2008

# Automated generation of SW design constructs from Mesa source code

Keith Needels

Rochester Institute of Technology - Masters Degree Program
School of Computer Science

# THESIS FINAL REPORT

# VOLUME I

SUBJECT: **AUTOMATED GENERATION OF SW DESIGN CONSTRUCTS FROM MESA SOURCE CODE**

AUTHOR: **DAVID EGERTON**

DATED: **4Q1993**

REVISION: **1.1**

Rochester Institute of Technology - Masters Degree Program
School of Computer Science

# Automated generation of SW design constructs from Mesa source code:

by

David Egerton

A thesis submitted to the School of Computer Science in partial fulfillment of the requirements of the degree of Master of Science in Computer Science.

Approved by:

Professor Alan Kaminsky

Professor James Heliotis

Professor Peter Anderson

Rochester Institute of Technology - Masters Degree Program
School of Computer Science

# TABLE OF CONTENTS

# VOLUME I Ver 1.1

## 1.0 INTRODUCTION AND BACKGROUND

# 2.0 PROJECT DESCRIPTION

# 3.0 CONCLUSIONS & FINDINGS



# 4.0 BIBLIOGRAPHY

Rochester Institute of Technology - Masters Degree Program
School of Computer Science

# ILLUSTRATION LIST

## ILLUSTRATIONS LIST CONTINUED

# TABLE LIST

Rochester Institute of Technology - Masters Degree Program
School of Computer Science

# TABLE OF CONTENTS CONTINUED

## VOLUME II Ver 1.0

### APPENDICES

# SECTION 1 - INTRODUCTION AND BACKGROUND

## 1.0 INTRODUCTION AND BACKGROUND

This section is split into five major categories. The introduction section which outlines the basic premises of the problems to be tackled, the Design Representation section which details IEEE standard representation guidelines for design, the Mesa Language section which introduces the Xerox proprietary programming language, the Reverse Engineering section bringing focus to the concept of extracting design information from code, and finally a section detailing the scope of the thesis.

## 1.1 Problem statement

The introduction section below, in conjunction with the scope section (1.1.5), details the essential elements of the items of emphasis for this thesis.

### 1.1.1 Introduction

This thesis is concerned with the Reverse Engineering of Mesa code. Mesa is a programming language used by the Xerox Corporation especially designed for ultra large scale software projects. Many Xerox projects have considerable amounts of code already developed and in service with very little up to date design documentation. The maintenance phase of Software requires that updates will occur continuously in response to customer demands. Software engineers, often not the original designers, will need to gain a thorough understanding of the code to make reliable changes. This thesis proposes to outline Software tools developed in Mesa that will extract design information. The extracted information should be suitable for the graphical and tabular representations of architectural constructs. The tools will be written and applied originally to a small portion of a current project. These will serve as prototype for tools that could be scaled up to parse several hundred thousand lines of code. Clearly, not all design information

is available from the code. The theory section of the thesis will develop an understanding of the design documentation expectations and compare these to the information that can be Reverse Engineered from code.

### 1.1.2 Design Representation

The IEEE has developed a design standard for software . This is one of the major reference sources of design expectations. The original charter for this endeavor was approved on September 22 1983 [# 26]. The standard was developed by committee meetings and working groups involving many members from top companies in Software development, AT&T, IBM and Digital to name but a few. The standard then represents a useful indication of the expected content of software design information in industry today. The final designation for the standard is IEEE Std 1016-1987 which was approved on March 12, 1987.

Section 5 of the IEEE standard identifies the minimum set of design documentation that should accompany a software product. This thesis deals with design information that can be retrieved from code. We will use the IEEE recommendations as a guide to establish the items that should be present within a software product (code and documentation) and compare this to the documentation available by automatic extraction. This approach may also yield recommendations for changes to source code development standards that will better align the automatically extracted set with the IEEE set.

The IEEE standard defines a software design description as follows:

> "A software design description is a representation or model of the software system to be created. The model should provide the precise design information

needed for planning, analysis, and implementation of the software system. It should represent a partitioning of the system into design entities and describe the important properties and relationships among those entities [Sec 5.1 page 10]."

The standard goes on to further describe the need for attributes for each design entity. The purpose of this is to enable a method of reducing the software project into manageable pieces (design entities) and then provide a consistent method of describing each piece. The intent is not to define methodologies or method of description but to ensure that a software system can be expressed as a collection of design entities, each possessing properties and relationships. These properties and relationships are the attributes which should be present with each design entity. The following list describes each of the required minimum set of attributes.

**Identification** [from section 5.3.1]. *The name of the entity.* Each design entity must have a unique identifier.

**Type** [from section 5.3.2]. *A description of the kind of entity.* For example subprogram, module, procedure, process, or data store.

**Purpose** [from section 5.3.3] *A description of why the entity exists.* This section should designate the functional, performance and any special requirements.

**Function** [from section 5.3.4] *A statement of what the entity does.* The type of transformation performed on data by this entity or type of data stored.

**Subordinates** [from section 5.3.5] *The identification of all entities composing this entity.* Indicates the parent/child relationships in the design decomposition.

**Dependencies** [from section 5.3.6] *A description of the relationships of this entity with other entities.* Often depicted in Data Flow Diagrams (DFD), Structure

# SECTION 1 - INTRODUCTION AND BACKGROUND

Charts (SC) and Transaction Diagrams (TD). The interactions may involve the initiation, order of execution, data sharing, creation, duplication usage, storage, or destruction of entities.

**Interface** [from section 5.3.7]. *A description of how other entities interact with this entity.* These interfaces deal with the methods of interaction. Such items as communication via parameters, common data area or messages, direct access to internal data, input and output meanings, acceptable ranges, formats and error codes.

**Resources** [from section 5.3.8]. *A description of the elements used by the entity that are external to the design.* Information such as physical devices (printers,disc partitions, memory banks) software services (math libraries, operating system services, and processing resources (CPU memory cycles, memory allocations, buffers).

**Processing** [from section 5.3.9]. *A description of the rules used by the entity to achieve its function.* Such items as timing, sequences of events or processes, prerequisites for process initiation, priority of events, processing level, actual process steps, path conditions, and loop back or loop termination criteria.

**Data** [from section 5.3.10]. *A description of the data elements internal to the entity.* Describes the content, structure and use of data elements. Includes such items as method of representation, initial values, use, semantics, format, and acceptable values of internal data. Some examples are file structures, arrays, stacks, queues, and memory partitions. Typically described in data dictionaries.

# SECTION 1 - INTRODUCTION AND BACKGROUND

The definitions above are an abridged version of those described in IEEE Std 1016-1987. Readers who require the complete treatment should refer to the body of the standard.

Section 6 of the IEEE recommendations introduces the concept of design views. They are a method of representation of the design attributes described above. The table below [table 1 page 13 of the standard] is a fairly self explanatory mapping of design views to design attributes and example representation formats.

| Design view | Scope | Entity attribute | Example representations |
|---|---|---|---|
| Decomposition description. | Partition of the system into design entities. | Identification, type,purpose, function, subordinates. | Hierarchical decomposition diagram, natural language. |
| Dependency description. | Description of the relationships between entities and system resources. | Identification, type, purpose, dependencies, resources. | Structure charts, Data flow diagrams, transaction diagrams. |
| Interface description | List of everything a designer.or tester needs to know to use the design entities that make up the system. | Identification, function, interfaces. | Interface files, parameter tables. |
| Detail description. | Description of the internal design details of an entity. | Identification, processing, data. | Flowcharts, N-S charts, PDL |

**Recommended design views - Table 1.1**

( Table 1 page 13 of IEEE Std 1016-1987)

This concludes this section of the IEEE treatise. The purpose of the section has been to introduce the reader to the standard expectation for software design documentation. These guidelines will be referred to throughout the thesis providing, wherever possible, an

understanding of the strengths of Reverse Engineering and its weaknesses compared to these IEEE expectations. It should already be apparent that these recommendations deal with the Forward Engineering situation. Therefore some items expected for this purpose will clearly not be available when Reverse Engineered. Section 1.1.4 "Reverse Engineering" will deal with this aspect in more detail.

### 1.1.3 The Mesa language

Mesa is a Xerox proprietary software programming language particularly suited to the development of ultra large programming projects. This language has been in use in Xerox communities throughout the 1980's and has spumed a number of significant software products. Many people are familiar with the Xerox line of workstations and the high quality document processing software Viewpoint and Globalview; these products are written in the Mesa language. Many of Xerox's electronic printer products also use Mesa, which is an ideal language for this type of demanding real time application. The tools written for this thesis are also written in the Mesa language using XDE (Xerox Development Environment) as the tool workbench and operating environment.

Mesa is a very powerful language. The language has many similarities with Pascal including the attractive feature of user-defined data types that enables data structuring capability and data abstraction. [# 45]. Mesa, however, extends the capabilities of Pascal to make it eminently suitable for the creation of ultra large software projects with multiple co-operative development personnel working congruently. The Mesa Course book states it like this:

# SECTION 1 - INTRODUCTION AND BACKGROUND

> "Standard Pascal does have significant shortcomings in terms of writing a large system: there is no way to break the system down into small separately compiled units and then integrate them into a consistent whole. This prevents the compiler from checking the type correctness of actual parameters in distinct units, inhibits the development of "libraries" to extend the language, and generally complicates the implementation of large systems constructed by a group of programmers. Further more, standard Pascal does not support dynamic array bounds: it is difficult to write general routines that process arrays of different sizes. Standard Pascal has no exception handling facilities and does not support concurrent processes."

A number of interesting items are mentioned in the paragraphs above. We will expand on these briefly to ensure we understand the importance of the Mesa differences.

First the "Strongly Typed" concept. The Mesa compiler is sensitive to the type checking of data items in different programming units. Programmers can develop separate modules and then bind them as part of the compilation process.The binder enforces the strong type checking. Thus modules will not have type mismatches from inconsistent declaration of data types. Comparatively, Pascal programs either need to be one monolithic program and, therefore, ensure data types are consistent, or break down their programs into multiple smaller modules introducing the inherent risk of unreliability due to data type mismatches.

Secondly the "block structured" concept. Mesa has significantly more modularization power than Pascal. Interfaces and program modules are an inherent structural part of the language. The

# SECTION 1 - INTRODUCTION AND BACKGROUND

interface has no executable code. Its purpose is to collect together classes of objects into an abstraction. It also provides the vehicle for the type checking facility described earlier. The interface declares the imports and exports of modules external to the abstraction, it defines and provides the boundary of operation for modules that do not export. Types, constants and procedure headers can all be declared within the interface. The interface contains the information to allow the compiler to type check and ensure that program module implementations are present and that data types are consistent. The implementation modules are where the computational implementation code is written. Implementation Modules can only service clients when accessed through the interface definitions. These are bound at compile time. The interface, once declared, allows programmers to work on the implementation module without affecting client implementation work which can go on independently. Thus information hiding is facilitated by the client / service relationship so developers can be assigned separate tasks in parallel.

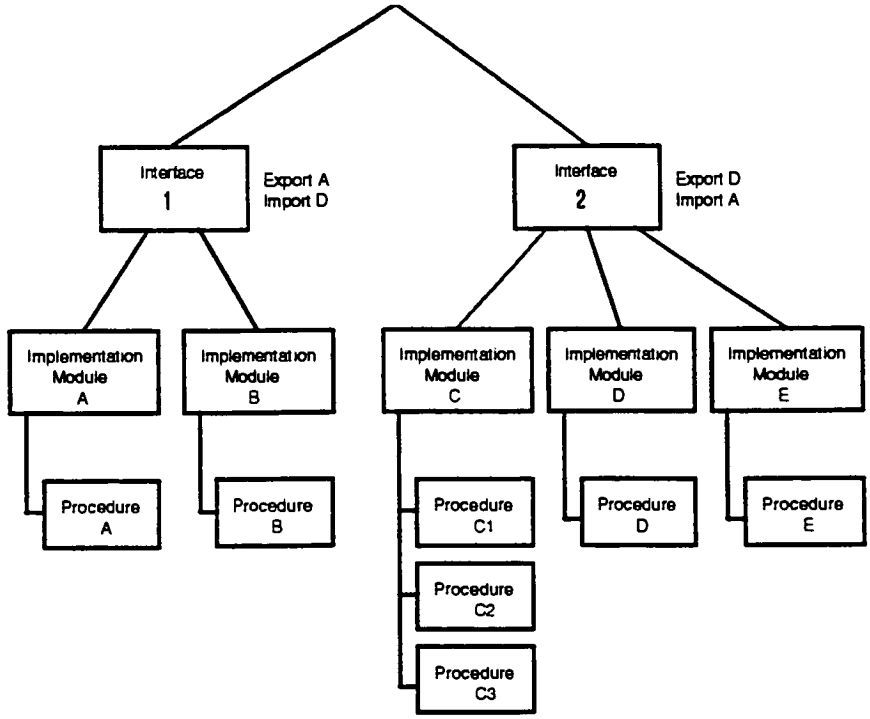



**FIGURE 1.1.3 - Mesa interface structure representation**

# SECTION 1 - INTRODUCTION AND BACKGROUND

The diagram above [Figure 1.1.3} shows a simple representation of the interface / implementation module concept. Implementation modules that are not on the export list can only be used within the scope of the parent interface, i.e., Implementation modules C,D and E can all access each other as clients or services within the confines of Interface 2. Access external to Interface 2 can only occur if an explicit export clause is defined. As the diagram shows there are implementation modules that can be exported beyond the confines of Interface 2. Module D is shown as exportable by declaration of an export clause within Interface 2. Note that Interface 1 has to have a complementary Import clause to enable the client service relationship to be established. With the diagram as shown, procedures within module D can be accessed by modules B and C if required. Similarly the Module A is accessible to the Interface 2 modules. Modules C and E are not exportable beyond Interface 2 and module B is not exportable beyond Interface 1, because no export/import clauses are present in their respective interface modules.

This export capability is only true if the procedures within the implementation modules are declared 'public'. If the procedures are 'private' then a further constraint on accessibility holds. In this case procedures are only available to other procedures within that single implementation module. This is illustrated by implementation module C where several procedures are indicated. If these procedures are declared as 'private' then access control is limited to within the implementation module C. In this case procedures C1, C2 and C3 cannot be accessed outside of the implementation module C boundary. Therefore module D and E will have no access capability to procedures within module C. Thus levels of abstraction and confinement of scope similar to the constructs of structured design can be represented well in the Mesa language through careful choice of import / export access controls.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Mesa also provides facilities for handling exceptions. Mesa uses signals which can be raised when an invalid condition occurs. For example, invalid inputs and allocators out of space are exception conditions. Exceptions are dealt with by use of 'handlers'. These are written in a distinctive syntax known as 'catch phrases' which execute a user defined clean-up sequence when exceptions occur. When a signal is raised, the Mesa run-time controller searches for the handler in the current procedure. If it does not find a catch phase, it will travel up the call stack to the next one, and so on up the call stack until it finds a valid handler. Uncaught signals can cause ambiguous system operation and therefore, to minimize risk of unreliability, care must be taken to provide catchall routines to handle unforeseen situations.

Mesa also has some very powerful concurrent processing capabilities. The keyword FORK enables a second process to start executing in parallel with the one where the FORK command occurred. Multiple processes can be running concurrently in a single system using this feature. Many procedures can FORK off new processes all running together in parallel. On occasion, however, programmers may desire that concurrent processes come together again at some convenient point. A good example of when this might be needed is when a program wants to perform a heavy mathematical calculation. The first process FORKS off a second procedure to complete the computation while the first process continues to operate on other data. After a short time the mathematical computation completes and a rendezvous with the first process is required. A JOIN command accomplishes this. Thus the keyword JOIN can be used to enable the synchronization of concurrent processes. When two processes are running after a FORK, then a JOIN will cause whichever process completes its computation first, to wait until the other process has completed execution, and then 'join' the two concurrent processes back into a single

executing process once again. Concurrent processes can share the same data address space and global variables. Thus a mechanism to prevent data corruption must be provided if integrity is to be maintained. Mesa uses the traditional form of monitors, including the monitor lock facility for ensuring proper mutual exclusion and protection of data in concurrent situations.

Mesa therefore is a very powerful language for large scale development of embedded real time systems. The strong type checking, package body / interface structure and extensive concurrency capabilities make it eminently suitable for the design of embedded control systems such as those in electronic printer products. These products require massive amounts of computational power in small time slices constrained by the process speed required to produce complex laser images on fast moving paper. Concurrent CPU's executing concurrent program code are often employed to achieve these demanding goals. Mesa provides the inherent capabilities to do this effectively. Also, large programming teams can work together in relative independence using the Mesa language and the XDE networked environment. The powerful abstraction feature provided by the interface / implementation module concept is ideal for this purpose. Mesa has proved to be a very effective tool and thus has large amounts of product code already existing in products currently marketed by Xerox

### 1.1.4 Reverse Engineering

Reverse Engineering has its origin in the world of hardware [# 14]. The idea is to take existing designs and figure out their strengths and weaknesses. In military situations, this can ensure that you maintain national security by understanding your enemies equipment and staying ahead. In

# SECTION 1 - INTRODUCTION AND BACKGROUND

the commercial world, this provides a competitive advantage leading to higher return on asset and a larger market share over time.

In software , Reverse Engineering can be applied for exactly these reasons. However, the more usual event is the pursuit of maintenance objectives. Reverse Engineering is the part of the software lifecycle that allows one to reconstruct code for the purpose of adding new features or improving an existing design. Reverse Engineering however is not normally condoned, it is an exception case where proper documentation is not available. The second process, improving on existing design, is called Re-Engineering. Although a very closely related topic, it will not be included in our discussions for the reasons discussed in the next paragraph. What then is Reverse Engineering? Reverse engineering is defined in the article 'Reverse Engineering and Design Recovery: a Taxonomy [ # 14 ] in the following way:

- Reverse Engineering is the process of analyzing a subject system to
  - Identify the system's components and their interrelationships and
  - To create representations of the system in another form or at a higher level of abstraction.

Reverse Engineering, then, does not involve changing the subject system or creating a new system based on the reverse-engineered subject's system. It is a process of examination, not a process of change or replication. Of course, having made this caveat, many of the outputs of Reverse Engineering are eminently suitable for the purpose of Re-Engineering. Many CASE tool vendors are beginning to bring together all three concepts, Forward Engineering (the usual design process), Reverse Engineering and Re Engineering. The article 'A CASE for Reverse Engineering [# 5] puts it like this:

# SECTION 1 - INTRODUCTION AND BACKGROUND

> "In targeting the maintenance, enhancement, and migration of existing application systems, the next generation of CASE products must open the door to a more reflective, cooperative mode of development. Such a design process is not the one way street of top down design, but assumes a give and take whereby changes can be propagated up and down at any point in the design process. The Re-Engineering cycle chart below provides an architectural view of this new CASE world, which features both Forward and Reverse Engineering."

The relationship between these three fields is clearly indicated in the diagram below taken from the CASE for Reverse Engineering article:

# SECTION 1 - INTRODUCTION AND BACKGROUND

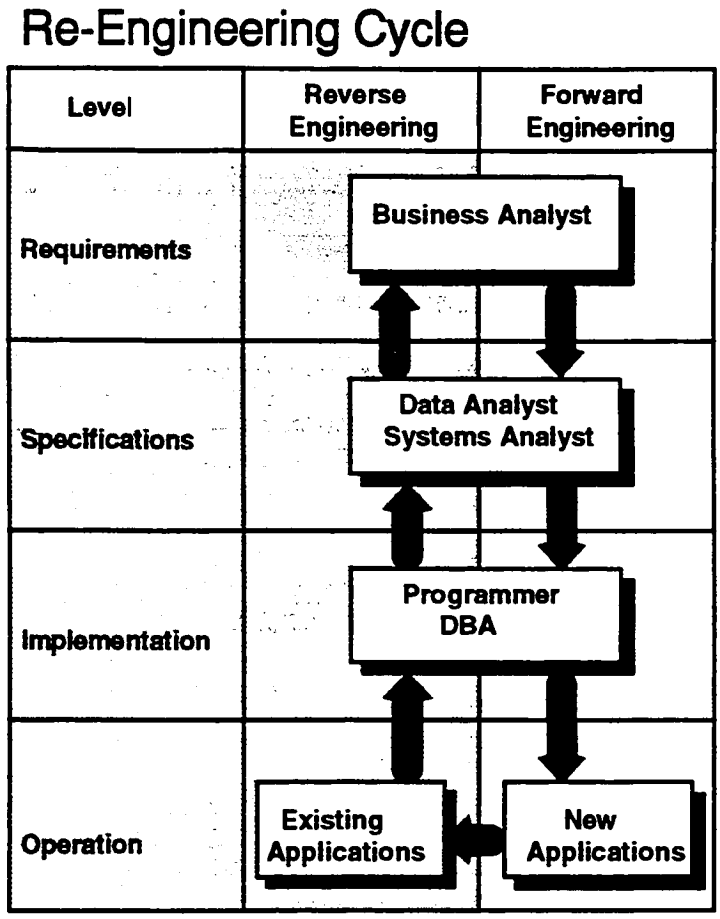



**Figure 1.1.4 - Reverse Engineering and Forward Engineering**

This diagram shows the close relationship between the two available directions of travel. Forward Engineering, going through the traditional channels of Requirements, Specification and Implementation on its way to active use in customer hands. Reverse Engineering, a return journey from Implementation back to requirements. When in operation the software is continually enhanced, usually in response to customer demands. The "new applications" often come in at the bottom as shown, and the temptation is for the programmer to 'enhance' the product at that level. Enhance is emphasized because often the outcome is degradation. Many organizations are

not equipped to go backwards up the Reverse Engineering track and, consequently, system architectures start to suffer from the patch syndrome. Patches are analogous to 'band-aid' where the purpose is to fix up the product temporarily. No attempt is made to consider the original specifications or requirements to see how the change (patch) affects the system as a whole. Consequently the patch may work well for the one particular fix in mind but side effects of the change may occur. As other patches to fix further problems are added the architectural integrity of the product starts to disintegrate. The rules for the higher levels of abstraction are violated, data structures are kludged and system interactions become erratic. Patching in this fashion will, eventually, lead to forced system retirement because the cost of fix and side effect removal becomes increasingly expensive as structural integrity diminishes.

