recipe '.' OPTIC_LO

        |recipe '.' OPTIC_HI |recipe '.' HIHI_LIM | UNLOCK '(' recipe ')'

        | CLEAR '(' recipe ')' | SELECT '(' recipe ')' ;


 recipe: TRIP_ALL | TRIP_LLL | TRIP_HH | TRIP_LL | TRIP_LIM | CODELL

        | SING_LIM | DUAL_LIM ;
```

```
f_value : LATCH ’(’ flag ’)’ | CLEAR ’(’ flag ’)’ | ON ’(’ flag ’)’ ;
flag :F | DF ;
e_flag :DF | F ;
 value : I | R | DI | DO | B | WI | WO | BA | BA ’[’ exp ’]’ | IA | IA ’[’ exp ’]’ | T text
        | T ’[’ exp ’]’ | DX | DX ’[’ exp ’]’ | RA | RA ’[’ exp ’]’ | TA | TA ’[’ exp ’]’ | e_flag ;

text: /* empty */ | ’.’ TEXT1 | ’.’ TEXT2 | ’.’ TEXT3 ;
ai_value: FTAU /* which can be assigned*/ | RAW | SRV ;
t_value: DELAY timer |timer ’.’ TCC |timer ’.’ TCP |timer ’.’ ENABL |timer ’.’ RESET
        |timer ’.’ TOUT ;

timer: FT | ST ; v_value: valve ’.’ CMMD | valve ’.’ OPENC | valve ’.’ CLSC
        | valve ’.’ OPND | valve ’.’ CLSD | valve ’.’ TRVL | valve ’.’ OSL | valve ’.’ CLS
        | valve ’.’ FTO | valve ’.’ FTC | valve ’.’ FAILD | valve ’.’ CLSTO
        | valve ’.’ OPNTO | valve ’.’ DSBLD | valve ’.’ LOCKD | valve ’.’ NRDY
        | valve ’.’ MOPEN | valve ’.’ OURDO | valve ’.’ OURDC | valve ’.’ STATUS
        | valve ’.’ VFLAG | valve ’.’ OTCP | valve ’.’ OTCC | valve ’.’ CTCP
        | valve ’.’ CTCC | LOCK ’(’ valve ’)’ | UNLOCK ’(’ valve ’)’ | OPEN ’(’ valve ’)’
        | CLOSE ’(’ valve ’)’ | RESET ’(’ valve ’)’ ;

valve: VSS | VDD ; exp:value | t_value | time | sfc_value | AI | AI ’.’ ai_value | v_value
        | r_value | NUM | bool |’(’ exp ’)’ | exp ’+’ exp | exp ’-’ exp | exp ’*’ exp | exp ’/’ exp
        | INCREMENT I | DECREMENT I | exp MOD exp | exp AND exp
        | exp OR exp | exp XOR exp | NOT exp | exp ’*”*’ exp | exp ’<’ exp
        | exp ’<”=’ exp | exp ’>’ exp | exp ’>”=’ exp | exp ’=’ exp | exp ’<”>’ exp | function ;

p_exp:value | t_value | time | sfc_value | AI | AI ’.’ ai_value | v_value | r_value | NUM
        | bool |’(’ p_exp ’)’ | p_exp ’+’ p_exp | p_exp ’-’ p_exp | p_exp ’*’ p_exp
        | p_exp ’/’ p_exp | INCREMENT I | DECREMENT I | p_exp MOD p_exp
        | p_exp AND p_exp | p_exp OR p_exp | p_exp XOR p_exp | NOT p_exp
        | p_exp ’*”*’ p_exp /* | p_exp ’<’ p_exp | p_exp ’<”=’ p_exp | p_exp ’>’ p_exp
        | p_exp ’>”=’ p_exp | p_exp ’=’ p_exp | p_exp ’<”>’ p_exp */ | function ;
```

```
procedure: UNPACK_BITS '(' exp ',' exp ')' | PACK_BITS '(' exp ',' exp ')'
        | BCDBIN '(' exp ',' exp ')' | BIT_ASSIGN '(' exp ',' exp ',' exp')'
        | BITCLEAR '(' exp ',' exp ')' | BITSET '(' exp ',' exp ')'
        | LOAD_ARRAY '(' exp ',' exp ')' ;

function: BITS_TO_INT '(' exp ')' | BITTEST '(' exp ',' exp ')' | EDGE '(' exp ')' ;
c_function: BITS_TO_INT '(' exp ')' | BITTEST '(' exp ',' exp ')' | EDGE '(' exp ')' ;
bool: TRUE | FALSE ;
while_loop: WHILE condition LOOP statements END LOOP ;

if_statement: IF condition THEN statements endif ;
endif: ENDIF | ELSIF condition THEN statements endif | ELSE statements ENDIF
condition: c_exp ;
bracket: c_exp ;

c_exp: c_value | c_t_value | time | sfc_value | AI | AI '.' ai_value | c_v_value
        | c_r_value | NUM | bool |'(' c_exp')' | c_exp'+' c_exp | c_exp'-' c_exp
        | c_exp'*' c_exp | c_exp'/' c_exp | INCREMENT I | DECREMENT I
        | c_exp MOD c_exp | c_exp AND c_exp |c_exp OR c_exp
        | bracket XOR bracket | NOT c_exp | c_exp'*''*' c_exp | c_exp'<' c_exp
        | c_exp'<''=' c_exp | c_exp'>' c_exp | c_exp'>''=' c_exp | c_exp'=' c_exp
        | c_exp'<''>' c_exp | c_function ;

c_v_value: valve '.' CMMD | valve '.' OPENC | valve '.' CLSC | valve '.' OPND
        | valve '.' CLSD | valve '.' TRVL | valve '.' OSL | valve '.' CLS | valve '.' FTO
        | valve '.' FTC | valve '.' FAILD | valve '.' CLSTO | valve '.' OPNTO
        | valve '.' DSBLD | valve '.' LOCKD | valve '.' NRDY | valve '.' MOPEN
        | valve '.' OURDO | valve '.' OURDC | valve '.' STATUS | valve '.' VFLAG
        | valve '.' OTCP | valve '.' OTCC | valve '.' CTCP | valve '.' CTCC
        | LOCK '(' valve ')' | UNLOCK '(' valve ')' | OPEN '(' valve ')'
        | CLOSE '(' valve ')' | RESET '(' valve ')' ;
```

```
c_t_value:/* DELAY timer |*/ timer '.' TCC |timer '.' TCP |timer '.' ENABL
        |timer '.' RESET |timer '.' TOUT ;

c_r_value: recipe '.' RTU |recipe '.' INUSE |recipe '.' DSTBL |recipe '.' DRDY
        |recipe '.' STATUS |recipe '.' L_LIMIT |recipe '.' H_LIMIT |recipe '.' LL_LIMIT
        |recipe '.' HH_LIMIT |recipe '.' BAD_XMT_LIM |recipe '.' HI_LIM
        |recipe '.' XMT_LOW |recipe '.' XMT_HIGH |recipe '.' OPTIC_LO
        |recipe '.' OPTIC_HI |recipe '.' HIHI_LIM | UNLOCK '(' recipe ')'
        | CLEAR '(' recipe ')' | SELECT '(' recipe ')' ;

c_value : I | R | DI | DO | B | WI | WO | BA | BA '[' exp ']' | IA | IA '[' exp ']' | T text
        | T '[' exp ']' | DX | DX '[' exp ']' | RA | RA '[' exp ']' | TA | TA '[' exp ']' | e_flag ;
```

APPENDIX III

MAPPING DOCUMENT

TABLE OF FIGURES

1. PLC APT OVERVIEW

1.1 INTRODUCTION

"A program in the APT is that portion of the process that can run on a single controller. The actual size of the program depends on controller memory size, safety considerations and other characteristics of the process line." [1]

The code that will be translated is written using three different languages using the APT system. Only the code that was used in the two programs (ESD and F&G) will be discussed in this mapping document. The code was written using three programming languages which combine together at compile time to produce RLL.

The continuous function blocks can be of type interlock, which has high or low priority or math blocks - only the active type was used; which means that they cannot be turned off.

1.2 BUILD ROUTE

When the software is compiled it is ordered depending on the block type and is in the following order:

1. System Logic (APT usage)
2. Interlocks (high priority)
3. SFC controllers
4. SFC transitions
5. SFC Steps
6. Flags
7. Device logic / CFB activation logic
8. CFB math logic / interlocks

   end of each scan

9. APT generated RLL subroutines

It is not known if the units are kept together, but for the purpose of this translation it will be assumed that they are. This should aid in the understanding of the translated code.

1.3 HARDWARE

The hardware of the controller has many features - these include:-

- the RLL program is stored in memory.
- interrupt I/O allows for fast reactions to external events *
- it supports a redundant remote base controller *
- immediate I/O updates allows the application program to access an I/O point multiple times during a scan.
- cyclic RLL allows the creation of an additional RLL program that runs independently to the main program.

- it allows external subroutines that can be written in a high level language (although ESD and F&G code does not seem to have used this function.)
- has PID (proportional integral derivative) loops for batch control
- the CPU contains a real time clock that contains a 2 digit year, month, day of month and day of week. The hour, minutes, second, tenth and hundredth of a second. This information can all be read by the program, only the hour though is actually accessed.
- The translated code does not use these facilities. The inputs are read into a buffer at the start of each cycle and that value is then used for the cycle.

2. STATIC DOS STRUCTURE OF FILE STORAGE

The programs that are written using the APT are stored as a directory tree structure in DOS. The programs are all stored in a directory called "**program**"; the leaves of this directory are the programs that are within the APT in this configuration. On moving into the directory of one of the programs there is a sub directory called "**units**". Within "units" there is a leaf directory for each of the units that have been declared within the program, (a unit is basically dividing a program into modules). Within a unit there can either be CFCs (continuous function charts) or SFCs (sequential function charts) or both. There are many sub directories but the important ones are "**cfc**" and "**sfc**" which contain the charts that are used to write the program.

In the "cfc" directory there is a directory for each of the CFCs implemented in the unit. Within the directory there is a file called "**graphic**" which gives the positioning and number of CFB (continuous function blocks) that are found within that CFC. The math language that is contained in the blocks is stored in individual text files with an extension ".mth" for the commented version and ".mt_" for the non-commented version. These are called by the same name as the CFB in the program.

It should be noted that the programs stored within the APT at a given time can be totally unrelated.

Directory structure in BNF form

```
<pathname> ::=
program\<progname>\units\<unitname>\<chart>

<chart> ::=     cfc\<cfcname>\textfile
                | sfc\<sfcname>

<progname> ::= <identifier>

<unitname> ::= <identifier>

<cfcname> ::= <identifier>

<sfcname> ::= <identifier>.sfc

<cfcname> ::= graphic | <identifier>.mth | <identifier>.mt_
```

Figure 2:1
BNF form of the directory structure

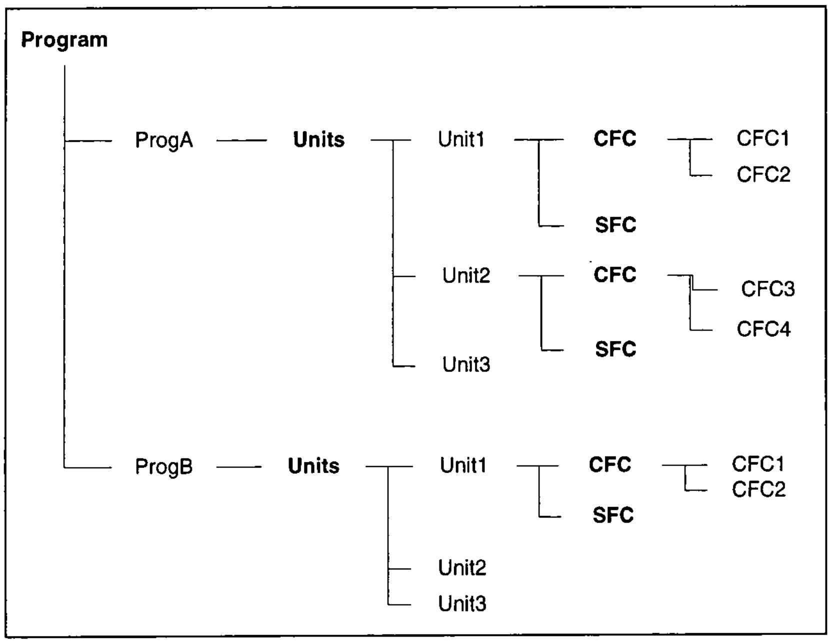


Figure 2:2
DOS file structure demonstrating key words

```
Directory PATH listing
Volume Serial Number is 0837-18EF
C:.
+---UNITS
¦  +---TEST1
¦  ¦  +---SFC
¦  ¦  +---CFC
¦  ¦  ¦  +---CHARTA
¦  ¦  +---PRR
¦  ¦  +---DEBUG
¦  +---TEST
¦     +---SFC
¦     +---CFC
¦     ¦  +---CHART1
¦     ¦  +---CHART2
¦     ¦  +---CHART3
¦     +---PRR
¦     +---DEBUG
+---PGMSUB
+---PGMSFC
+---PGMCFC
+---PRR
+---MAKE
+---DEBUG
+---SYMTABLE
+---MAITT
```

Figure 2:3
The directory tree of an APT program

2.1 CFC FILE STRUCTURE

ESD and F&G code only used Interlocks and Math Blocks. This implies that each block can be individually translated into a WSL procedure since each block contains a math language piece of code, (many of the CFBs that were not used have predefined meaning.) The CFB names can be determined from the file names in DOS. The file "graphic" in the CFC directory identifies the number of CFB, its type, its name and its position on the screen. If connecting CFBs had been used then the positions of the interconnections would also be included in this file. The math language files are stored as ".mth" files with comments and .mt_ without any comments. If the block of code contains no comments then it is not stored in mth format. When the APT creates the mt_ file it also includes abbreviations about the code.

File "graphic"	
5 51 0 720 PI35051 51 120 720 PI35053 51 240 720 LI35039 51 360 720 LI35042 51 480 720 LI35033	**5** indicates that there are 5 CFB. **51** indicates it is of type interlock **27** (where the 51 is would indicate a math function) two numbers after the 51 are the screen position of the CFB. Last item on the line is the CFBs name in ASCII

Figure 2:4
A sample graphic file

Each CFB stored in a separate file will be translated individually into a WSL, then the files will be combined as procedures into one piece of code. The type of CFB is either a high interlock, low interlock or an active mathblock. All the math blocks that have been used are active, which means that they cannot be disabled.

The initial part of each of the CFBs will be called first and then the main part of the CFC will be called on the second and future loops around the program.

2.2 SFC FILE STRUCTURE

All the code is stored in one file “sfc_name.sfc” this file needs to be divided into individual steps so that it can be translated, and then translated code recombined in WSL format. The SFC information is stored within the SFC directory in its' unit directory. The splitting of the code into steps and the translation was managed by Perl scripts. The recombination of the SFC after translation was more complex and required human intervention so a C program was developed to perform this task.

3. PROGRAM LAYOUT

High priority interlocks are executed before SFCs, other CFBs and devices: low priority interlocks and math blocks are executed after the SFC. (High level interlocks are commonly used for internal validation checks.) The ordering of the high priority interlocks is arbitrary and can not be defined. The random ordering will be maintained by using procedure calls to units and procedure calls to SFCs, CFCs and CFBs. This will allow the procedures to be moved if desired. If the code had been inserted into the main body then the block like structure would have been lost. The method of determining high and low priority is defined in 6.1

There is a main procedure of the program in WSL that calls each type of block, this then calls all the units and they in turn call all the relevant blocks. The original function templates were generated by a program that walks around the directory tree structure in DOS. The initial procedures called each of the units in the order that they are stored in DOS. Each of the units will call the relevant CFBs identified within the sub directories.

The main procedure calls a set of functions, which in term calls the CFBs that are stored within the units so that the procedures do not become to long. The main procedures are:

• **init_high_interlock(var)**	initial part of the high priority interlocks
• **init_low_interlock(var)**	initial part of the low priority interlocks
• **init_active_mathblock(var)**	initial part of the active math blocks
• **body_high_interlock(var)**	body of the high priority interlocks
• **sfc(var)**	the SFC in each program
• **body_low_interlock(var)**	body of the low priority interlocks
• **body_active_mathblock(var)**	body of the active math blocks

The last four functions calls are in a loop since they will be expected to execute continually. At the end of the code there is a procedure that is called after every time a timer is used. This is to simulate the changing of the value due to an external device.

Following is the initial block of code for the translated WSL program. Note italics are comments to aid the reader and do not appear in the code.

```
Var
FALSE := 0,
TRUE := 1,

the constants that are used as array elements for some of the translated variable
types.

Flags
LATCH := 1,
ON := 0,
Timers
RESET := 1,
ENABLE := 2,
TCC := 3,
TCP := 4,
TOUT := 5,

Inputs
RAW := 1,
SRV := 2,
FTAU := 3,

valves
CMMD := 1,
OPENC := 2,
CLSC := 3,
OPND := 4,

TRVL := 6,
OLS := 7,
CLS := 8,
FTO := 9,
FTC := 10,
FAILD := 11,
CLSTO := 12,
OPNTO := 13,
DSBLD := 14,
LOCKD := 15,
NRDY := 16,
MOPEN := 17,
ORRD := 18,
CLSD := 19,
VFLAGS := 20,
OTCP := 21,
OTCC := 22,
CTCP := 23,
CTCC := 24,

Recipes
RTU := 1,
INUSE := 2,
DSTBL := 3,
DRDY := 4,
STATUS := 5,
UNLOCK := 6,
CLEAR := 7,
SELECT := 8,
L_LIMIT := 9,
H_LIMIT := 10,
LL_LIMIT := 11,
HH_LIMIT := 12,
BAD_XMT_LIM := 13,
HI_LIM := 14,
XMT_LOW := 15,
XMT_HIGH := 16,
OPTIC_LOW := 17,
OPTIC_HI := 18,
HIHI_LIM := 19:

begin
comment: "Flags";
comment: "TIMER";
comment: "ANALOGUE INPUT";
comment: "VSS AND VDD2";
```

```
comment: "RECIPIES";

comment: "start of code";
```

Initialising the CFBs

```
init_high_interlock(var);
init_low_interlock(var);
init_active_mathblock(var);

do
comment: "do loop that will run continually for the whole program.  ";

body_high_interlock(var);
sfc(var);
body_low_interlock(var);
body_active_mathblock(var)
od;
comment: "end of the continual loop and the program"
```

An example of the rest of the code

```
where
        proc init_high_interlock(var) ==
                begin
                unit1(var);
                unit2(var)
                where
                        proc unit1(var) ==
                                begin
                                cfc_cfb(var)
                                where
                                proc cfb_cfc(var) == ..............     .
```

```
                                  end     .

                                       .

                                       .

                        end     .
                proc init_low_interlock (var) == ..................     .
.

.

.

```

The procedure to represent the counting down of time by the timer.

```
            proc  timer_set(var timer) ==
                comment:" this procedure simulates the behaviour of a timer";
                if ((timer[RESET] = TRUE) and (timer[ENABL] = TRUE)) then
                    timer[TCC] := timer[TCC] -1;
                    comment:"the timer is counting down"
                elsf ((timer[RESET] = TRUE) and (timer[ENABL] = FALSE)) then
                    timer[TOUT] := TRUE;
                    comment:" timer remains inactive but TCC value remains
                        where it was "
                else
                    timer[TCC] := timer[TCP];
                    timer[TOUT] := FALSE;
                    comment:" timer remains inactive and tout remains false. "
                fi;
                if (timer[TCC] = TRUE) then
                    timer[TOUT] := TRUE
                fi    .
end;
comment: "now the end of the constants being declared"
end
```

Figure 3:1
Sample layout of the entire program

3.1 PROCEDURE CALLS IN WSL

The units and CFBs are internal procedure calls where no value is passed. The procedures/ functions that are supplied by the APT were mapped as external procedure or function calls. Internal procedures and functions are also used. With a procedure there must be the keyword var present. Before var is the list of variables that do not change value during the procedure and after var are those variables that do change value.

The external procedure and function call is a WSL feature that allows the assumption that the procedure / function is declared in another part of the code. This feature is so that subsets of the code can be read into Maintainers Assistant the transformation tool. This was used for all procedures and functions that are pre defined by the APT. Below are all the possible procedures / function calls that were used in the translated code.

Internal Procedure /function Call

```
procedure_name(nochange1, nochange2 var change1)
function_name(var_list)
```

External Procedure / function Call

```
!p procedure_name(var)
!f function_name()
```

Internal Procedure Declaration