Reverse Engineering then provides the opportunity to travel back up toward the requirements when original design documentation is unavailable. Thus systems can be studied at the abstract level of design and requirements to ensure that system integrity can be properly maintained as changes occur. An ideal system would achieve this. However, the state of the art today allows us to travel up only to the specification level in the main, unless extraordinary efforts have been made to document the source code with requirements issues included in the comments. This is not usually done nor thought to be beneficial.

With this model in mind, we are now about to start the journey of understanding how far we can travel backwards into the specification from the source code. Clearly, many assumptions made at design time guide the structural makeup of any operational software product. Much of this information cannot be captured from the code. The type of system modelling at this level does not map well into code comment structures and, therefore, is documented at higher levels of

abstraction. Indeed, the traditional Forward Engineering model encourages this kind of information to be kept separate from the code. However, there is a great deal of information that can be extracted directly from the code. Process names, structure, data flow, data type / structure are all examples of what can be recaptured by Reverse Engineering. Creation of a toolset needs to focus on this information to determine its availability for automatic extraction from source.

What then can be drawn from the code in the case of Mesa? This question will be answered by considering the expected minimum set of design documentation expectations from section 1.1.2 evaluating the potential candidates versus those items that will not be extractable in any automated form. The thrust of our thinking at this stage is automation. What software tools can be written to extract the data required for the specification reconstruction. The table that follows evaluates each design attribute in turn, projects an expectation for extractability and follows through with an explanation of the reasons for this projection.

| Design attribute | Extractable | Explanation |
|---|---|---|
| Identification<br>The name of the entity | Yes | Mesa code has distinct syntax for procedures, implementation modules and configuration files. These can be extractable automatically. |
| Type<br>A description of the kind of entity | Yes | Procedures and modules are clearly defined in Mesa. The interfaces will help us differentiate between subprograms, full modules, procedures, processes and data stores. |
| Purpose<br>A description of why the entity exists | Partial | The intent here is to study the content of the comments section of each module and interface. This should yield information that is relevant to the details of function, performance and special requirements. Perhaps a good Reverse Engineering practice would be to include special code commenting structures to facilitate automated identification of relevant fields. |

# SECTION 1 - INTRODUCTION AND BACKGROUND

| | | |
|---|---|---|
| Function<br>A statement of what the entity does | Partial | Similar methodology comment as for purpose. Code writers generally comment fairly well in program headers on the type of transformations performed. However, this is not usually well controlled or consistent in level of content. |
| Subordinates<br>The identification of all entities composing this entity | Yes | Parent / child relationships are well documented in Mesa source code. Extraction of this information enables the construction of structure charts. |
| Dependencies<br>A description of the relationships of this entity with other entities | Partial | Extracted Structure chart information can help with this. The intention is to extend this to Data Flow Diagramming. This involves the definition of a number of rules and paradigms. These will be detailed in section 1.3 where the theoretical base for this will be developed. Real-time control information will be ignored for this thesis. See section 1.1.5 where the scope is discussed. |
| Interfaces<br>A description of how other entities interact with this entity | Yes | Mesa has strong interface structure which should help yield this kind of information. However, complexity will be introduced by the concurrency factor. Fork, Join and the semaphore operation will greatly complicate the control implications. Again, these interactions will be ignored as detailed in section 1.1.5 where scope is discussed. |
| Resources<br>A description of the elements used by the entity that are external to the design | Yes | The prognosis at this time expects that much of this can be captured. The inputs and outputs from any particular module are identifiable. In some cases, partial understanding of the resource implication may also be available from the code declarations. CPU cycle allocations to processes are not relevant at this level. |
| Processing<br>A description of the rules used by the entity to achieve its function | Partial | This kind of information is often missing at the code level. The design determinations were made to structure the design. The information then rapidly gets out of date as the software maintenance complexities rise. |
| Data<br>A description of the data elements internal to the entity | Partial | Much of this information can be captured. The intent in this thesis is for the tools to produce 'data dictionaries' detailing this content. |

**Table 1.1.4 - Analysis of source code, design extraction expectations**

A brief analysis of the table above shows some interesting observations. If the explanations are indeed accurate, then a great deal of design information is available from the source code. A

quick scan of the extractable field in the table indicates that five of the ten items are designated 'yes'. Secondly, the other five items also show potential for some level of information extraction. This indicates that a large amount of data is available from code, given appropriate extraction techniques.

Current projections of the level of effort expended on software maintenance today, indicate that, of all available software professionals 70-90% are working on system maintenance [# 24, # 23, # 14]. This could possibly explain the current interest from CASE tool companies in tools that can perform automated Reverse Engineering. Clearly, development of helpful tool sets that automate the analysis of this maintenance effort will bring about the greatest cost savings, at least in the short term, for the software profession. Capturing systems in Reverse Engineered form will also encourage the reliance on CASE tools and speed up their adoption for Forward Engineering development. If this hypothesis is true, then the key to CASE adoption on a wider basis will come through Reverse Engineering. I believe that Forward Engineering and Reverse Engineering need to form a partnership that uses the best from both areas. Reverse Engineering has the inherent advantage that the representations are, by definition, up to date and accurate. Forward Engineering has the advantage of being the proven correct method for the development of quality software systems.

### 1.1.5 Scope

The scope of this thesis is now beginning to be defined. Clearly, the intention is to develop a set of software tools that automatically extract design information from the Mesa source code. The theory portion of the thesis discusses the details of relevant data extraction from the Mesa code plus potential graphical display schemes. However the implementation phase of the thesis

involves only the extraction of the data from the code. This segmentation has been chosen to ensure that the thesis illustrates the effectiveness of extraction. The theory is proved if the data can be shown to be available. At the completion of the thesis, textual representations of the extracted data were provided automatically by the tools. Hand drawn graphics were also drawn from these to illustrate the utility of the potential graphical display schemes.

There are a number of other exclusions to the scope of this thesis. First, the intent is not to address the real-time issues. This decision was made after much consultation and reading. Real-time methodologies are still new and in many cases unproven. Also, the real-time aspects are secondary in priority. This activity looks to be a good avenue to pursue for further academic study which would readily build off the static analysis portion that is the subject of this thesis.

Secondly we will discuss the availability of "off the shelf" tools. There are a number of graphical tools available that can produce adequate representations of some of the items being pursued. McCabe Associates [ # 30] and Cadre Teamwork [ # 11] have vastly different, but yet highly applicable, capabilities in this regard. The two companies have been approached provisionally and each is willing to provide interface information to their tool sets. Therefore, a valid approach would be to provide data structures that can be interpreted and displayed by their tools. This approach has many advantages. For example, the tools are widely used in industry and, therefore, will evolve over time with features that will enable us to stay current. Also, the comparison of the results will have a wider benchmark base due to the adoption of facilities previously used on different language platforms. The thesis implementation spent no time concentrating on interfacing with these tools.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Thirdly, we will discuss the use of lexical analysis tools. The Mesa environment, XDE, has many lexical analysis tools available in the rigorous six pass compiler. The thesis implementation phase investigated the use of these compiler utilities. It was found that the first pass of the compiler was very useful, separating out tokens for further analysis. The first pass only was used in the implementation.

This thesis attempt represents a considerable amount of work, even when given the caveats indicated above. There was a strong driving force to develop these tools due to their obvious utility to the Xerox Corporation. The intent was to extract as much information as possible to satisfy the expectation of IEEE std 1016-1987. Tools were written in the Mesa language to automatically extract information from Mesa source code suitable for producing Structure Charts, Data Flow Diagrams, Data Dictionaries. These tools are described in detail in section 2.

# SECTION 1 - INTRODUCTION AND BACKGROUND

## 1.2 Previous work

This section deals with some of the articles, books and references read in preparation for this thesis. Section 1.2.1 deals with specific articles of interest, and section 1.2.2 illustrates the search process used.

## 1.2.1 Focused articles

This section deals with previous work documented through books, articles and other references. This is by no means an exhaustive treatment, but a development of key ideas based upon work done by others. Many other items were read. These tended, in the main, to confirm the utility of items detailed below. Should the reader require further information, then please refer to the Bibliography in section 4.

### 1.2.1.1 History

The article that gives the best historical perspective is the "Software Design - Tutorial Series 5" [# 20]. This article takes us from the beginning of software as an engineering discipline and the growth in size and complexity of programs. The point the author is making is that Software Engineering is a relatively new science and that design methodologies are even newer. Thus it can be understood, in this context, how systems have been implemented without large amounts of supporting design documentation. Only now (or at least recently) is documentation understood to be a key element of the process.

The article explains that in the early days, first generation computing machines were devoted to the Von Neuman concept of stored program machines. Hardware technology was based on vacuum tubes and magnetic storage was limited to magnetic core with 2048 words of storage.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Those were the days of the Univac 1 and the IBM 701. With these machines, the complexity of the customer requirements was small, and the operational space was not the major challenge. Software designers were usually single individuals who knew the customer, the application, and all the interfaces to the hardware. Designs were in designer's heads, and no penalty was apparent from keeping it that way.

The second generation of computing did not begin to appear until the late 1950's. It was in these times that solid state electronics increased power by orders of magnitude over the vacuum tube technology of their predecessors. Machines such as the IBM 7070 and Univac M460 were in production. Many other computer manufacturers were beginning to enter the market, for example Honeywell, Burroughs and CDC. Cycle times were now in the 1 to 10 microsecond range. These allowed the advent of High Level programming languages such as Algol, Fortran and Cobol. Thus the complexities of programming were reducing with high level languages easier to use than the machine code of the early days. At the same time, system power was rising dramatically, thus extending the reaches of the operational space.

The power of the machines continued to increase as the 60's approached. However, software design still remained buried within the confines of the art of programming. It was becoming apparent that something had to be done to improve the reliability of these systems as they grew larger. Support requirements were expensive and manpower intensive. This was indeed a software crisis that had to be overcome. Computer Science books began to appear about this time, addressing the software crisis and ideas about how to overcome it. Fred Brooks, The Mythical Man Month, [# 10] being one of the most notable examples, illustrating many of the pitfalls and indicating a fairly pessimistic view of available solutions.

# SECTION 1 - INTRODUCTION AND BACKGROUND

The third generation of computing increased the pressure even more. Integrated circuits became the standard building block, with cycle times of 0.1 to 1.0 microseconds. Interrupt systems and the use of parallelism were further increasing the power of the machine. The article [# 20] describes how this first software crisis came upon the industry. " The euphoria brought on by the optimism of the salesmen and programmers diminished in the late 1960's as one after another of the ambitious large scale systems faltered, failed or produced less than satisfactory results. Software technology had not sufficiently met the needs of these large scale systems". The need was for an engineering discipline to address the reliability and maintenance problems of the day. The NATO Science Committee that met in Garmisch in late 1967 coined the term "Software Engineering". The recognition of this as an engineering term, bringing software into the forum as an engineering discipline, began the path toward the development of a disciplined and theoretical treatment of software development.

The need for an Engineering approach to software was quickly recognized. The terms Structured Programming and Structured Design became catch words among experienced professionals. Dijkstra published a key paper on the subject of 'Structured Programming in 1969". He introduced the concepts of structure in a system and abstraction of design principles to separate implementation from the process of design. Other concepts such as 'Information Hiding' were introduced by Parnas, the role of Systems Analysts, Top Down Decomposition, and Modular development started to take shape within academia. This was the era when minicomputers such as the PDP8 appeared, encouraging the decentralization of computing power.

These concepts of software Engineering finally started to solidify, according to the author (Enos,

# SECTION 1 - INTRODUCTION AND BACKGROUND

Judith C.), when the second software crisis came upon us in the late 60's. The first crisis dealt with programming languages and methodologies whereas the second solution to the crisis dealt with the management process. The famous waterfall process began to take shape with the segmentation of the discipline of large scale software development into the discreet steps of Analysis, Design, Coding and Checkout, Test and Integration, Operation and Support. These driving forces have given rise to the design practices we have today.

Some current recommended design practices use the Tops Down Process, sometimes called divide and conquer or stepwise refinement. The purpose of this approach is to provide a systematic method of system derivation. The Tops Down Process assists designers in the mental process of design. The principle is to start at the highest level of abstraction and then work down through the design, continuously partitioning it into lower levels until the module or procedure level is reached. This method provides an ideal bridge between top level design concepts and the implementation modules below. When fully developed, the representation is ideal for understanding the effects, of changes to a localized section of code, on the whole system . Graphical representation of a design provides a continuous picture of the whole and its relationship to the component parts. The graphic facilitates the representation of each level of abstraction in turn. The top level may represent the whole system. The next level, five components of the system. The next level eight components within each of the previous components, and so on. Sections of the graphical representation can be further subdivided at each level. In this way, designers can deal with varying levels of detail without loosing sight of the conceptual model developed at higher levels. The model also provides an opportunity for continuous review of progress and content throughout development. The concept of Requirements followed by Design Review, High Level Design followed by High Level Design

# SECTION 1 - INTRODUCTION AND BACKGROUND

Review, Detailed Design followed by Detailed Design Review and finally coding, are all supported by this approach. These concepts were developed clearly and concisely during the seventies [# 50] preparing the industry for their adoption as the norm in the 1980's.

The author (Enos, Judith C.) goes on to describe the methodologies used today. She states, very concisely and effectively, the usefulness of the main representation schemes available. A paragraph from the article is worth repeating here; " In all of these methodologies there is a common pattern that is characteristic of good design. First, a representation of a good software design is highly descriptive, and since it generally employs a graphic form of representation, it is often less ambiguous than a narrative description. Second, special care is taken in the use of terms - that is, the semantics that define the actual concepts of the design elements. Third, since design is a dynamic process the methodology must provide for iteration of decisions and alternative approaches. Fourth, since design moves from an abstract concept to a more concrete representation of the solution, the methodology must provide for levels of refinement. Finally, since real constraints of schedule time, staff capabilities, and hardware capabilities exist, optimization to performance and implementation constraints is required. Three basic types of graphic tools used in the design representation are the data flow chart, structure diagrams and table descriptions."

We conclude this section by restating the observation that many systems in operation today may have been in design prior to the widespread adoption of structured techniques. Although this is not an excuse for the lack of good documentation, it may explain why some systems are in this disadvantageous position.

# SECTION 1 - INTRODUCTION AND BACKGROUND

### 1.2.1.2 Representation Schemes

Representation schemes are the method of displaying the concepts of design. Three basic methods, Structure Diagrams, Data Flow Diagrams, and Table descriptions were mentioned at the end of the last section. They will be further illustrated here. We will use the text of 'Structured Design by Yourdon and Constantine' [# 50] and 'Structured Analysis and System Specification by Tom DeMarco' [# 19] as the reference volumes for this section. These volumes were written in 1975 and 1978 respectively, fitting in nicely with the historical perspective above.

**Data Flow Diagrams**

Data Flow Diagrams (DFD's) are used to represent the flow of data as it undergoes transformation through a software system. It contains no control information thus sequences of action are not illustrated. The representation is procedural, showing the flow of data from input to output. The system also represents the data repositories required and, indirectly, their content. This type of diagram is useful for analysis and often used by designers for the first time decomposition of the system. It does not illustrate the final structure of the system, but there is a strong correlation as you will see in section 1.2.1.3. Lack of control or sequence information is not usually a problem at this stage since these items can be added at the structure chart level.

Four basic symbols are used in DFD illustrations:

- The named vector, called a data flow, which portrays a data path.

  The bubble, called a process, which portrays transformation.

- The double straight line, which portrays a file or database.

  The box, called a source or sink, which portrays a net originator or receiver of data.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Combining these primitives to make a DFD, we get a picture which might look like this:

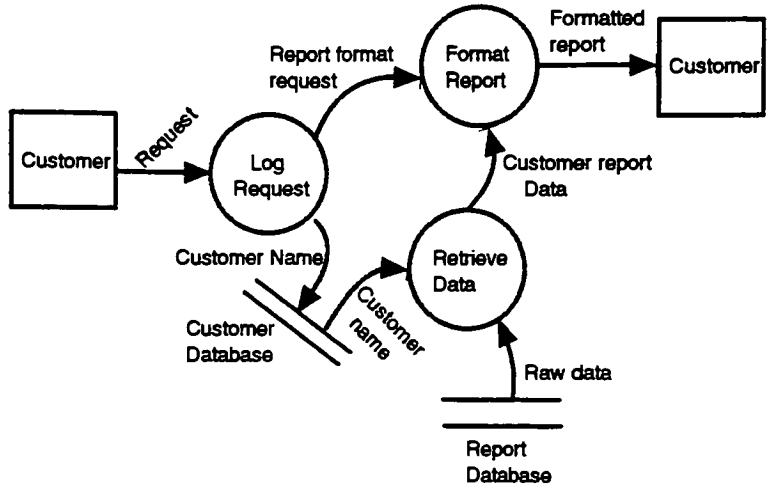



**Figure 1.2.1.2.1 - Simple Data Flow Diagram illustration**

The diagram above illustrates examples of the four primitives mentioned above. The Customer is the originator (source) and the receiver (sink) in this diagram. This could represent a terminal where a report is requested and then printed to the terminal. The data items are illustrated by the vectors or arrows. Request, Customer Name, Report format request, Formatted report, Customer report data and, Raw data, are all examples of data flowing between transformation processes. Log Request, Retrieve Data, and Format Report, are each examples of transformations. Some data is stored within internal databases. Customer Database holds customer name information, and Report Database holds the information required to construct a report. Note that for the purpose of simplicity, no way of entering data into the Report Database is shown.

Modelling a system using this representation scheme allows you to examine the transformations and data operations thus determining what is missing. Several iterations may be required to produce a workable system and several people may want to review the representation. The DFD

# SECTION 1 - INTRODUCTION AND BACKGROUND

shows a very good method for representing thought processes that can be communicated quickly and effectively to others. The scheme is a real picture of a workable system, sometimes known as the big picture. It can be used to model real situations to determine if the logic flow holds up. A large amount of system modelling can be done with this representation scheme, enabling designers to gain confidence that they have a system that will work when implemented.

Data Flow Diagrams, as previously mentioned, can represent multiple levels of detail. The diagram above [Figure 1.2.1.2.1] would be known as a 'Level 0' diagram. A higher level DFD would be the 'Context Diagram'. This diagram would simply show the system as one bubble with the three processes above assumed to be a hidden detail within it. You would normally start with the context diagram, then work downward to level 0. The context diagram for this system would look something like this:

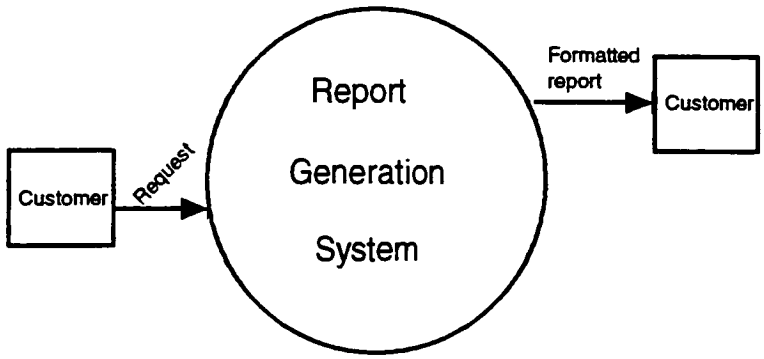



**Figure 1.2.1.2.2 - DFD Context Diagram**

The context diagram shows less detail than the level 0 diagram. Note, however, that the input data and output data of the context diagram are exactly the same as for the level 0 diagram. This is the principle of hierarchy within DFD's. First, start with the context diagram, then break this

# SECTION 1 - INTRODUCTION AND BACKGROUND

down into major transformation bubbles as per figure 1.2.1.2. Then, if desired, break each of the transformation bubbles down further. However, the data input and output should be exactly traceable to the next higher level. Suppose we decided to go to a level 1 DFD. Let's choose the process 'Format Report' to break down further. Note that the data in and data out would remain the same. Therefore, 'Report format request' and 'Customer Report Data' would be the inputs and 'Formatted Report' would be the output. The level 1 DFD then could look like this:

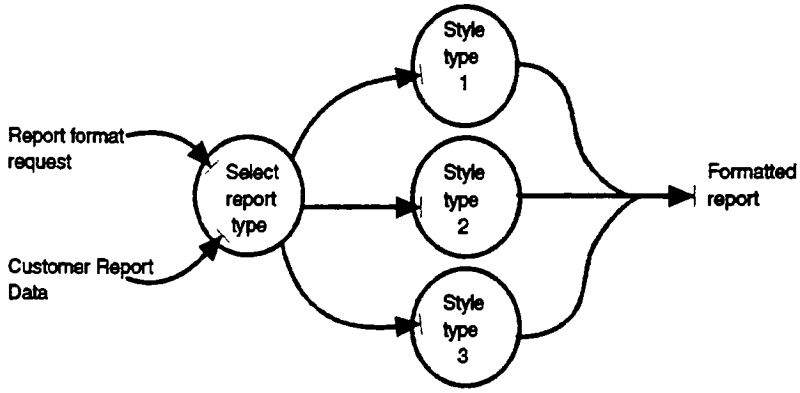



**Figure 1.2.1.2.3 - Level 1 Data Flow Diagram**

The diagram above illustrates a further level of detail. Data Flow Diagrams can be used in this way to decompose the system sequentially, starting at the context diagram and working down through to the detail. The bubbles eventually become simple enough to become code implementation modules or procedures. This is the goal of design, to illustrate the abstraction of the system. Thus, system level interactions can be studied and made robust prior to code implementation.