```
proc procedure_name(var) == .........        .
proc procedure2(var) == ................     .
```

Figure 3:2
An example WSL procedure calls

4. MATH LANGUAGE → WSL

The statement types and ordering are similar in the math language and WSL: both languages have comments, assignment and conditionals. The differences arise with the fact that WSL does not have types, and has no concept of Booleans, timers, flags and variables that can contain more than one variable.

Keywords and variables in the math language are not case sensitive, i.e. 'LATCH' or 'latch' are identical. Variables in the APT languages are allowed to start with a number but in WSL they are not so all WSL variables will be converted to capitals with WSL_ placed in front of them. The keywords in WSL are lower case letters.

5. LAYOUT

All of the math blocks (steps in SFCs) follow a strict layout of the program including keywords to define structures. The math block was translated into a WSL procedure, split up into body and init. The mathblock is set up so that if there is only a body portion and no initial part then the BODY word omitted. If there is no body but an initial part then the INIT keyword is used and not the BODY keyword. If both parts are included then both words are used.

At the top of an SFC step there is the keyword MATH before any of the math language, all the statements before this keyword are executed in parallel. The keywords PRAGMA "RLL" indicate that the program is to compile to ladder logic but they do not force compilation. The compile type is forced by the types of functions that are used and if functions that cannot be compiled to RLL are used then the code will not be compiled to RLL. All math language statements start with the keyword BEGIN.

Math block layout	**WSL procedure layout**
<mathblock> ::= [PRAGMA {"RLL"};]	
[<Declarations>]	
BEGIN	
[INIT	
<initialisations>]	
[BODY	
<body portion>]	
e.g.	e.g.
{A low priority interlock called A}	proc low_int_A() ==
PRAGMA ("RLL");	comment: "Comment at top of CFB";
Begin	comment: "PRAGMA {''RLL''};";
I_200 := (hs_20405);	comment: "Begin ";
I_201 := (hs_20405);	
	I_200 := (hs_20405);
IF NOT (Red_test_ovr) THEN	I_201 := (hs_20405);
A := B;	
C := B;	if not (Red_test_ovr = 1) then
ENDIF;	A := B;
	C := B
	fi
	.
	Note that this is the format for the body part f the code if there is an initialisation then it will go in a different procedure.

Figure 5:3
Assignment format in Math language and WSL

BEGIN	BEGIN	BEGIN	BEGIN
body portion	INIT	BODY	INIT
	initialisations	body portion	initialisations
			BODY
			body portion

Figure 5:1
Different orderings of keywords in a math block

In a CFB the INIT part of the code is executed once and only once. In a SFC the INIT part of the code is executed once each time through the SFC, but only once per time the step is active. The CFB INIT and MAIN parts are translated into different procedures and so are called separately.

5.1 DECLARATIONS

Declarations occur at the top of a function block and the variables are local to that function block. Only variables of type Boolean, retentive Boolean, integer, real, array or a timer (fast or slow) can be declared in the declaration section; they may be initialised. Arrays can be of any of the above types except timers. Retentive Booleans are not used within the analysed code. The other method of declaring variables is in tables and are global to either the entire program or an entire unit. The format of tables is discussed in chapter 6.

WSL does not support typed declarations and variable types so this part will be converted into comments. Variables in the math language can be of no more than 12 characters and must be at least one character long. They can contain an underscore and must contain a letter, unusually though they can start with a number. All variable names and types were added to a symbol table, for type checking and manipulation of the code.

Math block Declarations	**WSL Declaration**
<declarations> ::= <declaration> ; {<declaration>;} <declaration> ::= <type> : <variable> {,<variable>} [:= constant] I TIMER <ttype>: <variable> {,<variable>} [:= <int> <int> <bool> <bool>] I ARRAY (<num> .. <num>) of <type> : <variable> {<variable>} [:= <constant>] <type> ::= integer I Boolean retentive I Boolean I real <ttype> ::= fast I slow The timer information, the first integer is the current value of the timer, the second integer is the preset value of the timer, the first Boolean enables the timer while the fourth resets the timer.	WSL does not support typed declarations so the declaration in PLC format will be inserted as a comment.

e.g
 	
Boolean: button;
Integer: count;
Integer: count1;
	e.g.
	
	comment: " Variable declaration information as in the math language."
	comment:" Boolean: button";
	comment:" Integer: count ";
	comment:" Integer: count1 ";

Figure 5:2
Declarations in Math language and WSL

5.2 INIT

The math language code in the initial part of the block is only executed on the first iteration. It can contain any statements that the body of the code can contain.

Math block INIT	WSL procedure
<initialisations>::= <initialisation> {<initialisation>}	
<initialisation> ::= <variable> := <constant> ;	
<constant> ::= <Boolean> | <integer> | <real>	
e.g. INIT	e.g. proc cfc_cfb () ==

A := 3;
B:=2;
	A := 3;
	B:=2
	.

Figure 5:3
INIT in Math language and WSL

5.3 BODY

The body of the code is executed on all executions of the loop after the initial one. The body is either empty or a list of statements; these statements are comments, assignments, procedures or conditionals.

Math block Body	**WSL procedure**
<body portion> ::= <statements> <statements> ::= <statement> {<statement>} <statement>::=< comment> | <assignment statement> | < procedure statement> | <conditional> | <function statement> | <command statement> |<while>	

e.g.
 	
BODY
 	
IF (A = 1) THEN
A := 3;
B:=2;
ENDIF
	e.g.
	
	proc cfc_cfb () ==
	this procedure is called from within the procedure representing the units
	
	if (A = 1) then
	A := 3;
	B:=2
	fi

Figure 5:4
Body in Math language and WSL

5.4 COMMENTS

PLC comments become WSL comments but since the WSL delimiters are “” all “ were converted into a double ‘ which will look like ‘’. This should avoid confusion between “ and ‘ already in the comment.

Keywords that are no longer required are maintained by putting them into comments. This will be performed on the following words:

- PRAGMA("RLL");
- BEGIN
- INIT
- MATH (only used in SFC)

The comments about variables, CFB, units and the program can only be determined from the database file see chapter 6

Math block comment	**WSL comment**
<comment> ::= "{" <characters> "}" | (* <characters> *)	<comment> ::= comment: "<characters>"
e.g. { This is a comment} or (* This is a comment *)	e.g. comment: "This is the comment"

Figure 5:5
A math block comment and the WSL comment

5.5 TYPES

There are five types that can be declared in the CFB; integers, reals, Booleans, arrays, flags and timers. The rest of the variables are declared in tables outside the main programming environment and are stored in binary files see chapter 6.

5.5.1 Integers

Integers are one of the basic types in the math language; they use 16 bit words and 16 bit intermediate arithmetic. This means that the values of the integers are between -32768 and 32767. There are various operations allowed with integers all of which are also allowed in the WSL language.

Math language for integers

```
<integer>::= <sign> <unsigned integer> | <unsigned integer> |
        #2# <base 2 integer> | #16# <base 16 integer> |
        <sign> #2# <base 2 integer> |
        <sign> #16# <base 16 integer>

<sign> ::= + | -

<unsigned integer> ::=  <digit> {<digit>}
<digit> ::= 0 | 1| 2| 3 | 4 | 5 | 6 | 7 | 8 | 9

<base 2 integer> ::= <2digit> {<2digit>}
<2digit> ::= 0  1

<base 16 integer> ::= <16digit> { <16digit>}
<16digit> ::= <digit> | A | B | C | D | E | F
```

Figure 5:6
Integers defined

Math language integer operators		WSL operators	
Increment	INCREMENT a	Increment	a = a + 1
Decrement	DECREMENT a	Decrement	a = a - 1
Multiply	*	Multiply	*
Division	/	Division	/
Modulus	MOD	Modulus	mod
Addition	+	Addition	+
Subtraction	-	Subtraction	-
Logical AND	AND	Logical AND	and_bit
Exclusive OR	XOR	Exclusive OR	xor_bit
Logical OR	OR	Logical OR	or_bit

Figure 5:7
Integer operators

5.5.2 Reals

The value of a real is between -9.223372E^{+18} to 9.223372E^{+18}, although a value between -2.710501E^{-20} to 5.421011E^{-20} except 0.0 gives a control error. Although there is no declaration in WSL the assignment of real values is possible in WSL. When arithmetic is performed using real numbers the answer is always a real number.

Math language for reals

```
<real> ::= [<sign>] <unsigned integer>.unsigned integer> [E <sign>]
        <unsigned integer>]
```

Figure 5:8
Reals defined

Math language real operators		WSL operators	
Power	**	Power	**
Multiply	*	Multiply	*
Division	/	Division	/
Addition	+	Addition	+
Subtraction	-	Subtraction	-

Figure 5:9
Real operators

5.5.3 Booleans

There is no such thing as a Boolean in WSL; this means that they are assigned integer values 0 or 1. The convention is to assign true the value 1 and false the value 0. In conditional statements in PLC a Boolean can be just a variable. For a conditional comparison in WSL this will have to be converted to the following:-

- variable = false
- variable = true.

Since true and false are assigned values at the top of the WSL code. Booleans in the math language have the operators XOR, OR, NOT, AND although there is no such thing as Booleans in WSL, in conditionals, AND, OR and NOT are allowed. There is no equivalent of an XOR in WSL but it maps to the equation (a XOR b) to ((a or b) and (not(a = b))).

Math language for and Boolean

<Boolean> ::= <true> | <false>

<true> ::= 1 | on | true

<false> ::= 0 | off | false

Figure 5:10
Booleans defined

Math language Boolean operators		WSL operators	
Logical NOT	NOT	Logical NOT	TRUE xor_bit
Logical AND	AND	Logical AND	and_bit
Logical OR	OR	Logical OR	or_bit
Logical XOR	XOR	Logical XOR	xor_bit

Figure 5:11
Boolean operators

Relational operators are allowed with integers, reals and Booleans the result is a Boolean.

Math language integer operators		WSL operators	
less than	<	less than	<
less than or equal	<=	less than or equal	<=
greater than	>	greater than	>
greater than or equal	>=	greater than or equal	>=
equal	=	equal	=
not equal	<>	not equal	<>

Figure 5:12
Integer operators

5.5.4 Flags

Flags have the value of either on (true) or off (false). All the references to a flag are logically connected and the value of the flag can only be set in one place in the compiled program. The commands used to assign the flags are:-

- Clear - false - sets the value to false and remains false.
- Latch - true - sets the flag to on until there is a clear command
- On - true - sets the flag to on while the SFC/ CFB is active

When a flag is set to 'On' it remains true while the block is still active. At the end of a CFB or Step it is turned off and does not automatically come 'On' again when the block becomes active. In the case of an SFC the flag is 'On' while the step is active and false once the step becomes inactive. In the case of a safe SFC being called while the flag is 'On' the flag remains true until control returns to the main SFC and the previous step is then set inactive. (See 7 for information about SFCs)

.

Math language flag assignment	**WSL assignment**
<f_assign> ::= LATCH (<flag_variable>) | CLEAR (<flag_variable>) | ON (<flag_variable>)	<f_assign> ::= <flag_variable>[LATCH] := TRUE ; <flag_varibale>[ON] := TRUE | <flag_variable>[LATCH] := FALSE ; <flag_varibale>[ON] := FALSE | <flag_varibale>[ON] := FALSE
e.g.	e.g.

LATCH (var3)	var3[LATCH] := TRUE;
	var3[LATCH] := TRUE

Figure 5:13
BNF form of flag assignment

At the end of each mathblock the flags that have been used have to be tested to see if they have been turned on. If this is the case they have to be turned back off using the following WSL code. This is forced by the APT compiler so it was inserted with each procedure. A list of all flags used in the block was stored by the translator so as to be able to insert the code at the end of the WSL block.

```
if ((WSL_DFI_11A02_HR[LATCH] = FALSE) and
                (WSL_DFI_11A02_HR[ON] = TRUE)) then
        WSL_DFI_11A02_HR[ON] := FALSE;
fi
```

Figure 5:14
A sample end piece of code of a math block containing flags

5.5.5 Timers (type ST or FT)

A timer allows a delay to be set up within a SFC step or a CFB. They are either fast or slow and count down at that specific rate from the preset value. Timer values change during the execution of the program. When used in the SFC they have to run faster than the execution cycle of the program.

There are various values and keywords associated with timers; they are:-

- DELAY :- starts the timer counting as soon as the step is active and sets current value = preset value and .tout to false
- .TCC :- is the current timer count (i.e. current value) (read only integer)
- .TCP :- preset value count (for slow timer 0.1 * value) (read / write integer)
- .ENABLE :- indicates the timer has been activated (read / write Boolean)
- .RESET :- becomes false to indicate the current counter is reset to the preset value, true indicates that the timer can be activated if .enable is set to true. (read /write Boolean)
- .TOUT :- true when current counter = 0 false when current counter ≠ 0 (read only Boolean)

DELAY is used to set the timer counting which is done as soon as the step is active. For the timer to start counting the .reset and .enable values must both be true.

When .reset is false the current timer count .tcc equals the preset false, .tout remains false and the timer remains inactive.

When .reset is true and .enable is false the timer becomes inactive, but .tcc is not reset and .tout remains true. To manipulate the timers either the delay command can be used or the extension can be manipulated directly. The method of setting a timer should be consistent through the life of a timer.

Math language timer assignment/ conditional	**WSL assignment**
<timer_assignment> ::= DELAY (<timer>)	<timer_assignment> ::= !P DELAY (<timer>)
l <timer>.tcc	l !P TCC (<timer>)
l <timer>.tcp	l !P TCP (<timer>)
l <timer>.enabl	l !P ENABLE (<timer>)
l <timer>.reset	l !P RESET (<timer>)
l <timer>.tout	l !P TOUT (<timer>)
if last 5 are being assigned a value then of the format:-	**if last 5 are being assigned a value then of the format:-**
<timer>.enabl := bool	<timer> "[" enabl "]" := bool
e.g.	e.g
DELAY (timer_3)	!P DELAY(timer_3,null,R)
if (timer_3.tcc)	if (!P TCC(timer_3,null,R) .....
timer_5.reset := true	!P RESET(timer_5,true,W)

Figure 5:15
BNF form of a timer assignment

After each time a timer is read or written to during the code in the WSL there is a procedure call to timer_set which will emulate the behaviour of the timer counting down. This procedure is declared at the end of the code.

```
proc  timer_set(var timer) ==
          comment:" this procedure simulates the behaviour of a timer";
          if ((timer[RESET] = TRUE) and (timer[ENABL] = TRUE)) then
               timer[TCC] := timer[TCC] -1;
               comment:"the timer is counting down"
          elsf ((timer[RESET] = TRUE) and (timer[ENABL] = FALSE)) then
               timer[TOUT] := TRUE;
               comment:" timer remains inactive but TCC value remains
                    where it was "
          else
               timer[TCC] := timer[TCP];
               timer[TOUT] := FALSE;
               comment:" timer remains inactive and tout remains false. "
          fi;
          if (timer[TCC] = TRUE) then
               timer[TOUT] := TRUE
          fi   .
```

Figure 5:16
The timer set function

5.5.6 Arrays

An array is an indexed collection of values that can be referenced as a whole or individual value. Boolean, integer and real arrays can be assigned in the declaration section, the rest must be declared in tables see chapter 6. Arrays can be assigned as a whole e.g. array1 := array2 in this instance the whole array will be copied so the array's must be the same size and of a compatible type.

Math language array assignment	**WSL assignment**
An array can be assigned to equal another array;-	
<array> ::= <name> ['[' number']'] array := array **Note the arrays are of the same size**	<array> ::= <name> ['[' number']'] array := array
array[n] **the nth bit of the array will be accessed for assignment or reading**	array[n]
e.g.	e.g

array1 := array2
array2[2] := true
array3[10] := array4[4]
	array1 := array2
	array2[2] := true
	array3[10] := array4[4]

Figure 5:17
Array assignment

5.5.6.1 Integer Array (type IA)

An integer array is an indexed collection of integers, they are indexed 1→ 7 if there are 7 items in the array.

5.5.6.2 Boolean Array (type BA)

A Booleans array is an indexed collection of Boolean values, i.e. assigned true or false.

5.5.6.3 DO10 Array (type DX)

A DO10 array is a Boolean array with length 10 that is translated to PCS as a DO10 tag type.

5.5.6.4 Text Array (type TA)

This is an array of text variables, each element is 30 characters longs and is primarily used for PCS tags. Assignment to the text is not allowed in the coding and must be pre assigned. The array can only be assigned by literal values. Although variables of this type were declared they were not used.

5.5.6.5 Text (type T)

Text is similar to an array but it is a string variable that will contain text and is 1,2 or 3 fields long. Each field contains 30 characters of text. Text can not be written and assigned during the program although text can be copied from one text string to the other. Although variables of this type were declared they were not used.

Math language text assignment	**WSL assignment**
<text_assign> ::= <text_variable> . text1	<text_assign> ::= <text_variable>[1]
| <text_variable> . text2	| <text_variable>[2]
| <text _variable> . text3	| <text_variable>[3]
| <text_variable> [<int(1|2|3)>]	
e.g.	e.g
array2.text1 := array3.text2	array2[1] := array3[2]
array_4[2] := array_5[1]	array_4[2] := array_5[1]

Figure 5:18
Text assignment

5.5.7 Input /Output

There are six types of inputs and outputs that are used in the code.

5.5.7.1 Analogue Input (type AI)

These inputs are used for measuring flow meters, pressure transmitters, etc. The analogue input contains 4 different values that can be read, one of which can also be written to:-

• name	- filtered real value (read only)
• name.RAW	- integer input in module form (read only)
• name.SRV	- raw scaled value real pre filter (read only)
• name.FTAU	- time constant rate for filter (read/ write) real or integer

The filter was not used in the translated code.

Math language Analogue input expression	**WSL assignment**
<ai_assign> ::= < ai> I <ai> . RAW I <ai>. SRV I <ai>. FTAU e.g. if (analouge_in.RAW < another_variable) Bitassign(analouge_in,6,var)	<ai_assign> ::= <ai> "[" 0 "]" I <ai> "[" RAW "]" I<ai> "[" SRV "]" I <ai> "[" FTAU "]" e.g if((analogue_in[RAW])< another_variable) !p Bitassign(analogue_in[0], 6, var) **(Bitassign functions are discussed in 5.6)**

Figure 5:19
Analogue Input assignment

5.5.7.2 Digital Input (type DI)

A digital input is a signal from a field input; it reflects the status of field equipment. A digital input is a read only Boolean and so can be on or off. It is treated as a Boolean except it cannot be assigned. (see 5.5.3)

5.5.7.3 Digital Output (type DO)

The digital output changes the on / off state of field equipment. It is treated as a read / write Boolean (see 5.5.3)

5.5.7.4 Digital Flag (type DF)

The digital flag is a read/ write Boolean that can be used any where that a digital output would be used. All the references to a digital flag are logically connected, and the flag state is set in one place in the compiled program. The variables are manipulated as flags in both PLC and WSL code. (see 5.5.4)

5.5.7.5 Word Input (type WI)

The word input is a read only integer, that can be treated as an integer (see 5.5.1); it does not contain any scaling or special processing.

5.5.7.6 Word Output (type WO)

This is an integer signal from the controller to the process control device. It is treated as a read / write integer (see 5.5.1).

5.5.8 Devices

There are two types of devices that were used in the fire and gas system, both of which were assigned to.

5.5.8.1 Single drive/ single feedback valve (type VSS)

The VSS device is either open or closed, and is controlled by a single discrete signal with one discrete feed back signal. The two types of VSS devices are energise-open and energise-close.

There are various values and keywords associated with the VSS:-

• .CMMD	open / close command	Read only Boolean
• .OPND	opened	Read only Boolean
• .CLSD	closed	Read only Boolean
• .TRVL	traveling	Read only Boolean
• .OLS	open feedback	Read only Boolean
• .CLS	closed feedback	Read only Boolean
• .FTO	fail to open	Read only Boolean
• .FTC	fail to close	Read only Boolean
• .DSBLD	forced to manual mode	Read / Write Boolean
• .LOCKD	locked (auto mode)	Read / Write Boolean
• .NRDY	not ready	Read / Write Boolean
• .MOPEN	manual open	Read / Write Boolean
• .OVRD	override feedback	Read / Write Boolean
• .STATUS	device status	Read / Write Boolean
• .VFLAGS	device status	Read only integer
• .OTCP	open timer/counter preset	Read / Write integer
• .OTCC	open timer/ counter current	Read / Write integer
• .CTCP	close timer/ counter preset	Read / Write integer

- .CTCC close timer/ counter current Read / Write integer

The commands that are used with this valve are:-

- LOCK place in auto mode
- UNLOCK place in manual mode
- OPEN open valve
- CLOSE close valve
- RESET clear feedback override and/or issues open/close command after .FTO or .FTC is true.

If the energize-open valve is open (.MOPEN = true), the control signal (.CMM) is set to true. If the desired state is closed (.MOPEN = false) the .CMMD bit is set to false. The feed back signal (.OLS) should be true when the valve is open and false when the valve is closed.

If the desired state of the energize-close valve is open (.MOPEN = true), the control signal (.CMMD) is set to false. If the desired state is closed (.MOPEN = false) the .CMMD bit is set to true. The feedback signal for the energize-close valve (.CLS) should be set to false when the valve is open and true when the valve is closed.

If the CLEAR CMMD on FTO or FTC option is selected, the .CMMD bit changes to false when the .FTR becomes true. The .CMMD bit remains false until the RESET command is issued. The RESET command issues an OPEN/ CLOSE command that turns on the .TRVL bit. The OPEN/ CLOSE alarm timer starts counting down when the RESET bit goes false. The clear command was not used and the .FTR ending was not listed in the endings that can be used with the valve in the manual.

Math language VSS expression	WSL assignment
<vss_assign> ::= <vss> . CMMD	<vss_assign>::=<vss> "[" CMMD "]"
| <vss>.OPND	| <vss>"["OPND "]"
| <vss>.CLSD	| <vss>"["CLSD "]"
| <vss>.TRVL	| <vss>"["TRVL "]"
| <vss>.OLS	| <vss>"["OLS "]"
| <vss>.CLS	| <vss>"["CLS "]"
| <vss>.FTO	| <vss>"["FTO "]"
| <vss>.FTC	| <vss>"["FTC "]"
| <vss>.DSBLD	| <vss>"["DSBLD"]"
| <vss>.LOCKD	| <vss>"["LOCKD "]"
| <vss>.NRDY	| <vss>"["NRDY"]"
| <vss>.MOPEN	| <vss>"["MOPEN "]"
| <vss>.OVRD	| <vss>"["OVRD "]"
| <vss>.STATUS	| <vss>"["STATUS "]"
| <vss>.VFLAGS	| <vss>"["VFLAGS"]"
| <vss>.OTCP	| <vss>"["OTCP "]"
| <vss>.OTCC	| <vss>"["OTCC "]"
| <vss>.CTCP	| <vss>"["CTCP "]"
| <vss>.CTCC	| <vss>"["CTCC "]"
| LOCK (<vss>)	| !p LOCK (<vss> var)
| UNLOCK (<vss>)	| !p UNLOCK (<vss> var)
| OPEN (<vss>)	| !p OPEN (<vss> var)
| CLOSE (<vss>)	| !p CLOSE (<vss> var)
| RESET (<vss>)	| !p RESET (<vss> var)
e.g.	e.g
UNLOCK (XS_20001_D)	!p UNLOCK (XS_20001_D var)
XS_20001_D.MOPEN := XS_2	XS_20001_D[MOPEN] := XS_2

Figure 5:20
VSS assignment

5.5.8.2 Dual drive/ dual feedback valve (type VDD)

The VDD device is either open or closed and is controlled by two discrete signals with two discrete feedback signals. The two control signals consist of an open signal (.OPENC) and a close signal (.CLSC), which are both normally false.

There are various values and keywords associated with the VSS:-

• .OPENC	open command	Read only Boolean
• .CLSC	close command	Read only Boolean
• .OPND	opened	Read only Boolean
• .CLSD	closed	Read only Boolean
• .TRVL	traveling	Read only Boolean
• .OLS	open feedback	Read only Boolean
• .CLS	closed feedback	Read only Boolean
• .FTO	fail to open	Read only Boolean
• .FTC	fail to close	Read only Boolean
• .FAILD	failed (both feedback bits are true)	Read only Boolean
• .CLSTO	close timeout	Read only Boolean
• .OPNTO	open timeout	Read only Boolean
• .DSBLD	forced to manual mode	Read / Write Boolean
• .LOCKD	locked (auto mode)	Read / Write Boolean
• .NRDY	not ready	Read / Write Boolean
• .MOPEN	manual open	Read / Write Boolean
• .OVRDO	override open feedback	Read / Write Boolean
• .OVRDC	override closed feedback	Read / Write Boolean
• .STATUS	device status	Read / Write Boolean
• .VFLAGS	device status	Read only integer
• .OTCP	open timer/counter preset	Read / Write integer
• .OTCC	open timer/ counter current	Read / Write integer
• .CTCP	close timer/ counter preset	Read / Write integer
• .CTCC	close timer/ counter current	Read / Write integer

The commands that are used with this valve are :-

- LOCK place in auto mode
- UNLOCK place in manual mode
- OPEN open valve
- CLOSE close valve
- RESET clear feedback override and/or issues open/close command after .FTO or .FTC is true.

If the desired state is open (.MOPEN = true), the .OPENC bit is set to true. The .OPENC bit remains true until either the open feed back signal is true or the open alarm time expires; then .OPENC is set to false.

If the desired state is closed, the .CLSC bit is set to close the valve. The .CLSC bit remains true until either the close feedback signal is true or the close alarm expires; then .CLSC is set to false.

The only command that is used is the reset command. That is used in the INIT part of the CFBs that deal with the valves so they are all unlocked at the start of the code.
The RESET command issues an OPEN/CLOSE command that turns on the .TRVL bit. The OPEN/CLOSE alarm timer starts counting down when the RESET bit goes false.

The two feedback signals consist of an open feedback signal (.OLS) and a closed feedback signal (.CLS). The .OLS bit should be true when the valve is open; otherwise, it should be false. The .CLS bit should be true when the valve is closed; otherwise it should be false.

Math language VDD expression	**WSL assignment**
<vdd_assign> ::=	<vdd_assign> ::=
<vdd> .OPENC	<vdd> "["OPENC "]"
|<vdd>.CLSC	|<vdd>"["CLSC "]"
| <vdd>.OPND	| <vdd>"["OPND "]"
| <vdd>.CLSD	| <vdd>"["CLSD "]"
| <vdd>.TRVL	| <vdd>"["TRVL "]"
| <vdd>.OLS	| <vdd>"["OLS "]"
| <vdd>.CLS	| <vdd>"["CLS "]"
| <vdd>.FTO	| <vdd>"["FTO "]"
| <vdd>.FTC	| <vdd>"["FTC "]"
| <vdd>.FAILD	| <vdd>"["FAILD "]"
| <vdd>.CLSTO	| <vdd>"["CLSTO"]"
| <vdd>.OPNTO	| <vdd>"["OPNTO "]"
| <vdd>.DSBLD	| <vdd>"["DSBLD"]"
| <vdd>.LOCKD	| <vdd>"["LOCKD "]"
| <vdd>.NRDY	| <vdd>"["NRDY"]"
| <vdd>.MOPEN	| <vdd>"["MOPEN "]"
| <vdd>.OVRD	| <vdd>"["OVRD "]"
| <vdd>.STATUS	| <vdd>"["STATUS "]"
| <vdd>.VFLAGS	| <vdd>"["VFLAGS"]"
| <vdd>.OTCP	| <vdd>"["OTCP "]"
| <vdd>.OTCC	| <vdd>"["OTCC "]"
| <vdd>.CTCP	| <vdd>"["CTCP "]"
| <vdd>.CTCC	| <vdd>"["CTCC "]"
| LOCK (<vdd>)	| !p LOCK (<vdd> var)
| UNLOCK (<vdd>)	| !p UNLOCK (<vdd> var)
| OPEN (<vdd>)	| !p OPEN (<vdd> var)
| CLOSE (<vdd>)	| !p CLOSE (<vdd> var)
| RESET (<vdd>)	| !p RESET (<vdd> var)

e.g.
 	
UNLOCK (XV_16061_D)
XV_16061_D.MOPEN := TRUE
XV_16061_DD := XV_16061_D.FTO
	e.g
	
	!p UNLOCK (XV_16061_D var)
	XV_16061_D[MOPEN] := TRUE
	XV_16061_DD := XV_16061_D[FTO]

Figure 5:21
VDD assignment

When declaring valves the time that is takes for them to change state is also declared. This should prevent valves that are opening from being closed and vice versa.

5.5.9 Recipes

A recipe serves as a user defined storage place for a set of related values that have different data types. The recipe (similar to a struct in C) allows the variables to be accessed in the following way:-