# SECTION 1 - INTRODUCTION AND BACKGROUND

### Structure Charts

Structure charts (SC's) are used to illustrate the hierarchical organization of a system in graphical form. There is a strong correlation between Structure Charts and DFD's. The SC defines the relationship between the modules and the interfaces between them. In many cases the SC can be produced from the DFD. The transformations of the DFD become the modules or nodes of the SC, and the data flows become the interfaces between the modules.

Structure charts can be considered as analogous to system organization [# 50, Page 25-30] within a large company. The structure of a company is based upon the relationship between subordinates and superiors. Organization charts start with a top box, usually the president, with several reports (the VP's), each of which, in turn, have several reports. With each one having several reports below them, on and on until the lowest member of the organization is reached. In this way, the organization controls flow of data from the lowest levels to the highest.

There are a number of excellent analogies to software design and organizational effectiveness that hold well, according to Yourdon and Constantine. For example, when a manager has more than 7 or 8 reports, his effectiveness reduces. The scope of effect of each manager in a good organization is limited to his area and the interactions below. Finally, a structure with too many levels becomes very inefficient for communication. All these examples are relevant to software design practice. Structure Charts and organizations, then, have much in common. This makes understanding the SC concepts easy to comprehend intuitively as we all have real-World knowledge of human organization systems to compare it with. A cautionary note: other software professional have not seen all these analogies as proper constraints upon software design.

# SECTION 1 - INTRODUCTION AND BACKGROUND

To understand how Structure Charts work in more detail, we must briefly study their primitives. There are two major categories, the modules and the interfaces. We will consider each in turn. First the modules. There are three basic kinds of modules. Afferent, Transformation, and Efferent. Afferent modules concern themselves with the input streams of data. They are modules that receive data from subordinates and pass it on to their superior modules. Efferent modules concern themselves with output streams of data. They are modules that receive data from their superiors and pass it on to their subordinates. Finally, Transformation modules concern themselves with the changing of input data, received from its superior, from one form to another, passing the result back upwards to the superior again. Afferent and Efferent modules may have some transformation content but, because of their position in the data flow structure, their designation does not change. The diagram below illustrates the module types and their positioning within the SC.

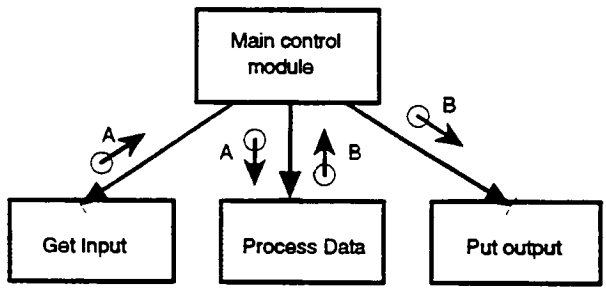



**Figure 1.2.1.2.4 - Structure Chart initial example**

The diagram shows a number of interesting things. Let us first consider the Afferent, Efferent and Transformation modules. These are 'Get input', 'Put output' and 'Process Data' respectively. Secondly, the 'Main Control Module', which has the responsibility for control of processing order. This item was not mentioned previously, but is always present at the top of any SC. It is clear to see from this representation why each module is so designated. The clarity is enhanced by the

# SECTION 1 - INTRODUCTION AND BACKGROUND

illustration of the data flow arrows between each of the modules. This is the interface illustration. The arrows indicate direction of data flow. That is, 'A' flows up to the 'Main control module, then down to 'Process data', which transforms the it to 'B', passes it back to the 'Main process module', and outputs 'B' via the 'Put output' process. There is another category called Control flow which has not been mentioned here because it falls outside the scope of this thesis.

How then can we use the DFD we produced in the last section to produce a SC? If we study the structure of Figure 1.2.1.2.1, we see that the elements of Afferent flow, Efferent flow and Transformation are all there. The Transformation modules are the clearest. They are indicated by the process bubbles 'Log Request', 'Retrieve Data', and Format Report'. The Afferent and Efferent modules are tougher to see. Examining the diagram, we see that data is input from the customer and output to the customer, these are the Afferent and Efferent data flows respectively, at least at the top level. We will designate modules to these functions in our SC. We also have a second level DFD in Figure 1.2.1.2.3. We can easily incorporate this into the SC by showing them as submodules to the 'Format report' process. I have used the heuristic that each database and input / output action require appropriate handler processes. Putting all this together produces the SC below.

# SECTION 1 - INTRODUCTION AND BACKGROUND

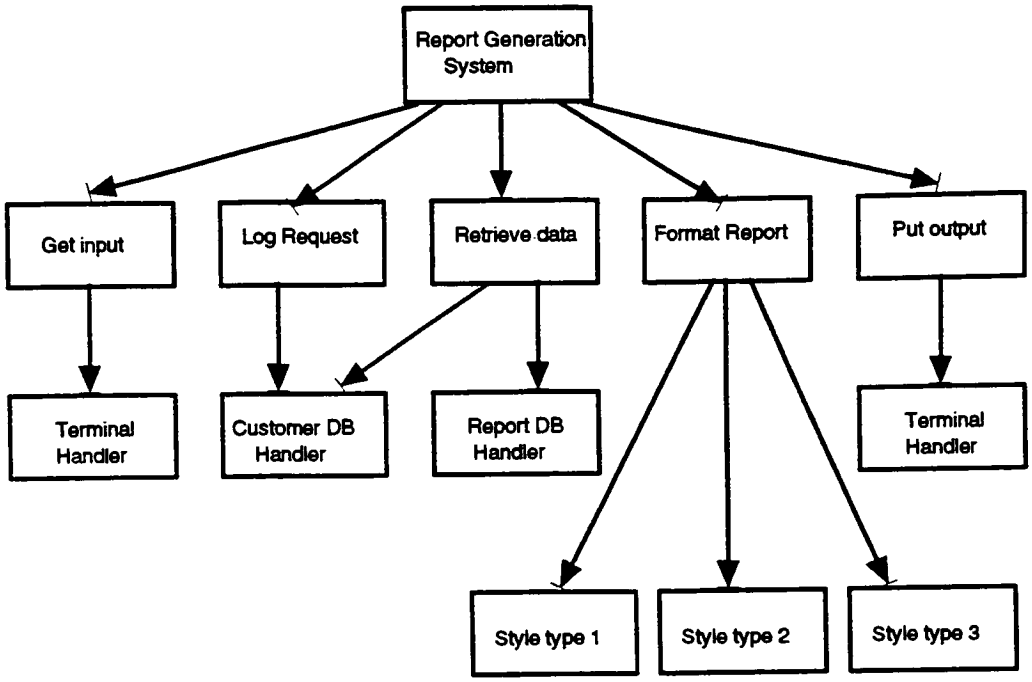



**Figure 1.2.1.2.5 - Structure Chart derived from our system DFD**

For simplicity at this stage, the Structure Chart above does not contain the data handling. When the data is added, the chart becomes much more complex in appearance. A couple of observations spring forth from the diagram. First, the 'Terminal Handler' is the same module for both input and output. This need not be the case of course. It is a matter of choice. However, writing of terminal input and output handling routines are well placed together. An option might be to further break down that module into an input submodule and an output submodule. Alternatively, the functions may be so similar that splitting them is unproductive. These are the kinds of choices that can be easily made when examining a design structure. Secondly, the process 'Format Report' spawns three subprocess. Do each of the subprocess perform work on the incoming data?. The answer, of course, is no. Only one of these is chosen with each invocation. These observations are touching on a level of detail I do not intend to go into at this

# SECTION 1 - INTRODUCTION AND BACKGROUND

stage. Cohesion and coupling deal with the first, and concepts of control with the second. Yourdon / Constantine and DeMarco go extensively into these concepts in the books referenced [# 50, # 19].

The data flows in and out of each modules are now added in to complete the picture. The diagram below shows exactly the same Structure Chart as before with the data now included. Note the convention indicating the flow upward or downwards. Some data items were difficult to fit in because of the length of the name. Refer to the key at the bottom of the chart to decode the acronyms.

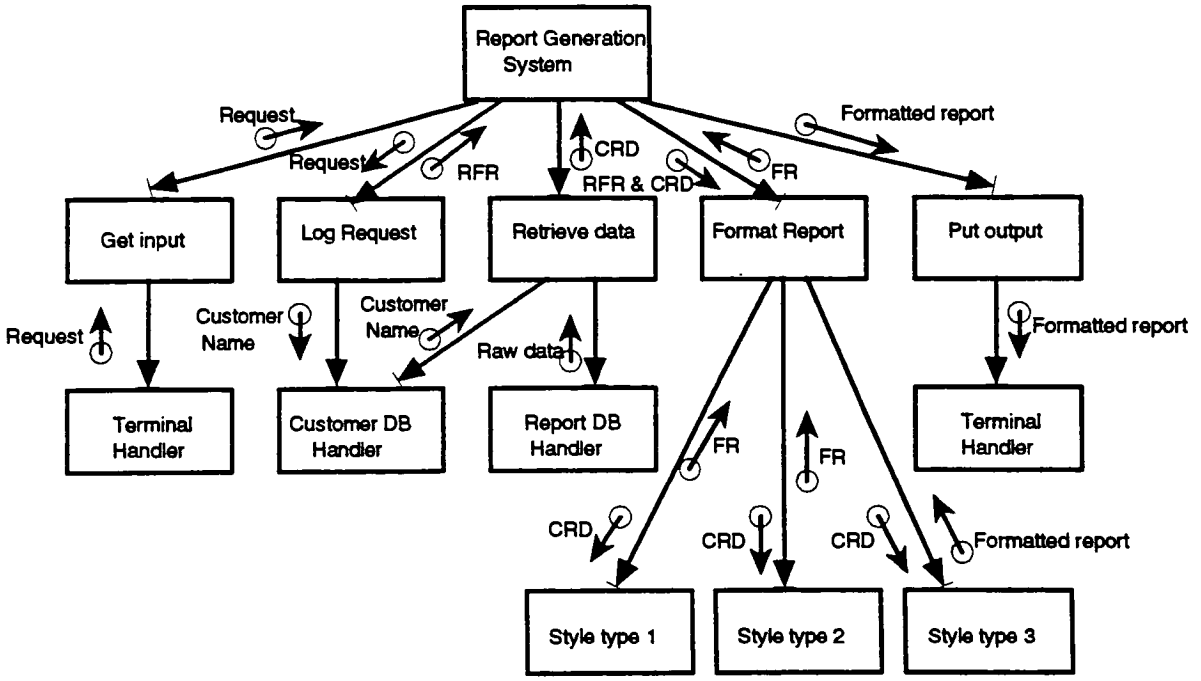



**Figure 1.2.1.2.6 - Structure Chart with Data Flows indicated**

# SECTION 1 - INTRODUCTION AND BACKGROUND

This concludes the section on SC's. From what we have discussed so far, it can be seen, that there is a forward path from DFD's to SC's, but it cannot be automatic. The hope is that this mechanism can be reversed in the process of extraction from the code. It looks promising. Code does contain structural information from the modules, procedures and configuration bindings, all of which are accessible. The code also contains the inter module data handling information needed to complete the SC. The process databases can also be seen through various semantics in the code. Transformation information is often available within the module header, or worse, buried within comments. With all this information available, meaningful representation may be possible on an automatic basis. The next section details an example of this being done with Pascal code.

#### 1.2.1.3 Extraction from Code

The next article, 'A Reverse Engineering methodology to reconstruct hierarchical data flow diagram for software Maintenance' [# 6], deals with the creation of Structure Charts and Data Flow Diagrams from Pascal code. The paper is based on research done in DIS (Department of Computer Science, University of Naples, Italy) and CRIAI (Consortium for Applied Research in Computer Science and Industry Automation, Italy). Note that the thrust of this article is to Reverse the discussions of section 1.2.1.2. Structure Charts are to be produced first then converted to Data Flow Diagrams using principles, or rules, defined in the above paper.

The authors begins by introducing the potentials for Reverse Engineering. They postulate that tools, in themselves, are insufficient to bring the required level of abstraction from code, without methodologies that support the constructs. They argue that, given appropriate rules, in

# SECTION 1 - INTRODUCTION AND BACKGROUND

conjunction with automatic extraction, advantages can be gained in the following areas:

| | |
|---|---|
| - In Maintenance: | Software comprehension, the implementation of changes, post maintenance testing |
| - In software Re-use: | documentation, classification, Adaptation, and integration of software components. |
| - In production: | the use in Forward Engineering phases of knowledge extracted from old software that cannot otherwise be used. |

The authors spend the first part of the paper describing the need to define expected outputs before embarking on the extractor tool path. Clearly, this is in the realms of architectural design or requirements for the tools. The options they suggest are for tools that produce 'decision support documents, to look at the characteristics of software', 'high level design documents, essentially SC's and DFD's', 'low level design documents', targeted at the internal construction of modules, and finally 'testing documents'. This thesis will confine itself to the high level design area as described previously in section 1.1.2. The article also cautions that the level and structure of information extracted will, in some part, be dependent on the method used to create, or Forward Engineer, the product. Design practices that do not follow structured techniques, will clearly not produce the well structured output desired. Haphazard design will produce haphazard Reverse Engineered output. This, however, is not necessarily a bad result. Poor constructs can be pointed out as opportunities for improvement by using the picture obtained by extraction.

The paper goes on to detail an experiment conducted to extract SC's and DFD's from a 117 procedure program with 350 variables written in Pascal. They pass over the production of structure charts as a trivial exercise, and mention only that '"the underlying structure of these SC's is a hierarchical tree of modules, in which modules activated by more that one superior are

# SECTION 1 - INTRODUCTION AND BACKGROUND

duplicated". The DFD's, they state, should have the same number of levels as the SC. They justify this by stating " since all the system modules must give rise to transforms distributed in the various DFD hierarchy levels, the choice of making these levels coincide with those of the SC's is justified". This is one of the major rules used as the foundation of this treatment. At this stage the article begins concentrating on the definition of the rules that they used to construct their DFD's and SC's. As these rules are explicit and essential to the rest of the treatment, they are duplicated below. The rules do not follow a sequential numbering system within the article and, therefore, one has been imposed.

| Rule Set | Sub Rule | Description |
|---|---|---|
| Rule 1 - SC's | 1.1 | **SC Modules.** Modules activated by more than one superior are duplicated. |
| Rule 2 - DFD to SC | 2.1 | **DFD / SC correlation.** DFD hierarchy levels coincide with each respective similar level of the SC. |
| Rule 3 - Data | 3.1 | **Data.** Static analysis should yield all constant, variable and formal parameters. |
| | 3.2 | **External outputs.** Static analysis should yield all candidates for output data stores. These outputs are toward file systems or standard I/O |
| | 3.3 | **External inputs.** Static analysis should yield all candidates for input data stores. These inputs originate from file systems or standard I/O |
| | 3.4 | **Internal outputs.** Static analysis should yield all candidates for output buffers. These outputs are toward other modules. |
| | 3.5 | **Internal inputs.** Static analysis should yield all candidates for input buffers. These inputs originate from other modules. |
| | 3.6 | **Non formal parameters.** Static analysis should yield all non formal parameters from each module and every activation of other modules within it. |
| Rule 4 - Nodes | 4.1 | **Terminal modules.** A terminal node in the SC is always represented as a transformation at the same level in the DFD. |
| | 4.2 | **Non terminal modules.** When represented in the DFD these modules are split into two representations.<br>***1 At the same level as the SC.*** The node includes the transformations within the body of the node plus all the subtree node transformations.i.e., Covers the scope of effect of the node.<br>***2 At the level below the SC.*** This is at the same level as the subtree nodes. The purpose of this inclusion is to show the transformation within the node body on the same level as the child nodes. |
| Rule 5 - Repositories | 5.1 | **Repositories.** Identify the data repositories at each level of the DFD. Completed bottoms up with the following subphases: |

**Table continued on the next page**

# SECTION 1 - INTRODUCTION AND BACKGROUND

| Rule Set | Sub Rule | Desription |
|---|---|---|
| | 5.2 | **Replacement** of all formal parameters of candidate variables with the actual parameters. Thus indicating the actual parameters used at the level of the module being represented.<br>***a) Multiple calls.*** Several actual parameters could be produced if multiple calls to a procedure are made from a parent procedure.<br>***b) Constants.*** In every replacement, all actual parameters that are constants are suppressed.<br>***c) Expressions.*** In every replacement, all actual parameters that are expressions are replaced only be the operands.<br>***d) Global.*** If a variable is global to a called module then the variable name is replaced by the formal parameter of the module that called it. This process works its way up the SC until the highest level of its global effect is found. |
| | 5.3 | **Reduction** of the variable list to the actual variables that certainly contribute to the formation of repositories.<br>***a) Terminal modules.*** All the local variables declared in the module are removed from the list. These clearly only have a scope or effect internal to that transformation node.<br>***b) Calling module body.*** The variables that are purely local to the body are eliminated. Again these are local to the node and need not be shown.<br>***c) Non terminal module.*** The union of all variables in the body of the module plus the directly called child module variables, then subtract the variables declared in the called modules. |
| | 5.4 | **Construction** of the repositories.<br>***a) Input repositories.*** If variables appear in the same input list at all times they can be replaced with a composite identifier. This allows us to construct the concept of 'data stores'.<br>***b) Output Repositories.*** If variables appear in the same output list at all times they can be replaced with a composite identifier. This allows us to construct the concept of 'data stores'. |

**Table - 1.2 .1 Rules for constructing Data Flow diagrams from Pascal**

This is clearly a very useful article. The table above shows a substantial selection of the rules we will require to construct our DFD's from the structure chart. The items can guide analysis of the Mesa code in the theoretical portion of the thesis. As discussed earlier, Mesa has some extensions over Pascal which may cause modifications to the above methodology, particularly the 'Interface / package body' structure. However, it would seem that most of what is provided here will have considerable utility.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Examining the table entries more closely, rules 1 and 2 set the scene. The SC avoids the crossover of lines by duplicating each repeated module. The level of hierarchy being identical, ensures that a one to one correlation exists, thus enhancing traceability and ease of understanding. Rule 3 deals with the analysis of data. The interesting idea in this section is the way the concept of internal and external inputs and outputs is used. These are not internal and external designation in the usual sense. The internal items refer to data shared with other modules, rather than to local data items. The external items refer to I/O devices such as file systems, printers etc. It would be good practice to define which are the designated external devices in order to facilitate this mechanism. Each Reverse Engineered system may have a different set, but these can be easily defined as the design clarifies.

Rules 4 and 5 deal with the actual graphic items to be displayed in the DFD. A methodology for the terminal and non-terminal nodes is introduced. Basically, terminal nodes differ from non-terminal nodes in that they have no calls to subordinate nodes. Thus, their utility is for transformation only. Non-terminal nodes clearly have subordinates and may also have transformation properties. The table shows two rules that have to be followed in representing these. First, at the peer level, where the transformation bubble is assumed to include those within the body and the transformation of its subordinates. Secondly, at the next level down (the subordinate level) a transformation bubble for the active code in the parent is shown alongside the child nodes.

The repositories section becomes quite complex (Rule 5). The concepts here are replacement, reduction and construction. Replacement refers to the proper allocation of formal parameters and

non formal parameters. The formal parameters are those defined in the procedure parameter list and the non-formal parameters are the actual variable names instantiated at call time. Multiple calls, constants, expressions and global variable types are all adequately dealt with. The global variable property is particularly neat. The concept is to use the upper level name for all global variables working upwards to its highest level in the SC. Thus, tracing of global variables across multiple levels of DFD is possible. Reduction deals with the trim down of the variable set to an appropriate number. At the terminal node level, local variables have no place in the DFD. The higher levels of the DFD, the non terminal nodes, also potentially have 'local data'. Clearly, this too should be suppressed. The non-terminal nodes should also collect together at the variables from their subordinates with the exception of those locally declared. Construction, deals with collecting together data items to try and recreate the design concepts of data storage, with collections of multiple data items. The authors use the notion of 'data stores' to refer to I/O plus file system files and 'main storage buffers' for other sets of variables.

The article has a number of detailed treatments of their concepts. A sample Pascal program is illustrated with four levels of DFD, and the appropriate Structure Chart. The derivation of each of the chosen data items displayed is given in all cases. The treatment is lengthy and detailed. Therefore, I recommend only dedicated readers pursue further study.

The article concludes by stating the success of the project. The data shows that from the 117 procedure Pascal program with 15 transformation modules, they were able to reduce the system to 89 procedures and provide highly consistent documentation. A drastic reduction in the number of indiscriminate uses of global variables, which are a considerable obstacle to program comprehension, was also realized. The results obtained from this exercise show that the DFD's

and SC's are a powerful tool in the maintainer's hands, enabling them to re-enter the Forward Engineering path for the purposes of both development and maintenance. The authors also caution that the documents automatically produced should not be entrusted to unspecialized personnel.

#### 1.2.1.4 Commercial graphics.

The Thesis implementation did not go as far as producing graphics automatically from the code. This section is added for the purpose of illustrating some of the methods of graphical representation available today.

This section will look into two of the best known tools for producing graphic representation schemes known in the industry today. The first, from McCabe Associates [# 30, # 32], a Software consulting company lead by the founder Thomas J. McCabe. The second, from Cadre the well known producers of the Teamwork CASE tool technology [ # 11, # 12]. McCabe is chosen because of his patented display method for design representation called the 'BattleMap' and Cadre is chosen because it has extensive DFD and SC drawing capabilities. The thrust of the thesis here, is to understand these graphic representation methods to ensure that data collected by extraction could utilize these approaches appropriately.

**McCabe**

Tom McCabe is famous for his software complexity metric work. He first published his findings in 1982 via the National Standards Special Publication 500 - 599, titled" Structured testing: A software Methodology Using Cyclomatic Complexity Metric". His emphasis, at that time, was on

# SECTION 1 - INTRODUCTION AND BACKGROUND

testing. The idea was to correlate graph theory with the complexity of a software module. By considering properties of the code, sequence, Case statements, If-then-else statements, Do-While statements, Do-Until statements etc, he was able to draw representative pictures of code. These pictures were automatically generated from a lexical analysis of the code. The pictures show the number of paths that needed to be tested, directly correlating to the complexity, and the decision nodes along the way. His tools are now able to show the predicates that require data for test cases. Thus, a comprehensive test case can be constructed almost automatically at the module level.

The company has since, while continuing on the former path, started to drive the process backward up the development cycle. They have extended their capabilities to 'Design complexity measurement and testing [# 31]. The methods of graphical display previously formulated at the module level, have been shown to have much utility at the inter-module level. The article just referenced, develops the technical theory behind McCabe's findings. The article talks to structured complexity metrics, complexity reduction techniques, and structured integration test. The interest in this thesis is limited to the graphical display capability and, therefore, will not concern itself with the complexity and test coverage theories. Additionally, the company flyer on "The BattleMap Analysis Tool™" [# 32] illustrates these graphical capabilities.

The BattleMap layout concerns itself with SC's and has no Data Flow Analysis capability. The intent is to analyze the fan in and fan out ot the modules and to discover discrepancies from good design. Extending then to test case generation which best ensure the robustness of the software package under examination. As many software systems are rather large McCabe Associates grappled with the problem of meaningful display of perhaps hundreds of software modules and

# SECTION 1 - INTRODUCTION AND BACKGROUND

their interconnections. This is where the BattleMap has its major strength. The technique is user configurable to permit zooming for detail representation and single page big pictures. Although I have four references to this technology, I see no attempt to study the interface parameters. The BattleMap is, however, capable of indicating complexities and maintainability ratings. It indicates the complexity with an associated table correlating module number, module name, complexity rating and chart location. It correlates maintainability by drawing structure chart boxes within boxes to indicate a quality rating; Maintainable, Unreliable or Unmaintainable and Unreliable. The ratings are again based mostly on internal module complexity. The big picture battlemap indicates these properties by color, quickly allowing design analysis personnel to see the scope of effect of a poorly rated module. The two charts below show the McCabe Structure Chart and the big picture display methods in BattleMap format.

# SECTION 1 - INTRODUCTION AND BACKGROUND

Standard Structure Chart
with overlaid McCabe notation

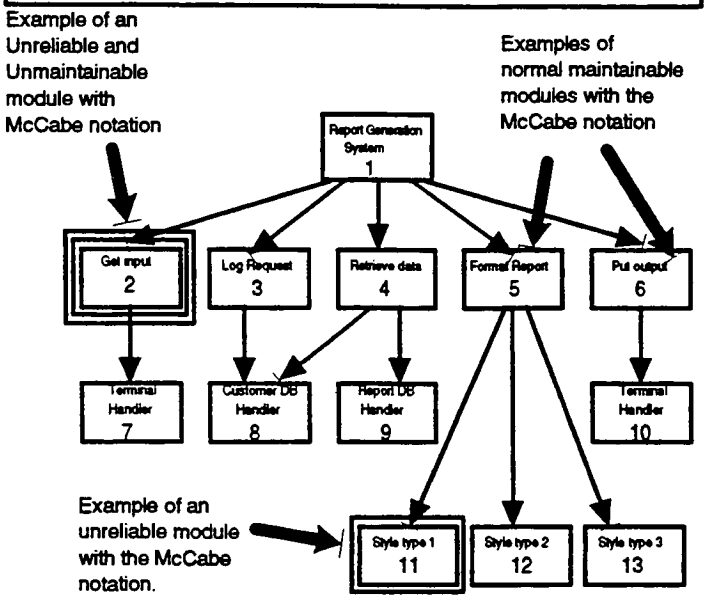



Figure 1.2.1.4.1 - Structure Chart using McCabe Notation

This chart shows the SC used in section 1.2.1. The McCabe notation for unreliability and maintainability have been arbitrarily applied. No structural weakness is implied.

Note in McCabes tools, the unreliable, unmaintainable designations are displayed in different colors. The colors transpose to the chart on the right by coloring the module numbers appropriately.

Battlemap big picture

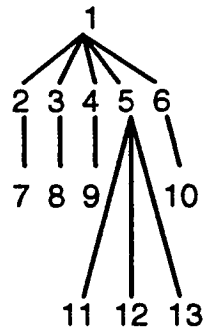



Figure 1.2.1.4.2 - McCabe Battlemap big picture notation

This chart shows the same SC as on the left Note the same structural picture is present, but much less space is taken. The chart takes probably less than a third of the space. It should also be noted thet the chart on the left is shrunk 50% from normal size. Therefore at least a 6 fold display space advantage is realized.

It was first thought that use of the McCabe tools as a graphics engine would be a very useful thing to do. It would allow the Mesa extraction tools to interface directly to them and thus bring Mesa into the commercial tool world at least in this arena. This may still be a useful thing to do. However, for the purpose of this thesis, the desired outputs of Structure Chart, Data Flow Diagrams, Data Dictionaries and Comment charts will not be realized through the McCabe route. The use of the big picture method, however, has considerable utility.

# SECTION 1 - INTRODUCTION AND BACKGROUND

**CADRE Teamwork**

Cadre has extensive facilities in its CASE tool suite.They have the capabilities for Information Modelling, Structured Analysis, Real Time Modelling, Structured Design and Ada. The concentration of this thesis is on the Structured Analysis and Structured Design section, although there may be considerable utility in the Ada section too. Mesa is a similar language to Ada with interfaces and package bodies defined separately. We will briefly discuss some of the observations relating to these last three features. [# 11], [# 12].

Teamwork has extensive capabilities in Structured Design. The CASE tools set has all of the Data Flow Diagram tools required for this thesis and many more. At the time of writing, no information is available as to whether these charts can be drawn by Data Driven Graphics. Data driven graphics is the capability of drawing the chart from a table of data. The reference articles only indicate that, at the DFD level, the user can only input requirements using the graphical editor. The tool does indeed create a table from the information displayed, detailing such items as modules, arcs, data items and the like. However, the reverse process is not mentioned.

There is more information about SC's. The product fliers indicate that, for the C language SD's can indeed be produced directly from the code. CADRE has indicated, in a preliminary discussion, that these tools will create the SC's given a table of appropriately formatted data. They would be happy to publish this for my use if Xerox have the tools in house. We do not currently have the tools and therefore this could be an obstacle. The SC capabilities of the CADRE offering are extensive. They have all of the basic display options mentioned in the thesis Representation section [section 1.2.1.2], plus some others. Included, are graphic display options for, Modules, Invocations, Data Couples, Connectors, Transaction centers, Iteration symbols,

# SECTION 1 - INTRODUCTION AND BACKGROUND

Text blocks, and Labels.

The third area of interest is the Ada section. It is worth spending some time in this section. Cadre use a different notation for SC's to help with the conceptualization of the Ada language constructs. As previously mentioned, Mesa and Ada both support the concept of interfaces (or specifications) and package bodies. Cadre use the notations introduced by Dr R.J.A. Buhr for the graphical representation of Ada structure charts [# 12]. "Buhr's notation provides a one-to-one mapping between a set of graphic elements and the corresponding features of the Ada language. Buhr uses Ada Structure Graphs (ASG's) instead of conventional program structure charts to model relationships that are specific to Ada systems". The diagram below illustrates an example of an Ada program using the Buhr notation as utilized by Cadre. The concepts of interfaces and package bodies are well illustrated.

# SECTION 1 - INTRODUCTION AND BACKGROUND

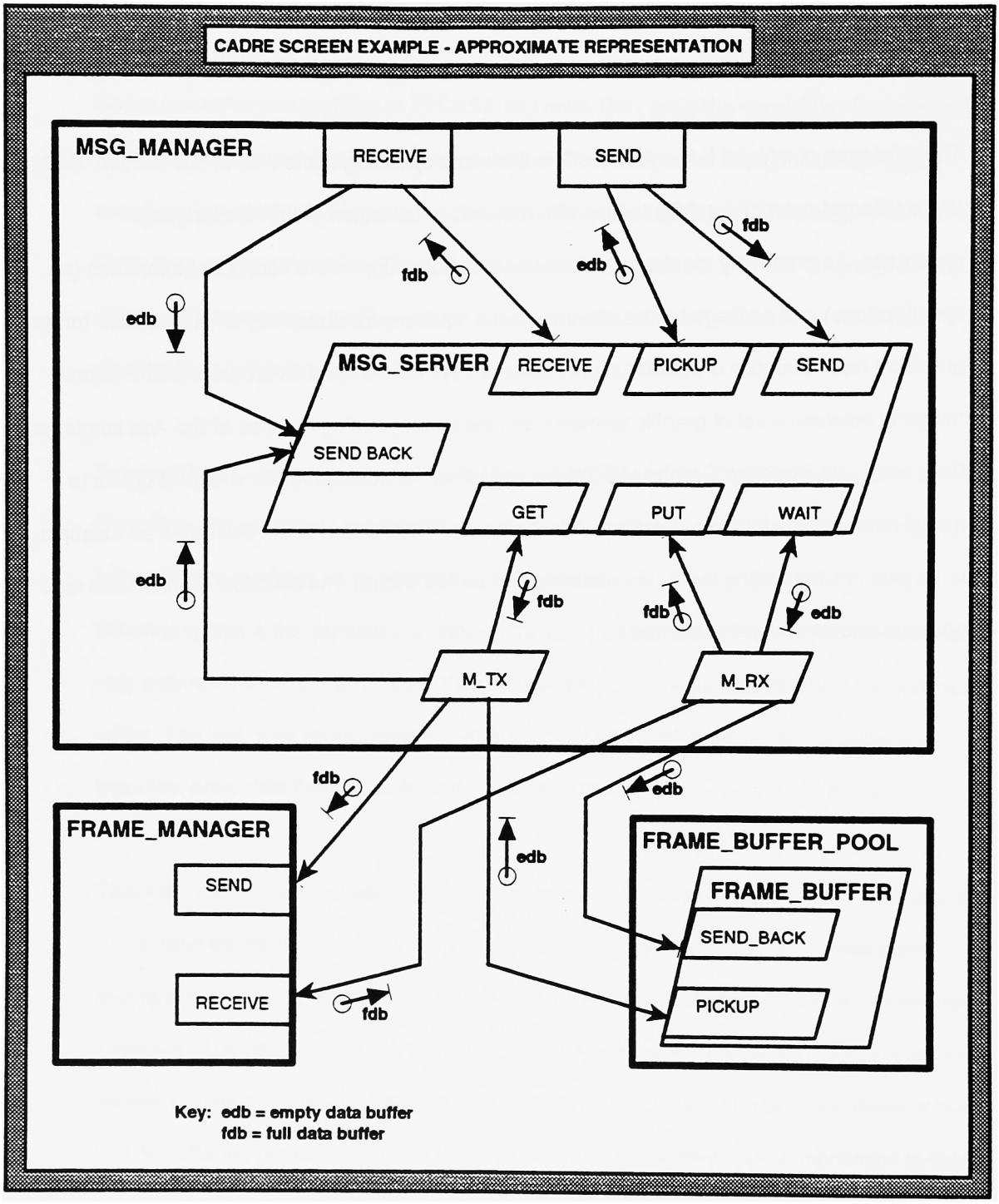



**Figure 1.2.1.4.3 - Cadre Ada Structure Chart Representation**

# SECTION 1 - INTRODUCTION AND BACKGROUND

The diagram shows a number of interesting things about representing Ada programs. First, the concepts of interfaces. In the program code for MSG_MANAGER, the first statements of the code define the specification portion. The procedures SEND and RECEIVE are declared here. These procedures appear as rectangles. The subunit MSG_SERVER is confined within the package body of MSG_MANAGER and is therefore represented as a parallelogram. The specification of MSG_SERVER contains each of procedures RECEIVE, PICKUP, SEND, SEND-BACK, GET , PUT and WAIT. They are therefore accessible within the confines of MSG_MANAGER but not external to it. The ASG also indicates the external connections to the subordinates FRAME_MANAGER and FRAME_BUFFER_POOL. These two external interfaces are defined in the code using the "with FRAME_MANAGER" and "with FRAME_BUFFER_POOL" declarations. They are interfaced through the tasks M_RX and M_TX which have explicit calls within their procedure bodies.

Within FRAME_MANAGER, both SEND and RECEIVE are declared within the specification section and therefore are externally accessible. The diagram shows this with the data couplings travelling to the procedures. Similarly, in FRAME_BUFFER_POOL the specification declares both FRAME_BUFFER and the internal components SEND_BACK and PICKUP, and therefore these too are externally accessible by the client.

The representation then of CADRE, derivations of Buhr, are indeed radically different to those for regular structure charts. The utility of this display method should be considered when designing the tools for automatic extraction from Mesa code. Does this representation show Mesa constructs better, or are the traditional structure chart representations better? These are

questions that will be answered in section 1.3 as we develop the theoretical concepts ot the tools themselves. CADRE clearly has very good graphical display capabilities in all areas of interest.

#### 1.2.1.5 Data Dictionaries

Data dictionaries (DD) have been implied throughout the discussions, although not discussed in any detail. The DD is simply 'an ordered set of definitions of terms used in the DFD' [# 19]. This section will develop the notations used within the DD structure.

At this point the concept of definitions can be expanded further at to illustrate DeMarco's discussions. DeMarco quotes Aristotle to define the word definition:

> "A definition is a description consisting of *genus* and *differentia*. The genus establishes some class that contains the word being defined, and the set of differentia distinguishes it from all other members of that class. ---- Adapted from Aristotle ----
>
> For Example: Man is the animal possessed of the capacity for articulate speech. Genus - animal, Differentia - possessed of the capacity for articulate speech". ---- Aristotle again ----

To comply with this viewpoint, DD's have to define these two pieces to make explicit DD entries. The DD entries will have a genus to describe a class of items and a differentia to define explicit members of that class. There are specific classes that are types of data that need to be considered. These classes refer to the type of data as seen on a DFD. Thus, all of the data elements of a DFD can be represented within these defined classes.

# SECTION 1 - INTRODUCTION AND BACKGROUND

- Data flow. A pipeline over which data of known composition is transmitted These tie directly to the DFD.
- File A time delayed repository of data. These correlate well to our treatment of External input and output data items discussed in section 1.2.1.3.
- Process A transformation of incoming data flows into outgoing data flows Again a direct correlation with the DFD
- Data element A data flow that cannot be decomposed into any subordinate data flow

The differentia is represented with a specific data item name. Each name must be unique to ensure that it is an explicit item and cannot be confused with any other. This applies at all levels of data abstraction. The data items within a Data Dictionary are best represented with a Tops Down approach, similar to our earlier discussions on structured design. The idea is to provide the greatest level of abstraction first. Data flow at the highest level may contain many subordinate data flow items which, in turn, are made up of primitive data elements. A smooth transition from the highest level of abstraction to the lowest is the key. Users of the DD can then choose the level of detail they need to address.

Data Dictionaries require several additional notations to display the information we need to hold. For example, working at the lower levels of data definition the data item 'character' is made up from the data elements 'a', 'b' or 'c' etc. Note that 'a', 'b','c' etc. are primitive data elements as they cannot be broken down any further. These data elements are the primitives of our definition process. Higher level data flows are a composite set of these primitives. How do we represent these choices? The example also brings in other interesting thoughts. How can a set be represented? The data item 'Character' may be the set of 26 alpha characters, or perhaps the 26 alpha characters + the numeric set, or some entirely different set. The set may have a single occurrence of a particular primitive element, or may be confined to one. For example, can

# SECTION 1 - INTRODUCTION AND BACKGROUND

'character' contains the string 'aaaaa' or is only one 'a' permissible. Are there choices either / or, or could an item be optional. The following conventions are offered by DeMarco to handle the expression of these different options:

| Symbol | Meaning | Example |
|---|---|---|
| = | IS EQUIVALENT TO | Flight-Manifest = {Passenger-Name}<br>This items shows that Flight-Manifest is equivalent to iterations of Passenger-Name |
| + | AND | Passenger-Name = First-Name + Second-Name + Middle initial<br>This item shows that Passenger-Name has three components all of which must be present. |
| [ ] | EITHER-OR | Passenger-Name = [First-Name + Second-Name + Middle-initial<br>I Initials].<br>This item shows that Passenger-Name can be made up of the three parts OR their initials. Note the symbol I which signifies the OR. |
| { } | ITERATIONS OF | As above<br>Flight-Manifest = {Passenger-Name}<br>This items shows that Flight-Manifest could have many iterations of Passenger-Name |
| ( ) | OPTIONAL | Passenger-Name = First-Name + Second Name +( Middle-initial)<br>This item shows the Middle Initial is an optional field and need not be present. |

**Figure 1.2.1.5.1 - Data Dictionary notation symbols**

We can use these definition conventions to display any type of data item in any class, starting from the highest level of abstraction and working downwards or vice-versa. Consider the following example starting at a high level of abstraction.

# SECTION 1 - INTRODUCTION AND BACKGROUND

| | | | |
|---|---|---|---|
| Invoice-body | = | {Invoice-line} | Multiple iterations of invoice-line |
| Invoice-line | = | Quantity + Item-number | Additive single occurrences of data items |
| | | + Unit-price + Item-total | |
| Quantity | = | Number + Number + Number | A three digit number |
| Number | = | [0 I1 I 2 I 3 I 4 I 5 I 6 I 7 I 8 I 9] | A single digit choice from the list |

Note the ascending level of detail. 'Number' finally defines the explicit data elements that make up Quantity. Quantity is one of the data items within Invoice line (Note: each data item needs to be defined to be complete, Item-number, Unit-price and Item-total are all left undefined for brevity.). Thus Invoice-line is a single occurrence of a data item in Invoice-body. The presentation this way is easily understandable and allows use of ascending/descending levels of abstraction of data or detailed treatment as desired.

There is one final representation that needs to be discussed to complete this section. The number of repetitions of a data element can be explicitly defined. Consider the data item 'Invoice-body'. This data item is made up of multiple iterations of Invoice-line. How many is the maximum, and how many is the minimum? As the definition stands above, {Invoice-line}, defines the maximum as infinity and the minimum as zero. If we wanted to specify the range more closely, then we can use a leading and trailing digit. For instance '1{Invoice-line}200' means that there always has to be at least 1 occurrence of Invoice-line and the maximum number of occurrences is 200.

That concludes the section on Data Dictionaries.

# SECTION 1 - INTRODUCTION AND BACKGROUND

## 1.2.2 Search Process

The document search for this thesis was done utilizing the Xerox library facilities at 800 Phillips Road, Webster, NY 14580. The thrust of the search was to investigate articles, books, conference proceeding and periodicals in two major subject areas. The facilities at Xerox provide for keyword item search, relevant title listing, and abstract printout. After the search, the items requested are either brought out from the archives if available, or requested from libraries holding the information. The details of the search are discussed below.