```
recipe_name.recipe_element := variable
or
recipie_name := recipe_name
```

Figure 5:22
Accessing a recipe element

The extensions associated with the recipes are:-

• .RTU	Request to unlock	Read / write Boolean
• .INUSE	Data is in use	Read / write Boolean
• .DSTBL	Data is stable	Read / write Boolean
• .DRDY	Data is ready for use	Read / write Boolean
• .STATUS	Recipe status	Read / write Boolean

The commands associated with the recipes are:-

• UNLOCK / CLEAR	Makes the recipe available for data
• SELECT	Makes data from one recipe available to the other.

Recipes have been used in both the ESD and F&G system. To make the translation possible the user defined recipes are being used as though they are part of the language definition. Each recipe has its own collection of recipe elements and they are as follows:-

5.5.9.1 Elements associated to Trip_all recipe (ESD)

• L_Limit	Real
• H_Limit	Real
• LL_Limit	Real
• HH_Limit	Real
• Bad_xmt_lim	Integer

5.5.9.2 Elements associated to Trip_lll recipe (ESD)

• L_Limit	Real
• LL_Limit	Real
• Bad_xmt_lim	Integer

5.5.9.3 Elements associated to Trip_hh recipe (ESD)

• HH_Limit	Real
• Bad_xmt_lim	Integer

5.5.9.4 Elements associated to Trip_ll recipe (ESD)

• LL_Limit	Real
• Bad_xmt_lim	Integer

5.5.9.5 Elements associated to Trip_lim recipe (ESD)

• LL_Limit	Real
• HH_Limit	Real
• Bad_xmt_lim	Integer

5.5.9.6 Elements associated to CODELL recipe (F&G)

• hi_lim	Real
• xmt_low	Integer
• xmt_high	Integer
• optic_lo	Real
• optic_high	Real

5.5.9.7 Elements associated to SING_LIM recipe (F&G)

• hi_lim	Real
• xmt_low	Integer
• xmt_high	Integer

5.5.9.8 Elements associated to Dual_lim recipe (F&G)

• hi_lim	Real
• hihi_lim	Real
• xmt_low	Integer
• xmt_high	Integer

Each of the recipe elements will be turned into an item in an array. The following language definition will only show the PLC functions used in the ESD or F&G code and the Trip_all extensions because they are all translated using the same theory.

Math language recipes	**WSL**
<recipe> ::= < recipe_name>	<recipe> ::= < recipe_name>
| <recipe_name> .RTU	| <recipe_name> "["RTU "]"
| <recipe_name>.INUSE	| <recipe_name>"["INUSE"]"
| <recipe_name>.DSTBL	| <recipe_name>"["DSTBL"]"
| <recipe_name>.DRDY	| <recipe_name>"["DRDY"]"
| <recipe_name>.STATUS	| <recipe_name>"["STATUS"]"
| UNLOCK(<recipe_name>)	| !p UNLOCK(<recipe_name> <var>)
| CLEAR (<recipe_name>)	| !p CLEAR(<recipe_name> <var>)
| SELECT (<recipe_name>)	| !p SELECT(<recipe_name> <var>)
| <recipe_name>.L_LIMIT	| <recipe_name>"["L_LIMIT "]"
| <recipe_name>.H_LIMIT	| <recipe_name>"["H_LIMIT"]"
| <recipe_name>.LL_LIMIT	| <recipe_name>"["LL_LIMIT"]"
| <recipe_name>.HH_LIMIT	| <recipe_name>"["HH_LIMIT"]"
| <recipe_name>. BAD_XMT_LIM	| <recipe_name>"[" BAD_XMT_LIM"]"
e.g.	e.g
IF(PT_13180 >= PT_1380_R.HH_LIMIT)	IF ((PT_13180) >= (PT_1380_R [HH_LIMIT]))
IF (PT_13180 <= PT_13180_R.LL_LIMIT)	IF ((PT_13180) <= (PT_13180_R[LL_LIMIT])))

Figure 5:23
Recipe usage

5.5.10 Hardware

Hardware addresses can be read and written to directly; these are treated in the same way as variable names are. Hardware addresses in WSL and the APT languages are a variable name except they start with a percentage sign.

5.6 PROCEDURE CALLS

Procedure calls are predefined operations that are available from within the math language and count as a statement. These operations will be represented as external procedure calls.

Only those procedures used in the ESD and F&G code are discussed.

Note

most significant bit — least significant bit

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1

second row demonstrates the number 153

Figure 5:24
Binary storage of integers

5.6.1 UNPACK_BITS (Boolean array , variable)

This procedure is only used in RLL and shifts a specified number of bits from an integer into a Boolean array. The number of bits moved from the integer to the array is determined by the size of the declared array.

Math Language Procedure call	WSL
UNPACK_BITS (<Boolean_array> , <integer>)	!p UNPACK_BITS (<integer> var <Boolean_array>)

Figure 5:25
UNPACK_BITS procedure call

5.6.2 PACK_BITS (Boolean array, variable)

This procedure is used only in RLL and shifts the values in the Boolean array into an integer variable. The size of the array will determine the number of bits that can be moved into the integer variable.

Math Language Procedure call	WSL
PACK_BITS (<Boolean_array> , <integer>)	!p PACK_BITS (<Boolean_array> var <integer>)

Figure 5:26
PACK_BITS procedure call

5.6.3 BCDBIN (variable, variable)

The BCDBIN procedure converts a binary coded decimal (BCD) value into an integer. The first variable in the parenthesis contains the four-digit BCD value to be converted, the second variable is the integer where the result is to be stored.

Math Language Procedure call	**WSL**
BCDBIN (<bcd_value> , <integer>)	!p BCDBIN (<bcd_value> var <integer>)

Figure 5:27
BCDBIN procedure call

5.6.4 BIT_ASSIGN (variable, integer, expression)

The BIT_ASSIGN procedure sets the individual position of an integer based on the result of a Boolean expression

The first value in the parenthesis is an integer that contains the bit that is to be set depending on the Boolean expression. The integer is treated as a binary value with the most significant bit as the number 1 bit and the least significant as the 16th bit. The second value in the parenthesis specifies which bit is to be changed (so is a number between 1 and 16). The Boolean expression evaluates to true or false and depending on the answer will be how the significant bit of the variable will be set.

Math Language Procedure call	**WSL**
BIT_ASSIGN (<variable> , <integer> , <bool_expression>)	!p BIT_ASSIGN (<integer> , <bool_expression> var <variable>)

Figure 5:28
BIT_ASSIGN procedure call

5.6.5 BITCLEAR (variable, integer)

The BITCLEAR procedure resets a specified bit in an integer variable to false, off (0).

The first variable in the parenthesis is the integer that contains the bit to be reset. The integer value represents which bit is to be reset; the value must be a number between 1 and 16.

Math Language Procedure call	WSL
BITCLEAR (<variable> , <integer>)	!p BITCLEAR (<integer> var <variable>)

Figure 5:29
BITCLEAR procedure call

5.6.6 BITSET (variable, integer)

The BITSET procedure resets a specified bit in an integer variable to true, on (1).

The first variable in the parenthesis is the integer that contains the bit to be set. The integer value represents which bit is to be reset; the value must be a number between 1 and 16.

Math Language Procedure call	WSL
BITSET (<variable> , <integer>)	!p BITSET (<integer> var <variable>)

Figure 5:30
BITSET procedure call

5.6.7 LOAD_ARRAY(input variable, array)

The LOAD_ARRAY procedure assigns a value to the elements of an array.

The input variable and the array elements must be of the same type. The value of the input variable is assigned to each of the elements in the array.

Math Language Procedure call	**WSL**
LOAD_ARRAY (<variable> , <array>)	!p LOAD_ARRAY (<variable> var <array>)

Figure 5:31
LOAD_ARRAY procedure call

5.7 FUNCTION CALLS

A function appears on the right hand side of an assignment statement or in a Boolean expression i.e. it returns a value. All functions are translated to an external function call.

5.7.1 BITS_TO_INTS (array variable)

The BITS_TO_INTS function moves an array of 16 Boolean variables into an integer value. Within the Boolean array true is 1 and false is 0. Element 1 of the array is the most significant bit.

Math Language function call	**WSL**
BITS_TO_INTS (<bool_array>)	!f BITS_TO_INT (<bool_array>)

Figure 5:32
BITS_TO_INT function

5.7.2 BITTEST (variable , integer)

The BITTEST function checks the status of a specified bit. The variable is the integer which contains the bit that is to be checked (the most significant bit is numbered as number 1 the least significant bit is the 16th bit). The integer value specifies which bit in the binary equivalent of the number is to be tested. If the tested bit is 1 then true is returned otherwise 0 is returned.

Math Language function call	**WSL**
BITTEST (<variable> , <integer>)	!f BITTEST (<variable>, <integer>)

Figure 5:33
BITTEST function

5.7.3 EDGE (expression)

Edge is a Boolean function that detects the change from false to true in the value of a Boolean expression. The variables in the parentheses are monitored to detect the first time that the expression changes from false to true. When this change has been detected the returned value of the function is true and remains true for one scan of the controller.

Edge stores temporary variables each time it is used. It performs an edge on the variables since last time an 'edge' was performed on it and not since the last time the current 'edge' was performed on the variable.

Math Language function call	**WSL**
EDGE (<expression>)	!f EDGE (<expression>)

Figure 5:34
EDGE function

5.7.4 ABS(expression)

ABS is a function that returns the positive values of an expression. Note that it is also a function supplied in WSL.

Math Language function call	WSL
ABS (<expression>)	abs (<expression>)

Figure 5:35
ABS function

5.8 TIME

The controller contains the date and time which can be accessed by the program. The only part of the time that is accessed by the translated code is the hour past midnight which is stored as a read only integer. This means that the controller clock can not be set from the program.

Math Language time expression	WSL
x := program_name . IHOUR	x : = !f TIME (var)

Figure 5:36
Time expression

5.9 ASSIGNMENTS

Numerical assignments map as would be expected in the following way:-

Math language assignment	**WSL assignment**
<assignment assignment> ::= <assign>	
<assign> ::= <variable> := <value> ;	<assign> ::= <variable> := <value>
<value> ::= <Boolean> <compound booeanl> I <real> I <compound real> I <integer> I <compound interger>	
e.g. X45 := 1; X44 := True;	e.g. X45 := 1; X44 := TRUE

Figure 5:37
Assignment format in Math language and WSL

When an assignment is Boolean value of true or false it has to be assigned as an integer value of 0 or 1. Rather than changing the Boolean assignment to 1 for true and 0 for false two global WSL constants will be declared.

```
FALSE := 0;
TRUE := 1;
```

Figure 5:38
WSL constant declaration

The assignment of Booleans in WSL can now remain identical to the assignment in PLC. True and False were not case sensitive in the APT languages but they are case sensitive in WSL. Variables with special methods of assignment are discussed when the variables are defined, below are two such examples.

Math language flag / timer assignment	**WSL flag / timer assignment**

clear (firstup_set)	firstup_set[LATCH] := FALSE
	firstup_set[ON] := FALSE

timer_name.ENABL := TRUE	timer_name[ENABL] := TRUE;
	timer_set(var timer_name)

Figure 5:39
Assignment format in Math language and WSL

Conditionals on the right hand side of an assignment statement are not allowed in WSL so they are translated to an if statement on the right hand side that will return true or false.

Math language conditional assignment	**WSL conditional assignment**
a:= (b = c)	a:= if (b =c) then TRUE else FALSE fi

Figure 5:40
If statement in an assignment

The equality operators that cause this sort of assignment are '=', '<', '<=', '>', '>=', '<>' i.e. anything that would create a Boolean result.

5.10 CONDITIONALS

The conditional in math language is an if statement of the following format:-

Math language conditional	**WSL conditional**
IF (<Boolean expression>) THEN	if (<Boolean expression>) then
<statements>	<statements>
{ ELSIF (<Boolean expression>) THEN	{ elsf (<Boolean expression>) then
<statements>}	<statements>}
[ELSE <statements>]	[else <statements>]
ENDIF;	fi
e.g.	e.g.
IF (X45) THEN	if (X45 =1) then
X45 := True;	X45 := True
ELSIF (X44) THEN	elsf (X44 = 1) then
X21 := X34;	X21 := X34
ENDIF;	fi

Figure 5:41
Conditional format in Math language and WSL

Note the missing ';' at the end of the WSL conditional, this is because in WSL a ';' joins two statements whereas in most languages it indicates the end of a statement.

The Boolean expression will have to be translated from the PLC format:-

```
X454 OR X3524 OR X234
```

to the WSL format:-

```
(X454 = TRUE) or (X3524 = TRUE ) or (X234 = TRUE)
```

There can be any number of ELSIF statements in both languages. There can also be nesting of if statements and this frequently occurred.

The xor logical operator is not declared in WSL so it is translated to a function call that returns true or false.

Xor function call

```
xor (a,b)
```

Xor function defined

```
funct xor(x,y) ==
        if ( x = y) then 1
        else 0
        fi
.
```

Figure 5:42
xor function defined

5.11 WHILE LOOP

While loops are allowed within the math language and are of the following format:-

Math language conditional	**WSL conditional**
WHILE (<Boolean expression>) LOOP	while (<Boolean expression>) do
<statements>	<statements>
END LOOP;	od
e.g.	e.g.
WHILE (X45) LOOP	while (X45 =1) do
X45 := True;	X45 := True;
X21 := X34;	X21 := X34
END LOOP;	od

Figure 5:43
While loop format in Math language and WSL

6. TABLE INFORMATION FOR DECLARING VARIABLES

Variables in the APT language can contain numbers, letters and the '_'; they can be up to 12 characters long and must contain a letter. Since the APT variables can start with a number and WSL variables cannot all of the variables will have WSL_ prefixed to the beginning of their name.

Most of the declarations within the APT are in the tables. The scope of the variables declared in the table are dependent on the position of the table. A table in a unit means the variables are global to only the SFCs and CFCs in that unit. Variables declared in the higher level i.e. the program level are global to CFCs and SFCs in all units.

All of the variable information is stored in two binary files. There is one file called App.d2 in the database directory which contains all the variable information about all the programs within the APT at a given time. It also includes information about previous programs because it does not seem to delete the information properly. The format of this variable information is as follows:-

```
hex   variable name type of declaration e.g. "INT_DEC"  key letter initial e.g. "I"
      comments      initial value    PLC address.