The search concentrated on the subject areas "Computer Aided Software Engineering" and "Reverse Engineering". The Reverse Engineering category was obviously the most focused on the subject of interest for this thesis. However, I wanted to have information regarding design methodologies that utilized existing automated techniques in order to understand current thinking in this regard. However, as can be seen in the following statistics, the CASE search showed a decreasing level of return after the first few categories. This is attributable to the fact that the objectives for this search were met early on, and consequently the acceptability criteria narrowed.

Each subject area provided a vast array of possibilities. Therefore, to create a manageable reading list, further methods of reduction were employed. All titles offered were printed and them scanned manually for applicability. Also the abstract from the selected items (where available) were printed and studied. The abstract list provided a further opportunity to focus the study, to reduce the volume of peripheral items. Articles, periodicals, proceedings and books were graded into priority categories 1,2 & 3. Category 1 documents were chosen for this concentrated study. Category 2 & 3 documents were filed for reference in case required for further study.

# SECTION 1 - INTRODUCTION AND BACKGROUND

As the study proceeded, other books and articles were found to be needed. In some cases, articles referenced books and other relevant articles which were also obtained for study. The statistics of the total process are provided below. Further details of the Priority 1 items are listed in the Bibliography in section 4.

Statistics of the search are shown below:

| Search area | Reverse Engineering | | CASE | |
|---|---|---|---|---|
| | Pri 1 | All | Pri 1 | All |
| IEEE proceedings 1988 - 1990 | 5 | 8 | 5 | 40 |
| Periodicals 1988 - 1990 | 2 | 3 | 1 | 20 |
| Computer journals | 11 | 131 | 1 | 743 |
| General library index | 3 | 54 | -- | -- |
| Compendex index 1990 | 4 | 15 | -- | -- |
| Further Articles & Books | 13 | 25 | 2 | 15 |
| Xerox Mesa books | | | 4 | - |
| **TOTALS** | **38** | **236** | **13** | **800** |

**Table 1.2.2.1 - Document search statistics**

# SECTION 1 - INTRODUCTION AND BACKGROUND

## 1.3 Theoretical and conceptual development

This section will deal with Mesa code and information extraction. We have discussed, so far, some of the theoretical basis for previous projects that have attempted Reverse Engineering. We will now move on to the specifics of Reverse Engineering of Mesa code. Each of the four subject areas; Structure Charts, Data Flow Diagrams, Data Dictionaries and Comment Dictionaries will be discussed. By the end of this section we should have a good understanding of what can be applied and how.

### 1.3.1 Structure Charts

**Mesa file types**

Section 1.2.1.2 discussed the traditional form of Structure Charts. We need to discuss the file types and structure of Mesa, one more time, to map the general theories into the specific extraction requirements. Mesa code has three types of files:

1 **Implementation or program files.** These files are the implementation modules for executable code. Procedures and straight line executable code are written in these modules.

2 **Interfaces or definitions files.** These files are predominantly for the control of EXPORT and IMPORT characteristics. Each implementation module usually has an associated interface to allow clients to gain access to the available procedures.

3 **Configuration files.** These files collect together, at higher levels of abstraction, a number of interfaces and implementations. Restrictions on IMPORTS and EXPORTS are defined at each configuration file.

# SECTION 1 - INTRODUCTION AND BACKGROUND

The following diagram illustrates the relationship between these types of files:

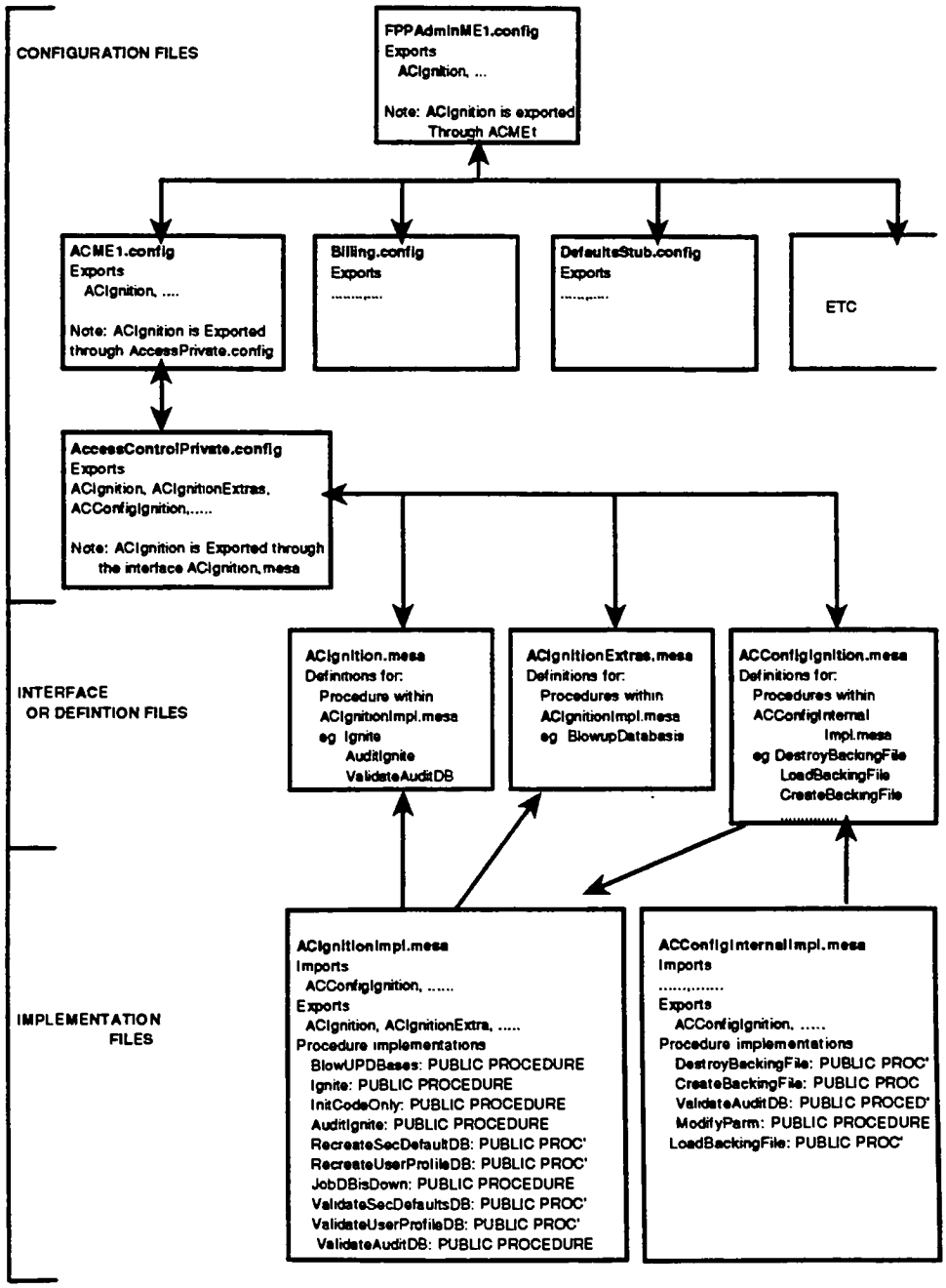



**Figure 1.3.1 - Mesa code implementation example**

# SECTION 1 - INTRODUCTION AND BACKGROUND

The diagram above, an example configuration taken from real live code, shows the characteristics of each of the files under discussion. The implementation files clearly hold the executable code. The diagram lists a number of examples of the procedures offered by each implementation module. The interfaces, or definitions files, are the method by which clients can access the implementations. The interface files list the available procedures. Note that in the example there are two interface files for ACIgnitionImpl.mesa. These are ACIgnition.mesa which is shown exporting Ignite, AuditIgnite and ValidateAuditDB; and ACIgnitionExtras which exports BlowUpDatabases. Any procedure that is to be exported beyond the local implementation module must be detailed in the appropriate interface. Also, procedures must be declared 'PUBLIC' to enable export. Clients access the procedures through that interface. The configuration files collect together groups of interfaces and implementation files to create a structured hierarchy.

At first blush, this looks a lot like a Structure Chart. In fact, the complete implementation of this system looks very much like a structure chart with some 20 top level modules and subordinate config files, with subordinate interfaces, with subordinate impl's, etc. This chart does indeed indicate something of the scope of effect. Note, in the diagram, that ACIgnition is exported right up to the top in FPPAdminME1.config. This means that the procedures declared in the ACIgnition interface are available beyond this level. You will probably have realized that this opens these procedures to an enormous number of potential clients. Any of the configs and interface files on the way up the tree (not all are shown either) can access ACIgnition [if Imported through a particular import declaration]. Thus, the clients for ACIgnitions procedures are not at all obvious at this level. The drawing, although useful for understanding the binding structure is not giving us what we need to construct Structure Charts

# SECTION 1 - INTRODUCTION AND BACKGROUND

**Developing the rules**

What then do we need to know to construct the SC? Well we must first understand the Client subordinate relationships. We must, therefore, understand which are the calling procedures and which are the called. Calls can come from within Impl's, from within either a straight line code section, or from a procedure. To do this, it will be necessary to interrogate all the code that has the potential to call a procedure and look for occurrences. For every call from a unique procedure or straight line code segment, an entry should be noted. We will note this as the first rule, Rule 1 in table 1.3.1 below. If multiple calls to the same procedure are made from within a parent procedure or segment of straight line code, then a separate SC node should be allocated for each call, Rule 2. We will need this later to create DFD's. This rule is similar to that of section 1.2.1.3 Table 1.2.1. The scope of the search can be limited to those areas where export access has been given through the interfaces and config files. It may however be easier to search all files for calls than to restrict the search. We will note this as Rule 3 in table 1.3.1 and evaluate its utility in the implementation phase.

The Structure Chart, then, has to consider all of the procedure calls throughout the system irrespective of the interfaces. The system is assumed to be in running order and, therefore, the strong type check of procedures and variable will have been done. It is, therefore, safe to assume that all calls will find their appropriate destination. It should be noted that Interface files and configuration files do not contain executable code and, therefore, have little bearing on the structure of the SC.

Mesa interface files allow external access to the procedures they export. As a result, implementation modules can access procedures from another implementation file through the interface that exports it. Each module wishing to use a procedure from another module, must

import that procedure. The IMPORT clause and USING statement defined in the header of the implementation module facilitate the usage of these external procedures. Any procedure within a module is given access by this method. Therefore, to represent these calls through foreign interfaces in the SC, the convention of allocating a subordinate node each time a procedure is called is adopted. The subordinate node is illustrated with the pattern Interfacename.procedurename. We will call this Rule 4.

Examples of this are not shown clearly in Figure 1.3.1. However, we can illustrate this by examining the code of the implementation module ACignitionImpl. This module has a number of import clauses, 18 to be exact. Each import clause has an associated directory entry with a USING statement which defines the exact procedure to be used from the interface being imported. For example, the interface ACconfigIgnition is imported into ACIgnition [see Figure 1.3.1]. There are three procedures that are made accessible to ACIgnition through the USING statement in the directory header of the module. The code writes ACConfigIgnition USING [CreateBackingFile, DestroyBackingFile, LoadBackingFile]. The point is that these procedures can now be used within ACIgnition. What should the structure chart look like? If we apply Rule 4, then the structure chart will look like Figure 1.3.2 below. This satisfies the rule, but is kind of counter intuitive to the representation in the configuration diagram Figure 1.3.1. Nevertheless, with regard to the calling relationships, the modules shown are subordinate to the Ignite procedure which is one of the procedures within ACIgnition implementation that uses the three imported procedures.

# SECTION 1 - INTRODUCTION AND BACKGROUND

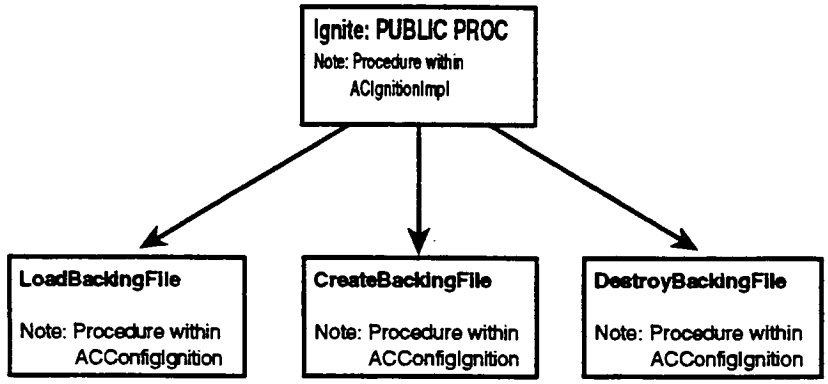



**Figure 1.3.2 - Subordinate relationship of imported procedures**

Let's now turn our attention to limiting the scope of the search. The previous discussion assumed that all modules had to be scanned to ensure that all calls were found. Only in this way could client relationships be confirmed. To implement this means that, every time an SC is to be generated, all the code must be interrogated. For a big system of 1 million lines of code or more, this is impractical. How then can we limit the scope without losing structure? Examining diagram 1.3.1 again, we see that potential clients have a path to access an implementation. At the top level, clients wishing to import ACIgnition must do so through FPPAdminME1. Thus we could insert a dummy client at this level to artificially limit the scope and prevent information overload. We could also limit our view by doing this at each lower level config file boundary. The lowest practical level in this case probably being the AccessControlprivate.config as this file filters the export of all ACIgnitionImpl's procedures. Note that any other client calls within this hierarchy should also be represented in addition to the dummy just discussed. Let's make this Rule 5.

How then are we to discover all the client / service relationships in order to construct a SC that is limited by a dummy configuration node? For example, if a dummy node was inserted for

# SECTION 1 - INTRODUCTION AND BACKGROUND

FPPAdminME1, all the client service relationships may not be present. The client may be outside of FPPAdminME1. First is to construct the SC in the normal way and add a dummy node at the configuration boundary for procedures exported to this level (Rule 5). Map all procedures below the configuration boundary with their subordinates. This should yield a complete listing of all the connected procedures. (The alternative clause of Rule 1). The terminal nodes can easily be identified at the base of the trace. Second, the straight line code in any impl should be parsed, and procedure calls for this mapped to the existing nodes from the procedure parse. The straight line code should be allocated an SC node (Rule 6). When all the branches are joined together, this should produce a complete picture. A third piece of information is the 'control'. Mesa has a method of starting an impl by naming it within the configuration file. It is not mandatory. This is a useful way to understand the starting point in the chart. This information should also be captured (Rule 7).The table below shows the collection of rules we have talked to above.

| Rule | Description |
|---|---|
| 1 | **Procedure calls.** Map all procedure calls throughout the complete software system to obtain the full client / service relationship tree. This includes all calls embedded in procedures and within straight line code.<br>OR alternatively map all procedures below the dummy client of Rule 5. |
| 2 | **Multiple calls.** If multiple calls to the same procedure are made from within a parent procedure or segment of straight line code then a separate Structure Chart node should be allocated for each call. |
| 3 | **Scope of search.** The scope of the search for clients can be limited to the modules that have export access to the impl's |
| 4 | **Imported interfaces.** A subordinate should be created for each imported procedure each time it is called from a unique client. |
| 5 | **Dummy client's.** A dummy client can be inserted at any configuration file boundary to limit the scope of the SC. |
| 6 | **Straight line code.** A portion of straight line code in an impl should be represented by a Structure Chart node. |
| 7 | **Control impl.** The control impl clause of the configuration file defines the starting point of code execution. |

**Table 1.3.1 - Rules for creating a standard Structure Chart Representation**

# SECTION 1 - INTRODUCTION AND BACKGROUND

### Data structure extraction

To conclude this section, we need to define the data structure that must be extracted from the Mesa code. What will we need to construct the initial SC? Note that, as previously discussed, data items will not be illustrated at this point. However, as the relationship between parameters and procedures is very strong, it would seem appropriate to include at least these at this stage. The table below illustrates the data to be extracted. The real live code example illustrated in Figure 1.3.1 is used to provide concrete naming constructs that map using the rules developed in Table 1.3.1. Also referring to section 1.1.2, the Design Representations section of the thesis the following fields should be collected:

| Identification | Type | Rule # | Subordinate config or impl file | subr # | # unique | Interface parameters | Returns |
|---|---|---|---|---|---|---|---|
| AccessControlPrivate.config | Config | Rule 5 | ACIgnition.mesa | 1 | 1 | Not shown | Not shown |
| Control: None (Rule 7) | | Rule 5 | ACIgnitionExtras.mesa | 2 | 2 | Not shown | Not shown |
| | | Rule 5 | ACConfigIgnition.mesa | 3 | 3 | Not shown | Not shown |
| ACIgnition.mesa | I/F | Rule 3 | Straight line code | 1 | 1 | None | None |
| | | Rule 3 | Ignite | 2 | 2 | Same as procedure | Same as procedure |
| | | Rule 3 | AuditIgnite | 3 | 3 | Same as procedure | Same as procedure |
| | | Rule 3 | ValidateAuditDB | 4 | 4 | Same as procedure | Same as procedure |
| | | Rule 3 | RecreateSecDefaultsDB | 5 | 5 | Same as procedure | Same as procedure |
| | | Rule 3 | RecreateUserProfileDB | 6 | 6 | Same as procedure | Same as procedure |
| | | Rule 3 | JobDBIsDown | 7 | 7 | Same as procedure | Same as procedure |
| | | Rule 3 | ValidateSecDefaultsDB | 8 | 8 | Same as procedure | Same as procedure |
| | | Rule 3 | ValidateUserProfileDB | 9 | 9 | Same as procedure | Same as procedure |
| | | Rule 3 | ValidateAuditDB | 10 | 10 | Same as procedure | Same as procedure |

**Table continued on next page:**

# SECTION 1 - INTRODUCTION AND BACKGROUND

| Identification | Type | Rule # | Subordinate config or impl file | subr # | # unique | Interface parameters | Returns |
|---|---|---|---|---|---|---|---|
| ACIgnitionExtras | I/F | Rule 3 | BlowUpDataBasis | 1 | 1 | None | None |
| ACConfigIgnition | I/F | Rule 3 | DestroyBackingFile | 1 | 1 | None | status: ACConfigTypes. StatusCode |
| | | Rule 3 | LoadBackingFile | 2 | 2 | None | status: ACConfigTypes. StatusCode |
| | | Rule 3 | CreateBackingFile | 3 | 3 | | status: ACConfigTypes. StatusCode |
| ACIgnitionImpl.m | Impl | Rule 6 | Straight line code (SLC) | 1 | 1 | None | None |
| | | Rule 1 | Ignite | 2 | 2 | CausedRollover: BOOLEAN <- FALSE, bucketid: CARDINAL <- 0, logInCSR:BOOLEAN <- FALSE | ready: BOOLEAN |
| | | Rule 1 | Blowupdatabasis | 3 | 3 | None | None |
| | | Rule 1 | InitCodeOnly | 4 | 4 | None | None |
| | | Rule 1 | AuditIgnite | 5 | 5 | None | ready: BOOLEAN <- TRUE |
| | | Rule 1 | RecreateSecDefaultDB | 6 | 6 | None | ready: BOOLEAN <- FALSE |
| | | Rule 1 | RecreateUserProfileDB | 7 | 7 | None | ready: BOOLEAN <- FALSE |
| | | Rule 1 | JobDBisDown | 8 | 8 | bucketid: CARDINAL <- 0, clear: BOOLEAN <- FALSE, | ready: BOOLEAN <- FALSE |
| | | Rule 1 | ValidateSecDefaultsDB | 9 | 9 | none | ok: BOOLEAN <- TRUE |
| | | Rule 1 | ValidateUserProfileDB | 10 | 10 | None | ok: BOOLEAN <- TRUE |
| | | Rule 1 | ValidateAuditDB | 11 | 11 | None | ok: BOOLEAN <- TRUE |
| ACIgnitionImpl. SLC | None | Rule 6 | None | 0 | 0 | None | None |
| Ignite | Public Proc | Rule 1,4 | ACFaultStateInternal. SSCDeclareFaultAndState | 1 | 1 | [Fault:boundsU,state:notReady. uniqueID:FaultLogger.nullLogicID]; | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. SSCDeclareFaultAndState | 2 | 1 | [Fault:boundsU, state:notReady. uniqueID:FaultLogger.nullLogicID]; | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. SSCDeclareFaultAndState | 3 | 1 | [Fault:boundsU, state:notReady. uniqueID:FaultLogger.nullLogicID]; | ? |
| | | Rule 1, 4 | ACConfigIgnition. LoadBackingFile | 4 | 2 | None | ? |
| | | Rule 1, 4 | ACConfigIgnition. CreateBackingFile | 5 | 3 | None | ? |
| | | Rule 1, 4 | ACFaultStateInternal. DeclareFaultAndState | 6 | 4 | [Fault:goneD, state:notReady. uniqueID:FaultLogger.nullLogicID]; | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. SSCdeclareFaultAndState | 7 | 4 | [Fault:inconsistentD, state:notReady. uniqueID:FaultLogger.nullLogicID]; | ? |
| | | Rule 1, 4 | ACConfigQuery. GetACDefaultsVersion | 8 | 5 | None | ? |

**Table continued on next page:**

# SECTION 1 - INTRODUCTION AND BACKGROUND

| Identification | Type | Rule # | Subordinate config or impl file | subr # | # unique | Interface parameters | Returns |
|---|---|---|---|---|---|---|---|
| | | Rule 1, 2, 4 | ACFaultStateInternal. DeclareFaultAndState | 9 | 5 | fault:badD, state:notReady, uniqueID:FaultLogger.nullLogID | ? |
| | | Rule 1, 2, 4 | ACConfigQuery. GetACDefaultsVersion | 10 | 5 | None | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. DeclareFaultAndState | 11 | 5 | fault:badversionD, state:notReady, uniqueID:FaultLogger.nullLogID | ? |
| | | Rule 1, 4 | UPIgnition.LOADUPDBase | 12 | 6 | None | ? |
| | | Rule 1, 4 | UPBackup.GetUPVersion | 13 | 7 | None | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. DeclareFaultAndState | 14 | 7 | fault:badU, state:notReady, uniqueID:FaultLogger.nullLogID | ? |
| | | Rule 1, 2, 4 | UPBackup.GetUPVersion | 15 | 7 | None | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. DeclareFaultAndState | 16 | 7 | fault:badVersionU, state:notReady, uniqueID:FaultLogger.nullLogID | ? |
| | | Rule 1, 4 | SessionBegEndExtras. ResetCSR | 17 | 8 | None | ? |
| | | Rule 1, 4 | ACFaultStateInternal. DeclareFault | 18 | 9 | fault:validateD, uniqueID:FaultLogger.nullLogID | ? |
| | | Rule 1, 2, 4 | ACFaultStateInternal. DeclareFault | 19 | 9 | fault:validateU, uniqueID:FaultLogger.nullLogID | ? |
| BlowUpDBases | Public Proc | Rule 1, 4 | InitCodeOnly | 1 | 1 | None | None |
| | | Rule 1, 4 | ACConfigIgnition. DestroyBackingFile | 1 | 1 | None | ? |
| | | Rule 1, 4 | UserProfileMgr. DestroyUPDBase | 2 | 2 | !UserProfileMgr.Error => CONTINUE | ? |
| InitCodeOnly | Public Proc | Rule 1, 4 | UPMgrLock. InitDBaseMonitor | 3 | 3 | None | ? |
| etc ..... | etc ..... | etc ..... | etc ..... | | | etc ..... | etc ..... |
| etc ..... | etc ..... | etc ..... | etc ..... | | | etc ..... | etc ..... |

**Table 1.3.2 - Data fields required for Structure Chart creation**

(Note: The question marks in the data field indicate that these values are not known at this time. Only when the impl relating to those procedures are interrogated will the data values be known.)

# SECTION 1 - INTRODUCTION AND BACKGROUND

### 1.3.2 Data Flow Diagrams

Section 1.2.1.2 discussed the traditional forms of Data Flow Diagrams and section 1.2.1.3 discusses the methods for developing DFD's from structure charts in Pascal code. In this section we will use these concepts and apply them to Mesa. The example Mesa code segment discussed in section 1.3.1 will be used as the template.

**DFD primitives**

To create the Data Flow Diagram from the code, we need to collect the information that will yield the appropriate details. The information required will be the transformations, the data flows, and the repositories. Table 1.2.1 gives the details of the rules that need to be applied to produce this data. We will examine our sample piece of Mesa code and see whether the information can be extracted appropriately.

The first observation of our piece of Mesa code is that external devices and data stores are not immediately obvious. The Pascal example of section 1.2.1.3 has these declared in the Main Program, plus the instructions read and write are used when accessing them. I do not see any similar constructs in Mesa for describing IO operations. Therefore, it will be necessary in the implementation of the Mesa extractors to add another manual processing step, see step 3 below. The sequence will be as follows:

| | | |
|---|---|---|
| 1 | Extract & map nodes for the Structure Charts | Automatic |
| 2 | Extract all the variables and categorize them | Automatic |
| 3 | Designate data items considered External | Manual |
| 4 | Create Graphic representation of DFD's | Manual |

# SECTION 1 - INTRODUCTION AND BACKGROUND

| Transformation nodes | | | Data | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Data description | | Data Type Information | | | Usage | | | | | | Scope (Note 4) | | |
| Procedure | Called procedure | Term Inel | Data Item identifier | Interface imported from ..... | Data values used .... | Type imported from ..... | Type | Int / Ext | Read | Write | Proc perem | Return perem | Eneble clause | Local | Globel | Exter nel |
| Ignite | Straight Line Code | No | CausedRollover | | FALSE | | BOOLEAN | Int | Read | | X | | | X | | |
| | | | bucketID | | 0 | | CARDINAL | Int | Read | | X | | | X | | |
| | | | logInCSR | | FALSE | | BOOLEAN | Int | Read | | X | | | X | | |
| | | | Ready | | TRUE | | BOOLEAN | Int | | Write | | X | | X | | |
| | | | BoundsFault | Runtime | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | AddressFault | RTOSPageFault | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | WriteFault | RTOSPageFault | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | props | | NIL | UserProfile | SecAdmin UserProfile | Int | | Write | | | | X | | |
| | | | auditNeedsValidate | | TRUE, FALSE | | BOOLEAN | Int | | Write | | | | | X | |
| | | | [] | | NON | | DUMMY | Int | | Write | | | | X | | |
| | DeclareFault AndState from ACFaultStateInternal | No | fault | | boundsU, goneD, inconsitentD, badD, badVersionD, badU, badVersionU | ACFaultStateInternal | Faults | Int | Read | | X | | | X | | |
| | | | state | | notReady | ACFaultStateInternal > EventHandler | State | Int | Read | | X | | | X | | |
| | | | uniqueID | FaultLogger | Value of: nullLogicID | FaultLogger | ID | Ext | | Write | X | | | | | X |
| | | | ReturnDFSA (Note 2) | | A | | CHAR | Int | | Write | | X | | X | | |
| | | | Dummy1 (Note 2) | | B | | CHAR | Int | | Write | | | | | X | |
| | SSCDeclare FaultAndState. from ACFaultStateInternal | No | fault | | boundsU | ACFaultStateInternal | Faults | Int | Read | | X | | | | | X |
| | | | state | | notReady | ACFaultStateInternal > EventHandler | State | Int | Read | | X | | | | | X |
| | | | uniqueID | FaultLogger | Value of: nullLogicID | FaultLogger | ID | InExt | | Write | X | | | X | | |
| | | | RetSSCDFSA (Note 2) | | C | | CHAR | Int | | Write | | X | | X | | |
| | | | Dummy1 (Note 2) | | D | | CHAR | Int | Read | | | | | | X | |
| Column 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

**Table 1.3.4 - DFD extraction IGNITE data descriptor table**

# SECTION 1 - INTRODUCTION AND BACKGROUND

| Transformation nodes | | | Data | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Data description | | Data Type information | | | Usage | | | | | | Scope (Note 4) | | |
| Procedure | Called procedure | Terminal | Data item identifier | Interface imported from ..... | Data values used .... | Type imported from ..... | Type | Int / Ext | Read | Write | Proc param | Return param | Enable clause | Local | Global | External |
| AuditIgnite | Straight line code | No | ready | | TRUE | | BOOLEAN | Int | | Write | | X | | X | | |
| | | | BoundsFault | Runtime | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | AddressFault | RTOSPageFault | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | WriteFault | RTOSPageFault | | | SIGNAL | Int | Read | | | | X | | | X |
| | | | status | | success | ACConfigTypes | StatusCode | Int | | Write | | | | X | | |
| | | | fileID | | ??? | RTOSFile | FileID | Int | | Write | | | | X | | |
| | | | auditNeedsValidate | | FALSE | FALSE | BOOLEAN | Int | Read | | | | | | X | |
| | | | filesizeInPages | | 480 | | LONG CARDINAL | Int | | Write | | | | X | | |
| | | | fileID | | | RTOSFile | FileID | Int | Read | Write | | | | X | | |
| | | | auditFileName | | "AuditTrail.db"L | | LONG STRING | Int | | Write | | | | X | | |
| | | | status | | success doesNotExist directoryError | ACConfigTypes | StatusCode | Int | Read | Write | | | | X | | |
| | | | created | | FALSE, TRUE | | BOOLEAN | Int | Read | Write | | | | X | | |
| | DeclareFault AndState from ACFaultStateInternal | No | fault | | boundsU, goneD, inconsIstentD, badD, badVersionD, badU, badVersIonU | ACFaultStateInternal | Faults | Int | Read | | X | | | X | | |
| | | | state | | notReady | ACFaultStateInternal > EventHandler | State | Int | Read | | X | | | X | | |
| | | | uniqueID | FaultLogger | Value of: nullLogicID | FaultLogger | ID | Ext | | Write | X | | | | | X |
| | | | ReturnDFSA (Note 2) | | A | | CHAR | Int | | Write | | X | | X | | |
| | | | Dummy1 (Note 2) | | B | | CHAR | Int | | Write | | | | | X | |
| | | | etc ..... | | | | | | | | | | | | | |
| Column 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

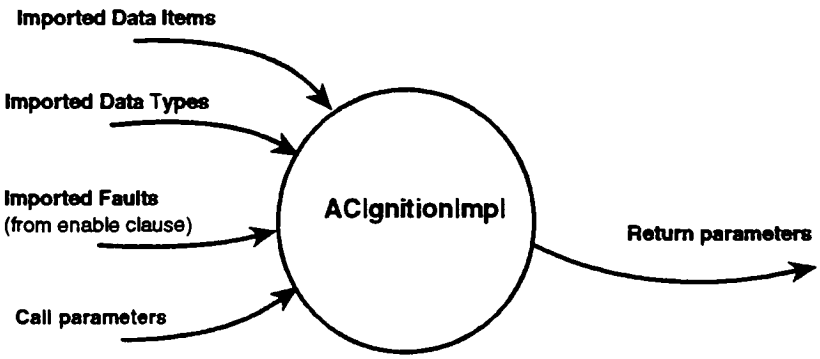
**Table 1.3.5 - DFD Extraction AUDITIGNITE data descriptor table**

# SECTION 1 - INTRODUCTION AND BACKGROUND

**Notes for Table 1.3.4 and 1.3.5**

1 Remember that internal data items are those shared between modules and external data items are those representing major data repositories, such as date files or IO.
2 The dummy data items are added for illustration purposes. The real data items, not studied, can be found by further examination of the procedure source code.
3 Not all procedure calls within Ignite and AuditIgnite are included in this table for simplicity.
4 Local means within a procedure, global means within an impl and external means the data item is accessible externally to the interface

The tables above illustrate the fields required to develop the DFD's. Step 3 above has been applied to the data items that look to be relevant in column 9. The data item 'nullLogicID' appears to be the only item that could be properly designated external. For the purposes of illustration we will assume this to be a valid assumption, Three levels of DFD are facilitated by the information in table 1.3.4. [See Figures 1.3.7, 1.3.8 and 1.3.9]. First a slightly artificial one using the impl module as the top level node [Figure 1.3.7]. The data inputs and outputs are shown with ACIgnitionImpl as the context node. The diagram is constructed with reference to each of the columns of Tables 1.3.4 and 1.3.5.




**Key:** Imported Data Items - See column 5 of Tables 1.3.4 & 1.3.5
ImportedData Types - See column 7 of Tables 1.3.4 & 1.3.5
Imported Faults See column 14 of Tables 1.3.4 & 1.3.5
Call parameters See column 12 of Tables 1.3.4 & 1.3.5
Return parameters - See column 13 of Tables 1.3.4 and 1.3.5

**Figure 1.3.7 - Highest level DFD for ACIgnitionImpl**

# SECTION 1 - INTRODUCTION AND BACKGROUND

The second level of DFD illustrated in figure 1.3.8 goes down one more level of detail. With the data we have available at this stage the representation becomes considerably more explicit. The figure shows the level of the main procedures within ACConfigImpl. We have only chosen to illustrate Ignite and AuditIgnite as the data for these two are used in our tables. Considerably more detail would be forthcoming if all procedures within the impl were illustrated.

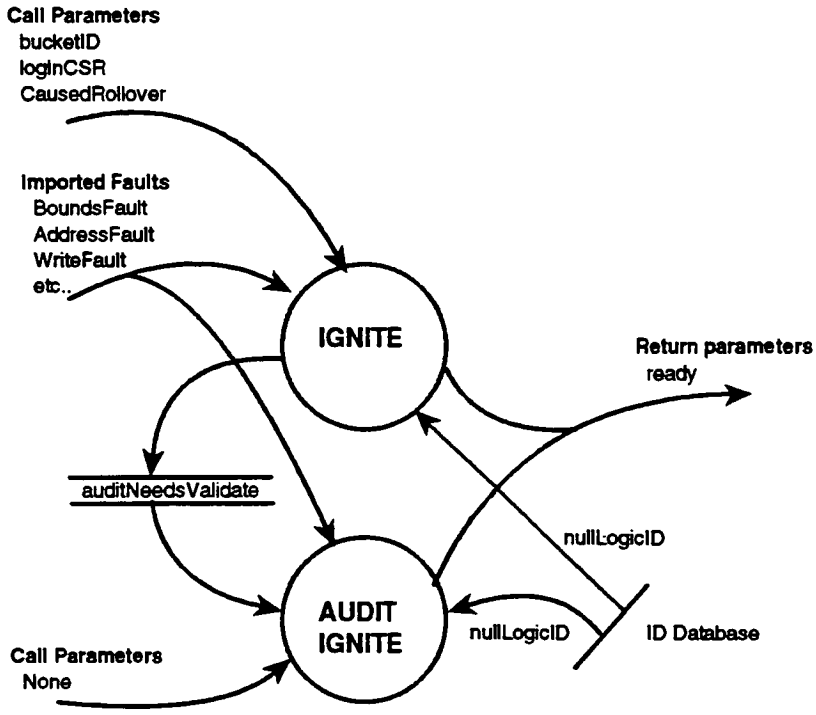



**Figure 1.3.8 - Main procedure level DFD**

This diagram shows the inputs and outputs of Ignite and AuditIgnite. These are the two main transformation nodes in column 1 of the data extraction tables [1.3.4 &1.3.5]. The input parameters to Ignite are obtained from column 12, AuditIgnite has no input parameters. The

# SECTION 1 - INTRODUCTION AND BACKGROUND

return value for both transformations is 'ready' as shown in column 13. From column 4 and 7, we can see that auditNeedsIgnite is passed between the two nodes. AuditNeedsIgnite is designated global in column16 because its scope of effect is within the impl, outside of the main procedures, but not external to the impl. The fault handling indicated by column 9 is common to both Audit and AuditIgnite. The ENABLE keyword identifies the fault handling area of the source code.The data item 'nullLogicID' is shared by both procedures. The value is externally imported.

The third level DFD shows the lowest level of detail that can be illustrated with our current level of information. This diagram represents nodes and data items down to the procedure call level within Ignite.

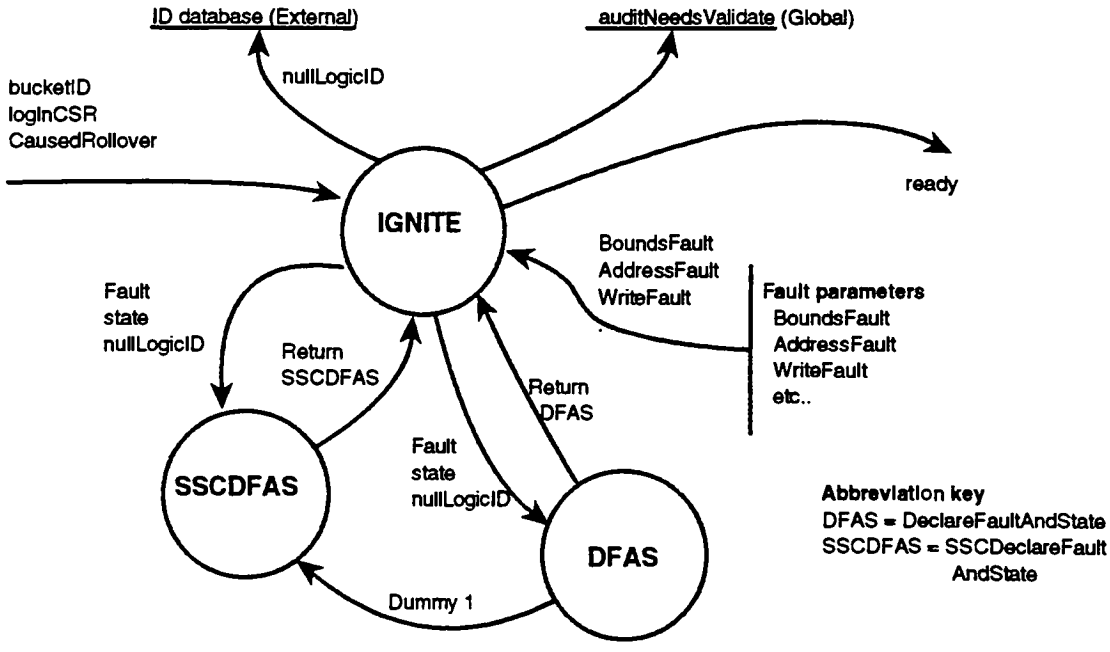



**Figure 1.3.9 - Lowest level DFD for the Ignite procedure**

Notes: 1 Many procedure calls have been omitted for simplicity.

This diagram puts together the rest of the information from Table 1.3.4. and 1.3.5. All the data items in the table are listed above with their relationships to appropriate transformations and repositories. The 'dummy 1' data item is added to illustrate that data items can be seen within transformation nodes that link them, similar to auditNeedsValidate in the level above. In the real situation DFAS and SSCDFA will have their own data items and calls. When the full parse of the code is done, all this information will become available for mapping.

All the rules identified in table 1.2.1 of section 1.2.1.3 have been applied, at least partially, in this analysis. With one exception, Rule 5.2. This rule has not been applied as data items involved with multiple calls, constants, expressions and global calls were not involved in the complications of the example. Of course, as soon as a larger sample of code is automatically parsed for the information the application, of these rules will quickly become necessary. These rules were applied in the thesis implementation phase.

### 1.3.3 Data Dictionaries

Section 1.2.1.5 details the theory behind Data Dictionary representation. Using this model and examining table 1.3.4 and 1.3.5, we can construct a sample representation of what would be expected in the Data Dictionary for the ACconfig impl.

The table below shows the examples using Fault, CausedRollover and bucketID. These are simply 3 types of data items for illustration purposes. Note that a further data item called 'digit' has been added for ease of readability. Digit is not a value supported in the implementation of

# SECTION 1 - INTRODUCTION AND BACKGROUND

ACConfig. The table could be re-written with the value of digit inside of the definition of bucketID. Take a look at the table that follows to see what I mean.

| | | | |
|---|---|---|---|
| Fault | = | [boundsU I goneD I inconsistentD I badD I badVersionD I badU I badVersionU] | An example of a data item with a fixed set of values declared in the ACFaultStateInternal interface file |
| CausedRollover | = | [TRUE I FALSE] | A standard boolean |
| bucketID | = | 1{Digit}N | A Cardinal with N bits. N represents the word size available on the machine |
| Digit | = | [0 I 1 I 2 I 3 I 4 I 5 I 6 I 7 I 8 I 9] | An added data item to represent digit. |

**Table 1.3.6 - Data Dictionary example from ACConfig**

It is not necessary to provide extensive examples as the method can be understood by way of the brief illustrations given. This concludes the Data Dictionary section.

### 1.3.4 Comment Information.

Commentary information is available in many forms. Examining the code, it appears that comments are most prevalent in the interface and in the header of implementation files. It is intended to look for methods of extracting those comments as backup information for the DFD's and Structure Charts. Unfortunately not all programmers have followed the same format and this will likely present some problems. Automatic extraction tools were not constructed. However, a cursory look at the code was done to see if these rules can be better defined. The conclusions are added to the recommendations section of the thesis.

# SECTION 2 - PROJECT DESCRIPTION

## 2.0 PROJECT DESCRIPTION

In this section, the exact details of the software implementation attempted for this thesis is described. The intent is to be brief. Extensive explanations of the theoretical details known before implementation start have been developed liberally in the previous sections. Whenever possible the earlier work will be referenced to avoid duplication. New findings will be highlighted as appropriate either in the text or through the Limitations [Limit] or Lessons Learned [Lessons] sections. The implemented program is called "Design Extractor".

This section is split into three main parts, the Functional Specification, the System Specification and the Implementation plan details. The Functional Specification section details what the expectations are of the software tools, the System Specification shows the architectural structure of the program and the Implementation plan details the progress steps along the path to completion.