e.g.
*M    TIGER                  INT_DECL   I      This is a tiger  5       Automatic
```

If the value is a constant then the hex number 4 before the letter 'A' of Automatic is 01.

Type of Declaration	Key letter
Boolean	B
Integer	I
Real	R
Flag	F
Text	T
Integer Array	IA
Real Array	RA
Boolean Array	BA
DO10 Array	DX
Text Array	TA
Slow Timer	ST
Analogue Input	AI
Digital Flag	DF
Digital Input	DI
Digital Output	DO
Word Input	WI
Word Output	WO
Valve single drive / single feed back	VSS
Valve Dual drive /dual feed back	VDD
Templates - user defined	

Figure 6:1
Types of Variable declarations

This information is stored in an arbitrary order and the number of spaces between the information seems to be arbitrary. The solution will be to search for the variable name so that the information about the variable can be obtained.

A list of the program name, units, CFB and variables is stored in the file called object.xrf. There is an object.xrf file for each of the programs that are declared. This can be found at path:-

\APT\PROGRAM\ESD_A\OBJECT.XRF

This contains a list of the unit name and then the name of the item that it is storing. The difference between the math blocks and variables cannot be determined but the math blocks can be identified from the graphic file (see 2.1). The units are declared as unit name then item name (also obtainable from directory names). So the variables are the names that follow a unit name and are not units or CFB. This information can then be used to identify the type of variable from app.d2. The unit name has to be maintained as this will indicate the globality of the variables and they have to be declared within their units. The global variables have the hex value 0007010000 or 1000000028 before the variable name. The information about recipes is stored in app.d3.

6.1 MATH BLOCK TYPE

The type of an interlock can be determined using a similar method, a high priority interlock should be run before a SFC and a low priority interlock should be after the SFCs. App.d1 in the database directory stores the information about the type of math block. This includes the name of the CFC that it belongs to, the comment of the math block (this is the only place that this information is stored) and whether it is high or low priority. The information stored with the math block gives the type of the math block which is active since all of the used math blocks are active.

Format of storing this information:-

name of CFB Keyword type key number type comment name of CFC hex keyword e.g.

W_DOG_0	STNDMATH	27	comment	CFCname	*M ACTIVE	OUS
PHASE_1	INTRLOCK	51	comment	SELFTEST	hex	HIGH
PHASE_2	INTRLOCK	51	comment	SELFTEST	hex	HIGH

All of the files listed above are fixed length files.

6.2 COMMENTS WITH UNITS AND PROGRAMS

App.d1 stores the program name and comment the programmer supplied with the program in following format:-

hex	Program_name	PRG	comment	hex

The first writing after the PRG is comment and then the comment continues until there is a specific hex character. App.d2 stores the comments with the units, the comments are the letters after the unit name until there is more hex which is not printable text.

7. SFC $\rightarrow$ WSL

A SFCs name can have up to 8 alpha numeric characters; the name includes at least one letter and may contain an underscore. The name cannot begin with a number.

The SFC diagram will be converted as a whole into WSL. Each of the steps will be an action system. An action system is a facility in WSL that allows pieces of code containing GOTOs to be translated Although the SFC does not explicitly have GOTOs depending on the transition options the control of the program could move to one or more different steps. The best way to map this will be via action blocks as there are very powerful transformations to remove the GOTO jumps in action systems. The transitions out of the action system will be an if statement (which will test on the transition condition) and jump to the next step/ action system. If none of the conditions of the transitions (if else if statement) are true then control will go to the start of the action block. There will be two action blocks for each step, the INIT part which is executed at least once and then the main part which is executed zero or more times. The action system will retain the same number as the step; these are numbered S1 to S500 (not in execution order but in coded order). The number is Ix for the initial action system and Ax for the main body of the action system. The code that is written within the safe SFC in the body of the step is put into a block as a procedure call; the name of the procedure is Sx and ISx for the initial part of the step. The name of the transition was retained in the first parse of the document to get the layout. When the math language was translated and inserted the name of the transition was no longer required but for reference purposes it was stored as a comment.

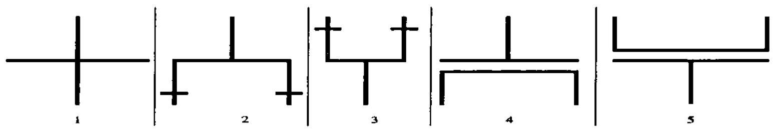
Where there is a choice of transitions the ordering is maintained from the left to right. Once the transition has been made there will be an action call to the step that is in that transition. The code does not contain parallel steps although the code does contain statements that are executed in parallel. It is understood that code executed in parallel is executed sequentially in an indeterminate order. This is located before the MATH keyword in each of the steps.

SFCs can be of two types normal and safe state SFCs. A safe state SFC once it has been turned on interrupt's its local SFC and takes over control as soon as a trigger condition becomes true. This is mapped by an extra condition in the transaction to enable a break out into the safe state SFC which is built into the same action system.

BNF form of a SFC

<SFC> ::= <name> <start step> <transition> {transition} {step} <end step>

<transition> :=



Non BNF rules

1. Steps must be separated by transitions
2. ⇔ 8 Alpha characters
3. there are constraints on loops
4. there must be a step before a transition.

<sfc1> control enters at start step and exits at 1 or more end steps possibly at a later time.
<transion1> if transition is true goto next step if false re-execute the previous step.
<transition2> if left hand side branch true more to that step if right hand side step is true move to the step on right hand side otherwise repeat the previous step.
<transition3> if transition on left hand side and right hand side are both true then move to the next transition
<transition4> not implemented
<transition5> not implemented

WSL

A step will be converted into a WSL action block
A transition will be converted into an if statement at the end of the action block and a goto statement to move to the required action block.

Figure 7:1
syntax of an SFC

There are two extensions associated with the main SFC in each of the units. These are the .enabl and the .abort extension and they allow all of the SFCs within a unit to be turned off. When the program is originally downloaded the .enabl extension is set to true and the .abort is set to false.

During the running of an SFC it can be halted and will not run again until reset. In other parts of the code the SFC can be reset or stopped so that it will not run again until it is turned on. If unitname.ENABL is set to false then all the currently active steps in the SFC become inactive. If the unitname.ABORT is set to true then the SFC will become inactive. The execution does not resume until the enabl command is set to true, if the .abort command is set to false only, the SFC does not resume execution.

SFC unitname extensions	**WSL unitname extensions**
<extension> := <unitname> '.' ENABL | <unitanme> '.'ABORT	<extension> := <unitname> '[' ENABL ']' | <unitanme> '[' ABORT ']'
e.g	e.g
selftest.abort := TRUE	WSL_SELFTEST[ENABL] := TRUE

Figure 7:2
syntax of unitname extensions

If a controller looses power then all the units become inactive and remain inactive until the power returns. When the power returns each unit / program starts at the initial step of the main SFC; except for specifically set safe state SFCs.

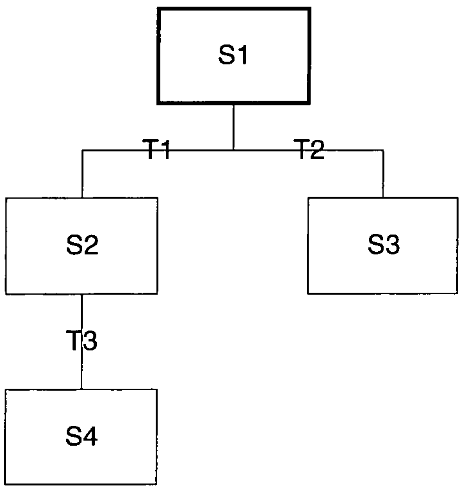
An SFC can be turned off from elsewhere in the code, by giving the variable within the unit the value of false. As such at the top of each SFC there will have to be a test to see if the code should be run or not and this will be in the form of:-

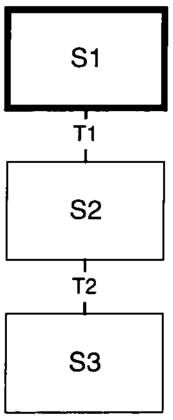
```
if ((unitname[ENABL] := TRUE ) and (unitname[ABORT] := FALSE))  then
        sfc code
fi
```

Figure 7:3
Initial if statement before a piece of code

It should be noted that an entire action system is thought of as one statement.

SFC	**WSL**

Normal SFC 	begin
	if ((normal[ENABL] = TRUE) and
	    (normal[ABORT] = FALSE))
	    then
	    WSL_safe[ARM] := FALSE;
	    actions: MI1:
	MI1==IMS1(var);
	if (normal[ENABL] = FALSE) then
	        call z
	elsf ((sstrigger = TRUE) and
	    (WSL_safe[ARM] = TRUE))
	    then call safe
	    elsf ((MT1 = TRUE))
	        then call MI2
	    elsf ((MT2 = TRUE))
	        then call MI3
	        else call MA1
	    fi .
	
	MA1==MMS1(var);
	if (normal[ENABL] = FALSE) then
	        call z

Safe SFC

	```
	        elsf ((sstrigger = TRUE) and
	(WSL_safe[ARM] = TRUE))then
	                call safe
	        elsf ((MT1 = TRUE))
	                then call MI2
	        elsf ((MT2 = TRUE)) then call MI3
	                else call MA1
	        fi .

	MI2==IMS2(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((sstrigger = TRUE) and
	        (WSL_safe[ARM] = TRUE))then
	                call safe
	        elsf ((MT3 = TRUE))
	                then call MI4
	                else call MA2
	        fi .

	MA2==MMS2(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((sstrigger = TRUE) and
	        (WSL_safe[ARM] = TRUE))then
	                call safe
	        elsf ((MT3 = TRUE))
	                then call MI4
	                else call MA2
	        fi .

	MI3==IMS3(var);
	```

```
if (normal[ENABL] = FALSE) then
            call z
      elsf ((sstrigger = TRUE) and
      (WSL_safe[ARM] = TRUE))then
            call safe
            else call MA3
      fi .

MA3==MMS3(var);
if (normal[ENABL] = FALSE) then
            call z
      elsf ((sstrigger = TRUE) and
      (WSL_safe[ARM] = TRUE))then
            call safe
            else call MA3
      fi .

MI4==IMS4(var);
if (normal[ENABL] = FALSE) then
            call z
      elsf ((sstrigger = TRUE) and
      (WSL_safe[ARM] = TRUE))then
            call safe
            else call MA4
      fi .

MA4==MMS4(var);
if (normal[ENABL] = FALSE) then
            call z
      elsf ((sstrigger = TRUE) and
      (WSL_safe[ARM] = TRUE))then
            call safe
```

	```
	                else call MA4
	        fi .

	safe==ISS1(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((ST1 = TRUE))
	                then call SI2
	                else call Asafe
	        fi .

	Asafe==MSS1(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((ST1 = TRUE))
	                then call SI2
	                else call Asafe
	        fi .

	SI2==ISS2(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((ST2 = TRUE))
	                then call SI3
	                else call SA2
	        fi .

	SA2==MSS2(var);
	if (normal[ENABL] = FALSE) then
	                call z
	        elsf ((ST2 = TRUE))
	                then call SI3
	```

	else call SA2
	fi .
	SI3==ISS3(var);
	if (normal[ENABL] = FALSE) then
	call z
	else call SA3
	fi .
	SA3==MSS3(var);
	if (normal[ENABL] = FALSE) then
	call z
	else call MI3
	fi .
	end_actions
	fi
	where
	proc IMS1(var) == ....
	proc MMS1(var) == ....
	proc IMS2(var) == ....
	proc MMS2(var) == ....
	proc IMS3(var) == ....
	proc MMS3(var) == ....
	proc IMS4(var) == ....
	proc MMS4(var) == ....
	proc ISS1(var) == ....
	proc MSS1(var) == ....
	proc ISS2(var) == ....
	proc MSS2(var) == ....

	proc ISS3(var) == ....
	proc MSS3(var) == ....
	.
	end

Figure 7:4
An SFC to converted WSL

The SFC will be translated in three stages. The first stage will be to split up the document into individual files and procedures; these will then individually be translated using the parser. The layout of the code will then be translated and the WSL procedures inserted at the correct point.

7.1 MATH LANGUAGE ASSOCIATED WITH SAFE STATE SFCS

A safe state SFC is a special SFC that is designed to interrupt the execution of an SFC, this is so that an emergency procedure can be performed. Or so that special processing can be performed that is out of normal flow of control. There are three types of safe state SFCs the ESD and F&G code uses the second type following.

1. **A level safe state SFC** which is designed to interrupt the processing of a main or subordinate SFC. A level safe state SFC can only interrupt another safe state SFC that has a lower priority.
2. **A local safe state SFC** which is designed to interrupt the processing of a single SFC or and of its subordinates.
3. **A subordinate safe state SFC** which is called by another safe state SFC.

Although an SFC can have more than one local safe state SFC only one of them can be active at any point in time.

There are 7 math commands that are related directly to the use of safe state SFCs. All but SSABORT are used in the sample code.

- **SSENTRY** this is the point in the normal SFC of re-entry after the safe state SFC has finished executing.
- **SSRETURN** is the return point from the safe state SFC.
- **SSDEFINE** sets which SFC the safe state SFC is local to.
- **SSTRIGGER** the Boolean which when true calls the safe state SFC.
- **SSARM** indicates when to start looking for SSTRIGGER = TRUE to call the safe state SFC.
- **SSDISARM** indicates when to stop looking for SSTRIGGER = TRUE to call the safe state SFC.
- **SSABORT** stops all the SFCs

7.1.1 SSENTRY (label)

SSENTRY is written in a step of the main SFC (and there must be only one connected to each label) indicating where the control is to return to once the safe SFC has finished. SSRETURN is the command that returns the control to the main SFC at the corresponding label. The command will be converted into a comment in WSL.

7.1.2 SSRETURN (label)

This command transfers the program execution from the safe SFC to the step identified by the corresponding SSENTRY command (i.e. with the same label). When the SSRETURN command is executed the previously active step in the main SFC and the safe SFC are deactivated and the step containing the SSENTRY is activated. The command will be converted into a comment in WSL.

7.1.3 SSDEFINE

Used in the initial step of the safe state SFC, and every initial step must contain one and only one. It defines the type of safe state SFC and with the case of a local safe state SFC it defines which SFC it is local to. The command will be converted into a comment in WSL.

SSDEFINE LOCAL TO sfc_name

This command will also be converted to a comment, as the calling action system will know which safe state action to call.

7.1.4 SSTRIGGER (identifier)

This command specifies the condition that triggers the execution of a main safe state SFC. If the trigger condition becomes true while the safe state SFC is disarmed then the condition is ignored. If the trigger condition becomes true when the safe state SFC is armed then the control of the program passes to the initial step safe state SFC. The identifier is a Boolean, a flag or a digital input.

7.1.5 SSARM (Safe State SFC name)

This command 'arms' a safe state SFC and thus makes it possible for the safe state SFC to interrupt the execution flow if the trigger value becomes true. Once armed the safe state SFC will continually monitor the trigger for when it should interrupt the flow of control.

A safe state SFC starts executing when:-

1. Safe state SFC is armed
2. SFC executing is a normal SFC
3. A trigger condition becomes true.

SSARM remains in effect until SSDISARM is called. The math language code is converted into WSL code see below.

7.1.6 SSDISARM (Safe State SFC name)

This command 'disarms' a safe state SFC and thus it no longer monitors the trigger condition, and the safe state SFC can no longer perform an interrupt. SSDISARM takes precedence over SSARM

7.1.7 SSABORT

This command suspends all SFC execution in the current unit.

7.1.8 Translation of the commands

PLC safe state SFC commands	WSL safe state SFC commands
<sfc_commands> :- SSENTRY <label>	<sfc_commands> :- comment: "SSENTRY <label>"
I SSRETURN <label>	I comment: "SSRETURN <label> "
I SSDEFINE LOCAL OF <sfc_name>	I comment: " SSDEFINE LOCAL OF <sfc_name>"
I SSTRIGGER <identifier>	I comment: " SSTRIGGER <identifier>"
I SSARM <safe_sfc_name>	I comment: " SSARM <safe_sfc_name>" ; safe_sfc_name '['ARM']' := FALSE
I SSDISARM <safe_sfc_name>	I comment: " SSDISARM <safe_sfc_name>"; safe_sfc_name '['ARM']' := TRUE

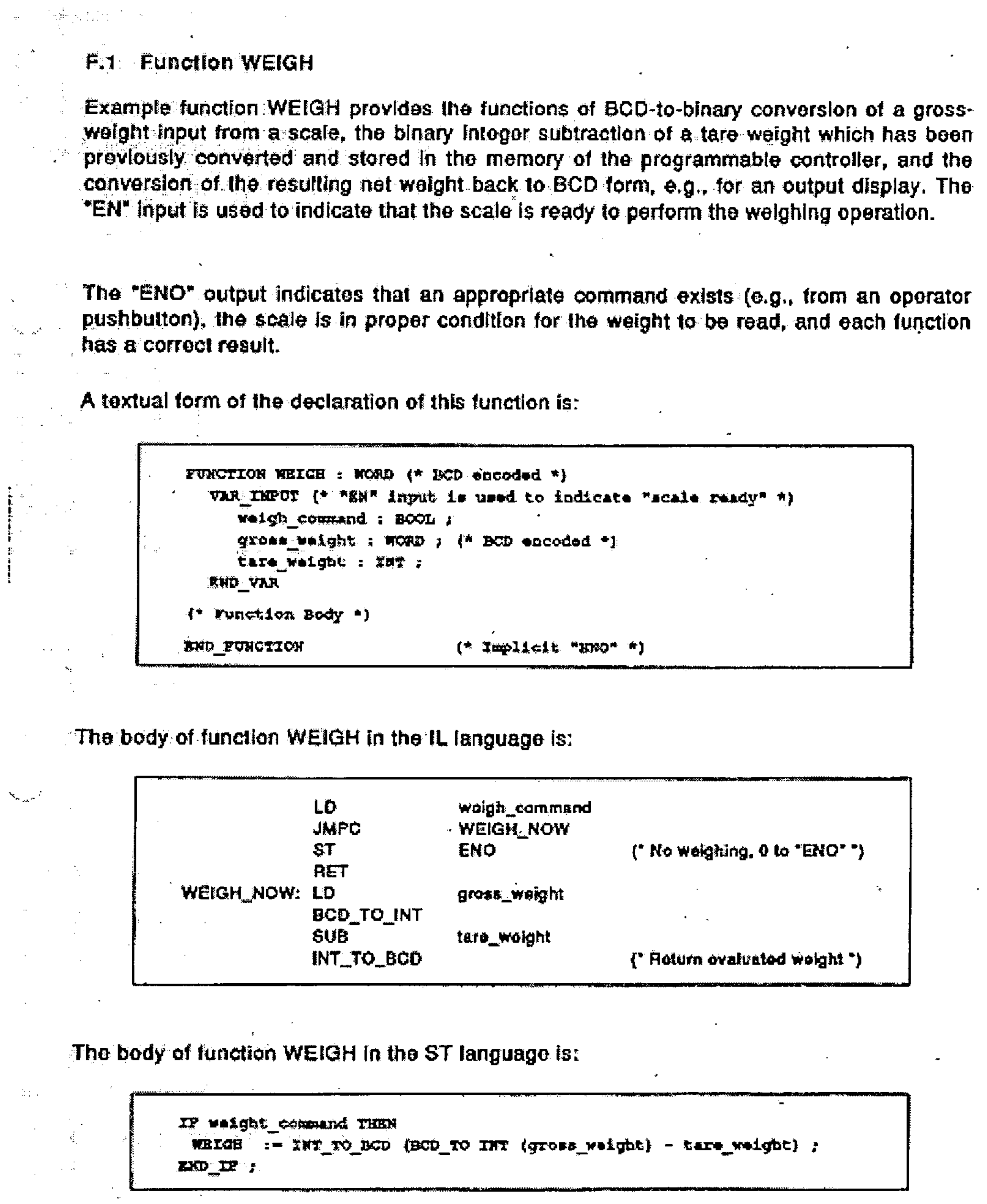
Figure 7:5
Safe state SFC commands

REFERENCES

[1] Siemens, "SIMATIC APT programming manual (4.2.1 and 4.3)," .

APPENDIX IV

WEIGH FUNCTION IN EACH OF THE FOUR IEC 1131-3 LANGUAGES



F.1 Function WEIGH

Example function WEIGH provides the functions of BCD-to-binary conversion of a gross-weight input from a scale, the binary integer subtraction of a tare weight which has been previously converted and stored in the memory of the programmable controller, and the conversion of the resulting net weight back to BCD form, e.g., for an output display. The "EN" input is used to indicate that the scale is ready to perform the weighing operation.

The "ENO" output indicates that an appropriate command exists (e.g., from an operator pushbutton), the scale is in proper condition for the weight to be read, and each function has a correct result.

A textual form of the declaration of this function is:

```
FUNCTION WEIGH : WORD (* BCD encoded *)
  VAR_INPUT (* "EN" input is used to indicate "scale ready" *)
    weigh_command : BOOL ;
    gross_weight : WORD ; (* BCD encoded *)
    tare_weight : INT ;
  END_VAR
(* Function Body *)
END_FUNCTION                (* Implicit "ENO" *)
```

The body of function WEIGH in the IL language is:

```
            LD          weigh_command
            JMPC        WEIGH_NOW
            ST          ENO           (* No weighing, 0 to "ENO" *)
            RET
WEIGH_NOW:  LD          gross_weight
            BCD_TO_INT
            SUB         tare_weight
            INT_TO_BCD                (* Return evaluated weight *)
```

The body of function WEIGH in the ST language is:

```
IF weight_command THEN
 WEIGH  := INT_TO_BCD (BCD_TO INT (gross_weight) - tare_weight) ;
END_IF ;
```

Figure 1

Page 157 of IEC 1131-3 the weigh function [1]

An equivalent graphical declaration of function WEIGH is:

```
            +------------------------+
            |         WEIGH          |
    BOOL---|EN                   ENO|---BOOL
    BOOL---|weigh_command  net_weight|---WORD
    WORD---|gross_weight             |
     INT---|tare_weight              |
            +------------------------+
```

The function body in the LD language is:

```
|              +---------+             +---------+          |
|              |  BCD_   | +--------+  |  INT_   |          |
| weigh_command|  TO_INT | |  SUB   |  | TO_BCD  |    ENO   |
+--------| |---|EN    ENO|--|EN   ENO|---|EN    ENO|----( )-----+
|              |         | |        |  |         |          |
| gross_weight--|         |--|        |---|         |--net_weight |
|              +---------+ |        |  +---------+          |
| tare_weight---------------|        |                      |
|                          +--------+                       |
```

The function body in the FBD language is:

```
                  +---------+             +---------+
                  |  BCD_   | +--------+  |  INT_   |
                  | TO_INT  | |  SUB   |  | TO_BCD  |
weigh_command---|EN    ENO|---|EN   ENO|---|EN    ENO|---ENO
gross_weight----|         |---|        |---|         |--net_weight
                  +---------+ |        |  +---------+
tare_weight-------------------|        |
                              +--------+
```

Figure 2
Page 158 of IEC 1131-3 the weigh function [1]

REFERENCES

[1] IEC, "IEC 1131_3 Programmable controllers - Part 3: Programming languages,"

.