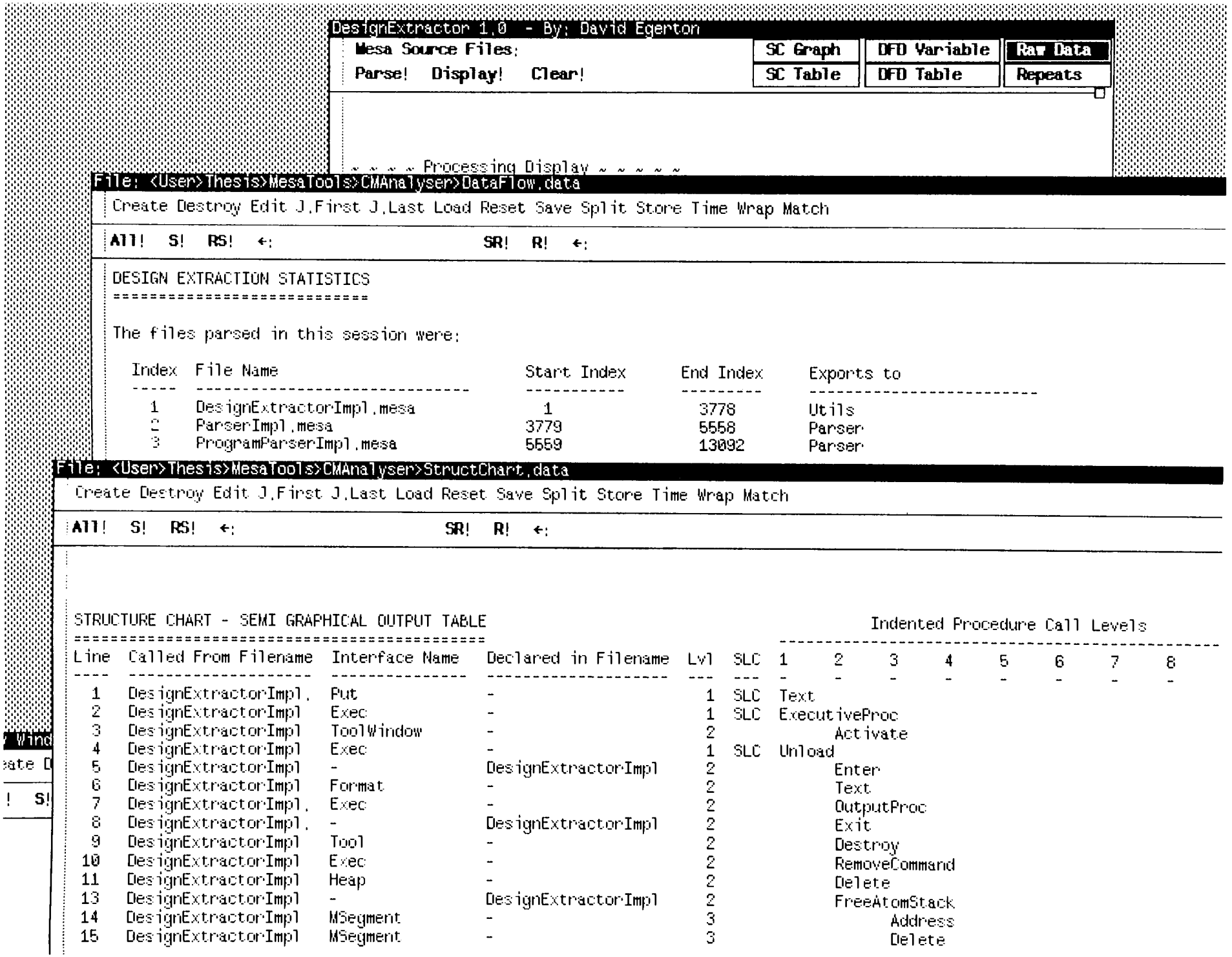
### 2.1 Functional specifications

The software that was developed for this thesis was constructed in the Mesa language. This language is a proprietary language developed by Xerox as stated in Section 1.1.3. The software is intended to be capable of parsing a section of code. Test files were taken from various places to illustrate functionality. However the utility is designed to be generically suitable for parsing any code that has been developed in Mesa. The input requirements are specified in Section 2.1.2.

The Design Extractor program produces outputs in two forms. Firstly as a file that can be displayed on the workstation user interface and secondly a printed output. The diagram on the next page illustrates the Design Extractor tool window opened with files parsed, plus the three

# SECTION 2 - PROJECT DESCRIPTION

output windows on the same screen. The output windows are linked to stored files, Extractor.data, StructChart.data and DataFlow.data. Samples of the printed output can be seen in Section 2.1.3.


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files:
Parse! Display! Clear!
SC Graph
DFD Variable
Raw Data
SC Table
DFD Table
Repeats
~ ~ ~ ~ Processing Display ~ ~ ~ ~ ~
File: <User>Thesis>MesaTools>CMAnalyser>DataFlow.data
Create Destroy Edit J,First J,Last Load Reset Save Split Store Time Wrap Match
All! S! RS! ←: SR! R! ←:
DESIGN EXTRACTION STATISTICS
The files parsed in this session were:
Index File Name Start Index End Index Exports to
1 DesignExtractorImpl.mesa 1 3778 Utils
2 ParserImpl.mesa 3779 5558 Parser
3 ProgramParserImpl.mesa 5559 13092 Parser
File: <User>Thesis>MesaTools>CMAnalyser>StructChart.data
Create Destroy Edit J,First J,Last Load Reset Save Split Store Time Wrap Match
All! S! RS! ←: SR! R! ←:
STRUCTURE CHART - SEMI GRAPHICAL OUTPUT TABLE
Indented Procedure Call Levels
Line Called From Filename Interface Name Declared in Filename Lvl SLC 1 2 3 4 5 6 7 8
1 DesignExtractorImpl. Put - 1 SLC Text
2 DesignExtractorImpl Exec - 1 SLC ExecutiveProc
3 DesignExtractorImpl ToolWindow - 2 Activate
4 DesignExtractorImpl Exec - 1 SLC Unload
5 DesignExtractorImpl - DesignExtractorImpl 2 Enter
6 DesignExtractorImpl Format - 2 Text
7 DesignExtractorImpl. Exec - 2 OutputProc
8 DesignExtractorImpl. - DesignExtractorImpl 2 Exit
9 DesignExtractorImpl Tool - 2 Destroy
10 DesignExtractorImpl Exec - 2 RemoveCommand
11 DesignExtractorImpl Heap - 2 Delete
13 DesignExtractorImpl - DesignExtractorImpl 2 FreeAtomStack
14 DesignExtractorImpl MSegment - 3 Address
15 DesignExtractorImpl MSegment - 3 Delete

# SECTION 2 - PROJECT DESCRIPTION

## 2.1.1 User Interface

This section develops in more detail the functions performed by the DesignExtractor tool from the perspective of the user interface. The DesignExtractor tool uses a standard XDE interface window modified to perform the functions required. Examples of the window are shown on the next few pages. The window is structured into three main area, the herald, the control window and the feedback window.

**2.1.1.1 The herald window:** This portion is the black stripe at the top, this contains the name of the program, the current release level and the author. The herald remains constant throughout operational sessions.

**2.1.1.2 The feedback window:** This portion of the window shows the feedback information provided by the program. This window will indicate the progress of processing in any particular session. The examples on the following pages show some of the feedback information provided.

- Parse of 1 file example
- Display processed example
- Conflict information example
- Raw Data selection example
- Multiple file Parse example

**2.1.1.3 The control window:** This portion of the tool window allows users to make selections. The operation of each selection is detailed below:

**Mesa Source Files** - at the top left, this is the entry point for files to be parsed. Click the mouse just to the left and type in the files to be parsed. The .mesa extension is not necessary. Only .mesa files can be processed. Add as many file names as you wish [Limit #3] separated by a space.

# SECTION 2 - PROJECT DESCRIPTION

**Parse!** - This selection starts of a parsing session for the file names already inputted into the " Mesa Source Files" space. After completion of a Parse session, further files can be input and another session started. The new files are added to the end of the database. The feedback window tracks each file as it is parsed and provides a summary of the current database size together with the file count.

**Display!** - This selection starts off the next phase in the processing cycle. Parse should be completed first before selecting the Display option. Display can be selected as many times as you wish. It creates a display (or really fills in the files Extractor.data, StuctChart.data and DataFlow.data if selected) of the data currently in the database.

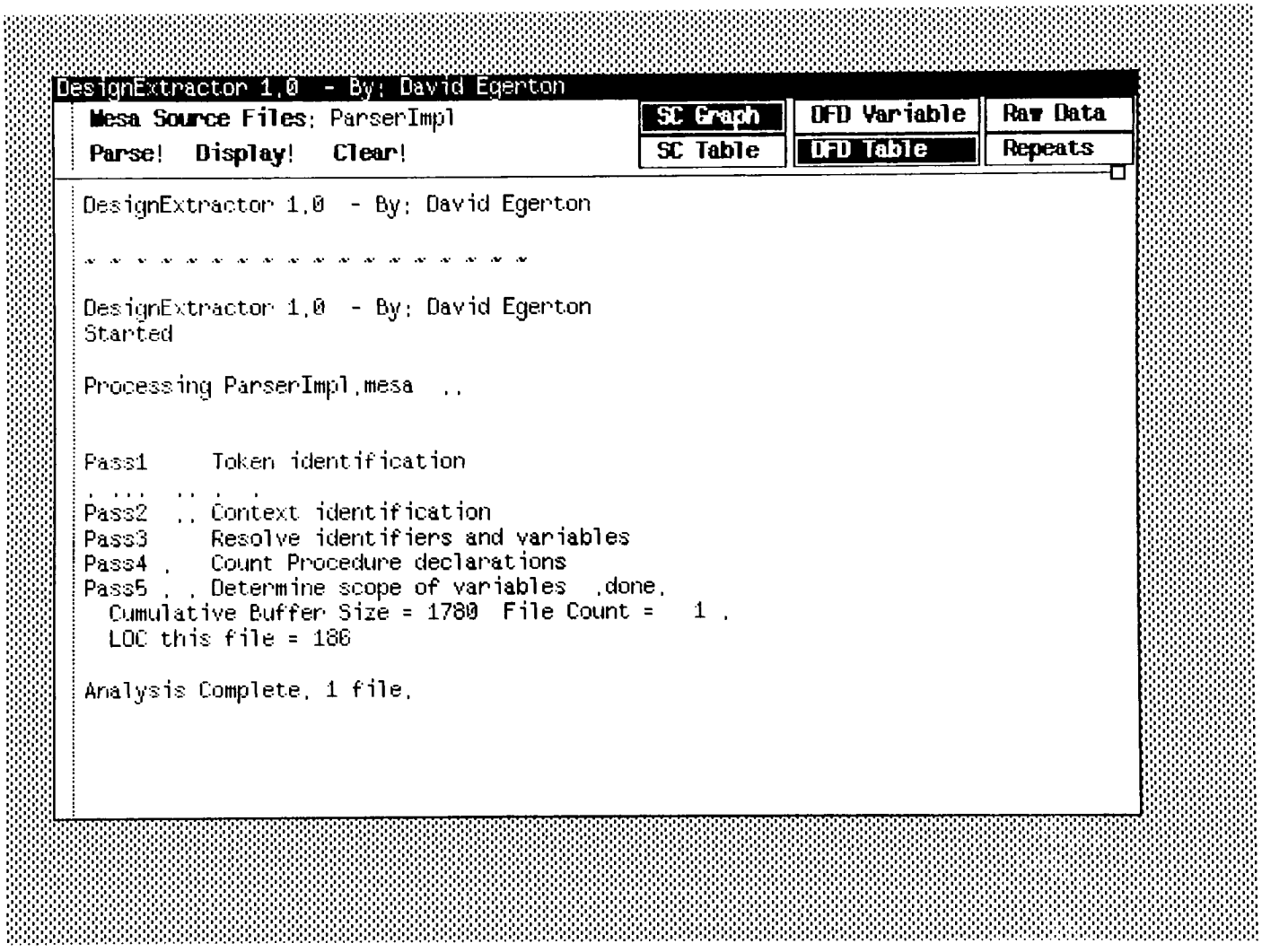
**Clear!** - This selection can be made if it is desired to delete the contents of the database. Use this selection if you want to start with a new root file.

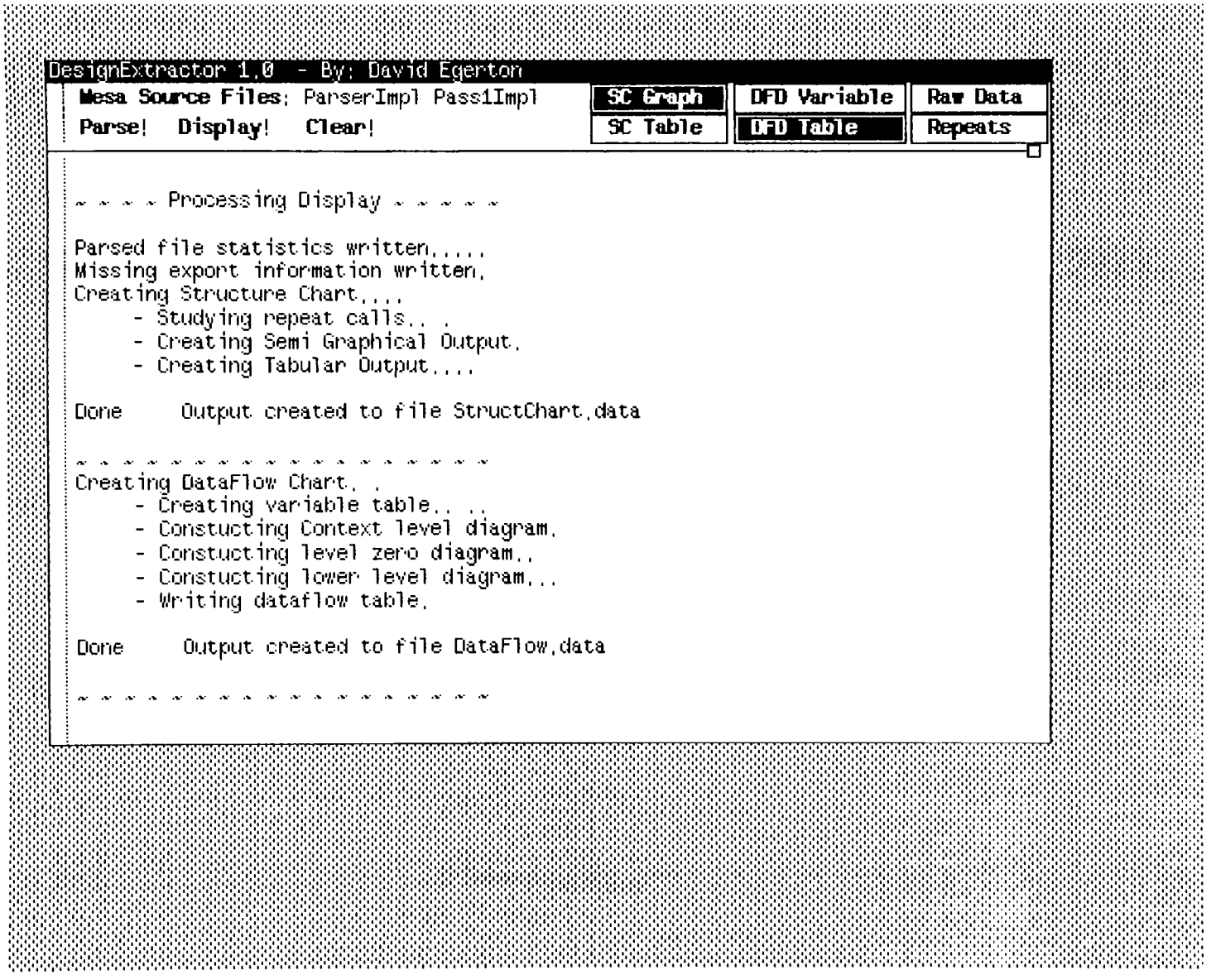
**The six buttons** - The buttons (SCGraph, SCTable, DFDVariable, DFDTable, RawData and Repeats) are selections that can be made prior to starting a Display session. Whichever buttons are highlighted in black (mouse switchable) will be output during the session. The buttons have no effect on a Parse session. The contents of the various output options achieved will be described in detail in the following sections.

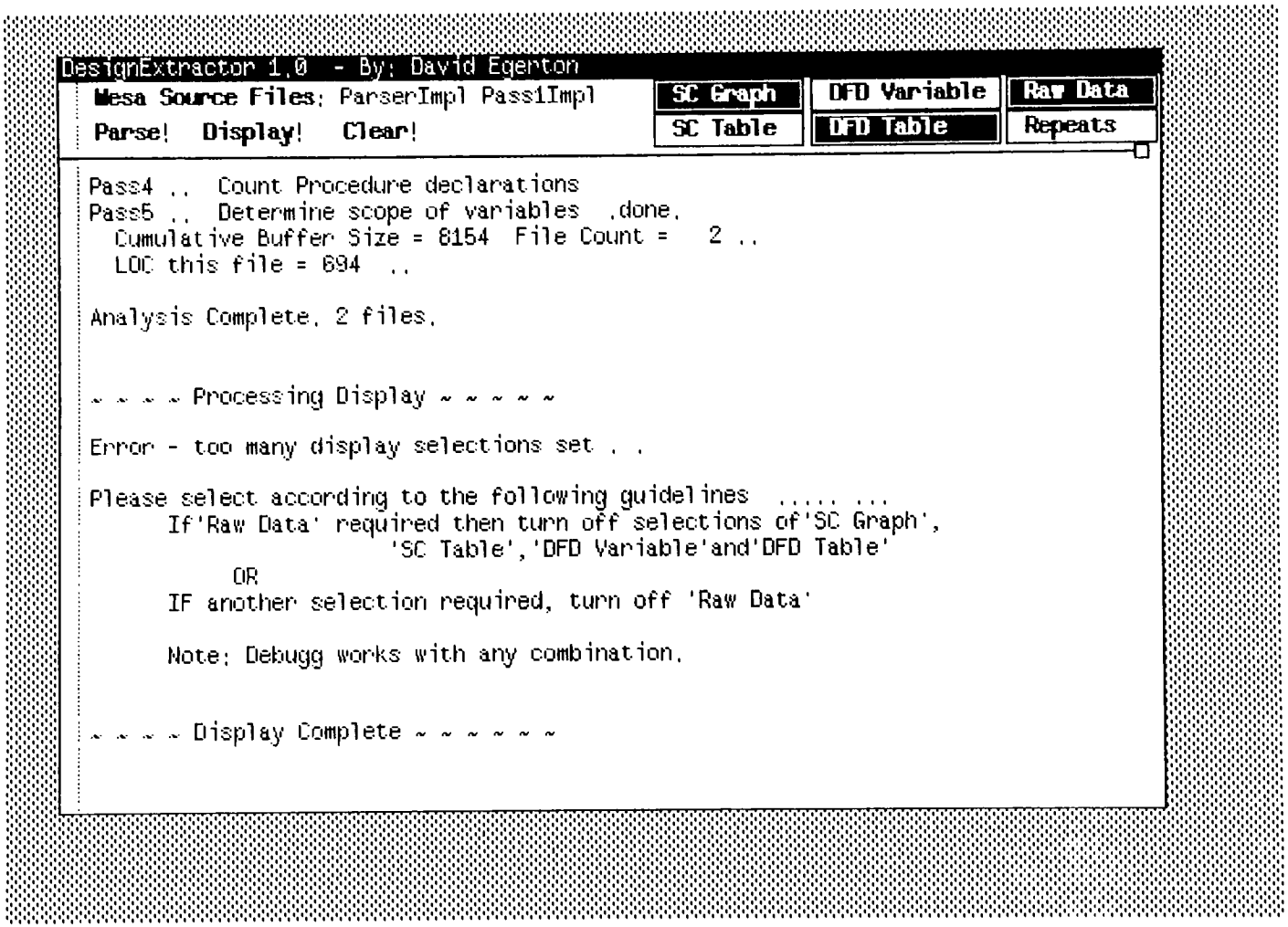
The Repeats button, if selected, allows the program to output all occurrences of an item. It was found [Lessons #3] that output representation was very ugly in some cases with all occurrences detailed. Therefore the default is to suppress repeats at various levels. Further information on repeats is described in Appendix F.

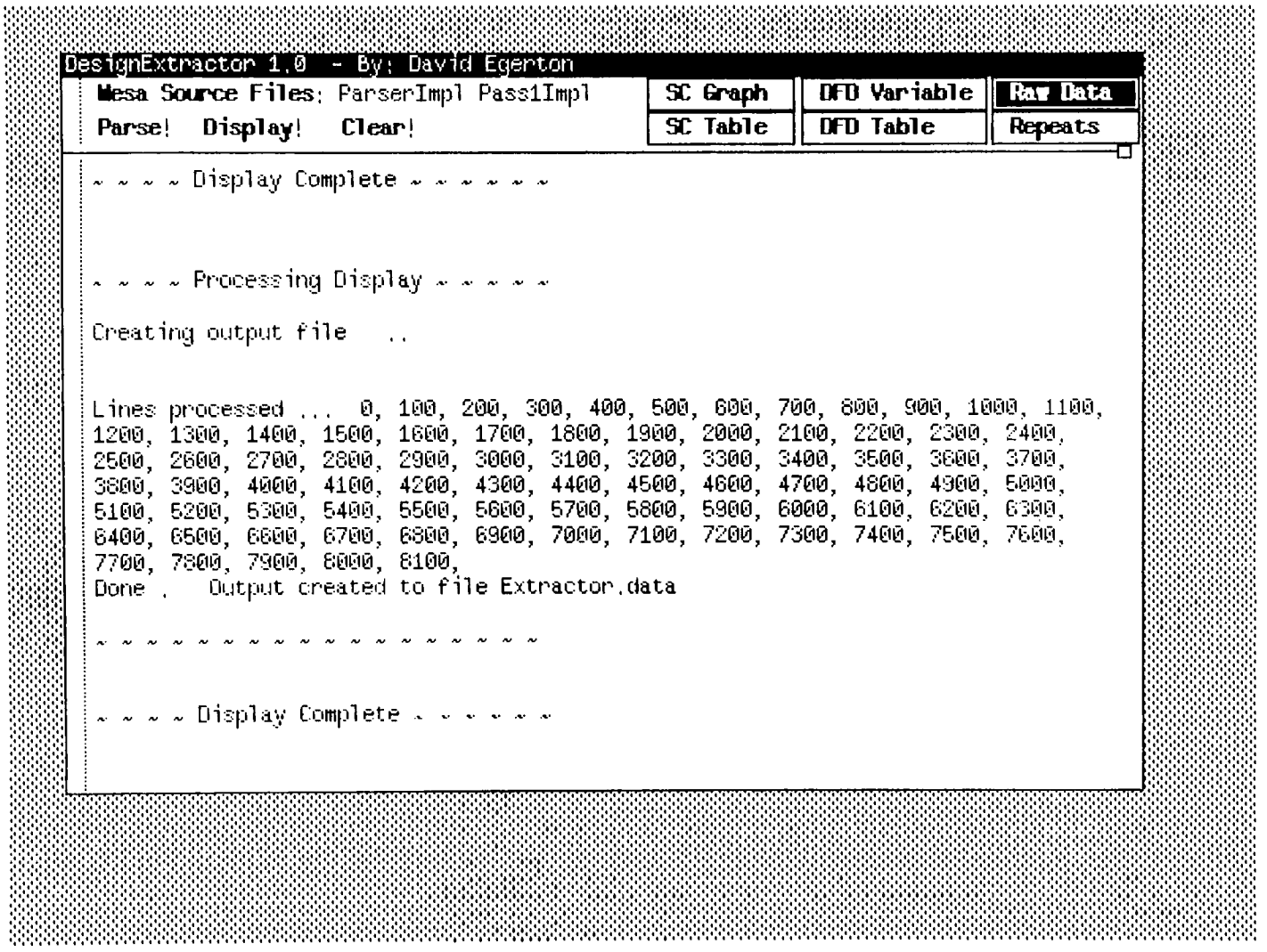
# SECTION 2 - PROJECT DESCRIPTION

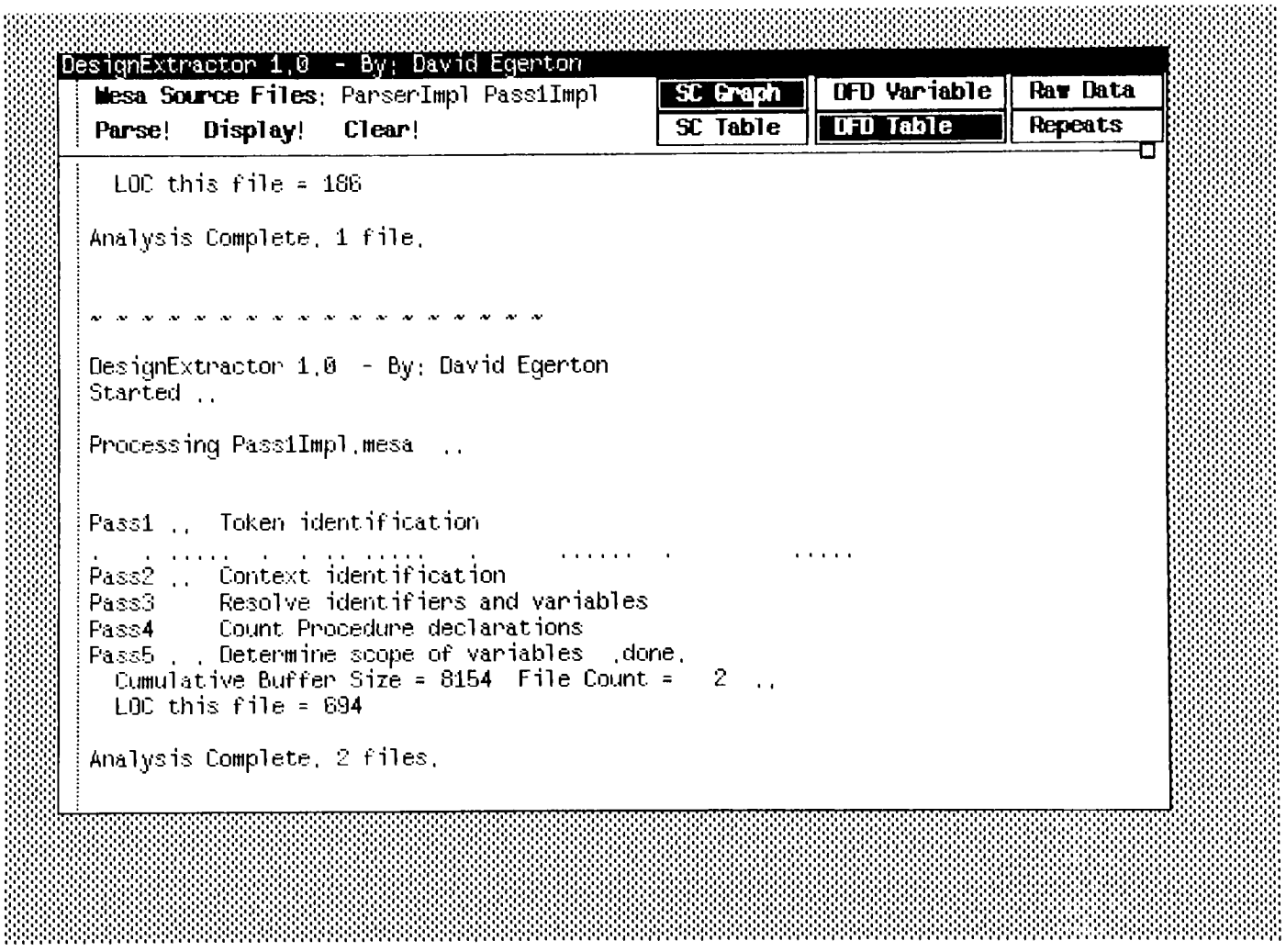
RawData has conflict with all other selections (except Repeats). When this button is selected all others must be deselected. An explanatory conflict message is displayed.


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files: ParserImpl
Parse! Display! Clear!
SC Graph
SC Table
DFD Variable
DFD Table
Raw Data
Repeats
DesignExtractor 1.0 - By: David Egerton
DesignExtractor 1.0 - By: David Egerton
Started
Processing ParserImpl.mesa ..
Pass1 Token identification
Pass2 .. Context identification
Pass3 Resolve identifiers and variables
Pass4 . Count Procedure declarations
Pass5 . . Determine scope of variables .done.
Cumulative Buffer Size = 1780 File Count = 1 .
LOC this file = 186
Analysis Complete, 1 file.


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files: ParserImpl Pass1Impl
Parse! Display! Clear!
SC Graph
DFD Variable
Raw Data
SC Table
DFD Table
Repeats
~ ~ ~ ~ Processing Display ~ ~ ~ ~ ~
Parsed file statistics written....
Missing export information written.
Creating Structure Chart....
- Studying repeat calls.. .
- Creating Semi Graphical Output.
- Creating Tabular Output....
Done Output created to file StructChart.data
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
Creating DataFlow Chart. .
- Creating variable table.. ..
- Constucting Context level diagram.
- Constucting level zero diagram..
- Constucting lower level diagram...
- Writing dataflow table.
Done Output created to file DataFlow.data
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files: ParserImpl Pass1Impl
Parse! Display! Clear!
SC Graph
SC Table
DFD Variable
DFD Table
Raw Data
Repeats
Pass4 .. Count Procedure declarations
Pass5 .. Determine scope of variables .done.
Cumulative Buffer Size = 8154 File Count = 2 ..
LOC this file = 694 ..
Analysis Complete. 2 files.
~ ~ ~ ~ Processing Display ~ ~ ~ ~ ~
Error - too many display selections set . .
Please select according to the following guidelines ..... ...
If'Raw Data' required then turn off selections of'SC Graph',
'SC Table','DFD Variable'and'DFD Table'
OR
IF another selection required, turn off 'Raw Data'
Note: Debugg works with any combination.
~ ~ ~ ~ Display Complete ~ ~ ~ ~ ~ ~


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files: ParserImpl Pass1Impl
Parse! Display! Clear!
SC Graph
DFD Variable
Raw Data
SC Table
DFD Table
Repeats
~ ~ ~ ~ Display Complete ~ ~ ~ ~ ~ ~
~ ~ ~ ~ Processing Display ~ ~ ~ ~ ~
Creating output file ..
Lines processed ... 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100,
1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400,
2500, 2600, 2700, 2800, 2900, 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700,
3800, 3900, 4000, 4100, 4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000,
5100, 5200, 5300, 5400, 5500, 5600, 5700, 5800, 5900, 6000, 6100, 6200, 6300,
6400, 6500, 6600, 6700, 6800, 6900, 7000, 7100, 7200, 7300, 7400, 7500, 7600,
7700, 7800, 7900, 8000, 8100,
Done . Output created to file Extractor.data
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ Display Complete ~ ~ ~ ~ ~ ~


DesignExtractor 1.0 - By: David Egerton
Mesa Source Files: ParserImpl Pass1Impl
Parse! Display! Clear!
SC Graph
SC Table
DFD Variable
DFD Table
Raw Data
Repeats
LOC this file = 186
Analysis Complete, 1 file.
DesignExtractor 1.0 - By: David Egerton
Started ..
Processing Pass1Impl.mesa ..
Pass1 .. Token identification
Pass2 .. Context identification
Pass3 Resolve identifiers and variables
Pass4 Count Procedure declarations
Pass5 .. Determine scope of variables .done.
Cumulative Buffer Size = 8154 File Count = 2 ..
LOC this file = 694
Analysis Complete, 2 files.

# SECTION 2 - PROJECT DESCRIPTION

### 2.1.2 Inputs.

The input files required for this thesis will be the Mesa project files from an archive database. The files are stored as individual implementation modules and interface definitions at the source level. These will be brought onto the working desktop, from a file server, parsed by the Design Extractor program for relevant information then deleted if desired. The workstation disc memory proved to be more than sufficient to keep the files and the parsed information on the disc during a working session [ Lessons #2]

The Design Extractor program has been designed to parse a number of program files at one time. Any number [Limit #3], boundary or order can be specified, although there are limitations to the amount of data [Limit #1] that can be processed. These limitations are detailed in Section 2.1.4. Files should be on the workstation in the current directory.

Originally it was thought that Program Files, Interface Files and Configuration Files would all need to be parsed. However it turns out that, as a prerequisite of the program is that files should already have been successfully compiled before processing, nearly all [Limit #2] the information is present in the Program Files and therefore only these need to be parsed.

**Single files:** Parsing a single file will give a satisfactory result. Structure Chart and DataFlow diagrams can be produced detailing that file only. Where references to imported data or procedures are made these are detailed in the output tables.

# SECTION 2 - PROJECT DESCRIPTION

**More than one file:** Files can be stacked up and all parsed at once or input one [or several] at a time [Limit #3]. The program maintains a cumulative database and therefore does not perceive a difference between these input methods. The program however always uses file 1 (the first file in) as the reference, this is explained in further details under Structure Charts Section 2.1.1.1.

**Preferred order:** As explained in the last paragraph file 1 is the reference. Therefore to successfully parse a series of files and get the desired architectural picture the root file must be parsed first. This is because all calls are assumed to come from the root. Therefore the first file is always the one to where calls are traced back. Any order can be used after that, the program searches and finds the relevant files whatever the order.

**Groups of files:** To study a group of files and their interactions, simply input the root file then any other you wish connections to be made with. The program will work out the explicit calls between the files if there are any. If there are none this will also be obvious from the output.

**Configuration files:** It is a simple matter to understand a configuration grouping with this program. Simply open up the configuration file and what you see is a list of files in the configuration plus the starting file. The starting file, specified as the CONTROL (Mesa Keyword), should be used as the root to best understand the file relationship for this particular grouping. See the configuration code sample in Appendix B.1. To use the Design Extractor program simply type in the starting file, then type in all the other file names in the configuration file. The Design Extractor program will then produce output based on the analysis of that complete configuration file [Lessons #1].

# SECTION 2 - PROJECT DESCRIPTION

## 2.1.3 Outputs.

Output from the Design Extractor program can either be viewed on screen or printed. A maximum of three output files can be produced, generated during the Display operation. This database holds to a close approximation the data specified in 1.3.4. The output files are as follows:

**Extractor.data** - This file is produced when the RawData option is selected. This is basically a dump of the DesignExtractor database. See Section 2.1.3.1

**StructChart.data** - This file is produced when either SCGraph OR SCTable or both are selected from the tool window. The output is an analysis of the current files in the database resulting in a Structure Chart representation. See Section 2.1.3.2.

**DataFlow.data** - This file is produced when either DFDVariable or DFDTable or both are selected from the tool window. The output is an analysis of the current files in the database resulting in a Data Flow Diagram representation. See Section 2.1.3.3.

**2.1.3.1 Extractor.data.** A complete Design Extractor generated example of the printed output can be seen in Appendix A.1. A one page example, for illustration purposes is included on the next page. The meaning of each field on the output chart is explained below.

**Index #:** This field simply provides and sequential index number reference.

**Atom Description:** The actual string that makes up the Token or Atom for that line. This is the program token extracted during parse operation.

**Sym #:** Symbol Number, This number is closely related to the PGS grammar values provided by the first pass of the Mesa parser. Appendix C details the Symbol Number meaning. Each Atom has a unique numerical value. Mesa keywords, punctuation and operators all have unique numbers assigned. Identifiers take the number 1. Identifiers are therefore unresolved in this column.

# SECTION 2 - PROJECT DESCRIPTION

**Stmt Cnt:** Statement Count Number, each Mesa line of code is counted and incremented. The count is incremented on end of Mesa source line characters (that is the semi colon).

**BEnd lvl:** Begin End Level Number, This field increments on BEGIN atoms and decrements on END atoms. Thus a count of nested beginnings can be referenced. This assists with determining the scope of identifiers.

**Proc lvl:** Procedure Level Number, Increments on PROC headers, decrements at end of procedure. Thus a count of nested procedures can be referenced with this field.

**List mark:** List Mark Number, This field marks the beginning, end and type of a list. List types are detailed in Appendix G.

**id mark:** Identification Mark Number, This field resolves the identifiers. The exact meaning is determined from the code parse and recorded here. For example, a procedure identifier is marked 37 to show that this identifier is resolved as a procedure declaration, a 237 would be a procedure call etc.

**Var mark:** Variable Mark Number, this field tracks the procedure or file within which a variable is referenced. Each procedure has a unique number assigned.

**Var Scpe:** Variable Scope Indentification Character, this field identifies the resolved scope of the variables. Characters are assigned to designate the scope type. The complete list of possible scope characters is detailed in Appendix E. An example of a scope character is L = Local, i.e., declared within the current procedure, G = Global, declared within the current file etc.

**Read Write:** Read Write Identification Character, this field contains the character W if the variable is written too, and the character R if it is a read.

**Decl Loca:** Declaration Location, this field contains the number of the database index that defines the declaration. If this is a procedure, the call line will have the declaration value in this field. If it is a variable, the line where the variable is used will contain the declaration location for that variable in this field. etc.

**Import Op:** Import Operation, details whether I for Import or E for Exported.
Import Identifier: The name of the host atom.

**Type:** This field holds the type information about variables. This may be a user declared type or a Mesa Keyword type value.

**Values:** Holds the variable value information. This field is a rough approximation and has not been used for any specific purpose in this implementation.

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.2 StructChart.data.** A complete Design Extractor generated example of the printed output can be seen in Appendix A.2. It should be noted that two outputs are available for StructChart.data. First the SCGraph and second the SCTable. The original intent of the Thesis was to produce the table only which could be translated by hand into some kind of graphical output. However during the course of the work it was discovered that it was a simple matter to show the information in a semi graphical form [Lessons #5]. Thus two selections are available, described below.

At the front of the StructChart.data, in addition to the generic file listing (Section 2.1.3.4), is an output explaining the missing export information. This section is provided to illustrate that many interfaces may be referenced in the parsing session, however not all interfaces are loaded. The output then indicates which interfaces (or files) should be loaded if further information is required. If not it is knowledge enough to know that these interfaces are not included in the session. An example of the missing export information page is shown on the next page. The meaning of each field in the output chart is explained below.

**Line #:** A sequential line index number.

**Missing export:** Indicates which exported interface information is not available.

**Call Made From File:** Tells you where the call to the missing export was made.

**Procedure Called:** The name of the procedure call.

STRUCTURE CHART - TEXTURAL OUTPUT TABLE - MISSING EXPORT INFORMATION

The following is the list of unknown export information. The files loaded in this session do not contain the exported procedures that follow:-

| LINE # | MISSING EXPORT | CALL MADE FROM FILE | PROCEDURE CALLED |
|---|---|---|---|
| 1 | Put | DesignExtractorImpl.mesa | Text |
| 2 | ToolWindow | DesignExtractorImpl.mesa | Activate |
| 3 | Exec | DesignExtractorImpl.mesa | OutputProc |
| 4 | Tool | DesignExtractorImpl.mesa | Destroy |
| 5 | Exec | DesignExtractorImpl.mesa | RemoveCommand |
| 6 | Heap | DesignExtractorImpl.mesa | Delete |
| 7 | UserInput | DesignExtractorImpl.mesa | UserAbort |
| 8 | file | DesignExtractorImpl.mesa | delete |
| 9 | Utils | DesignExtractorImpl.mesa | CardToString |
| 10 | MStream | DesignExtractorImpl.mesa | Create |
| 11 | Format | DesignExtractorImpl.mesa | Line |
| 12 | String | DesignExtractorImpl.mesa | CopyToNewString |
| 13 | String | DesignExtractorImpl.mesa | FreeString |
| 14 | Stream | DesignExtractorImpl.mesa | SetPosition |
| 15 | Parser | DesignExtractorImpl.mesa | ScanInit |
| 16 | Utils | DesignExtractorImpl.mesa | ClearMetFile |
| 17 | Process | DesignExtractorImpl.mesa | Detach |
| 18 | Process | DesignExtractorImpl.mesa | SetPriority |
| 19 | Process | DesignExtractorImpl.mesa | Yield |
| 20 | Utils | DesignExtractorImpl.mesa | OpenMetFile |
| 21 | Utils | DesignExtractorImpl.mesa | MetFileLength |
| 22 | Utils | DesignExtractorImpl.mesa | CloseMetFile |
| 23 | Token | DesignExtractorImpl.mesa | StringToHandle |
| 24 | token | DesignExtractorImpl.mesa | FreeStringHandle |
| 25 | token | DesignExtractorImpl.mesa | MaybeQuoted |
| 26 | String | DesignExtractorImpl.mesa | Empty |
| 27 | Token | DesignExtractorImpl.mesa | FreeTokenString |
| 28 | MFile | DesignExtractorImpl.mesa | EnumerateDirectory |
| 29 | Format | DesignExtractorImpl.mesa | LongNumber |
| 30 | TextSW | DesignExtractorImpl.mesa | ForceOutput |
| 31 | String | DesignExtractorImpl.mesa | EquivalentSubStrings |
| 32 | String | DesignExtractorImpl.mesa | AppendStringAndGrow |
| 33 | fileList | DesignExtractorImpl.mesa | Add |
| 34 | ChangeTable | DesignExtractorImpl.mesa | Init |
| 35 | ChangeTable | DesignExtractorImpl.mesa | MakeTable |
| 36 | fileList | DesignExtractorImpl.mesa | AllEntryNames |
| 37 | ChangeTable | DesignExtractorImpl.mesa | CleanUp |
| 38 | FormSW | DesignExtractorImpl.mesa | AllocateItemDescriptor |
| 39 | FormSW | DesignExtractorImpl.mesa | StringItem |
| 40 | FormSW | DesignExtractorImpl.mesa | BooleanItem |
| 41 | FormSW | DesignExtractorImpl.mesa | CommandItem |
| 42 | Tool | DesignExtractorImpl.mesa | MakeFormSW |
| 43 | Tool | DesignExtractorImpl.mesa | UnusedLogName |
| 44 | Tool | DesignExtractorImpl.mesa | MakeFileSW |
| 45 | ToolDriver | DesignExtractorImpl.mesa | NoteSWs |
| 46 | HeraldWindow | DesignExtractorImpl.mesa | AppendMessage |
| 47 | Exec | DesignExtractorImpl.mesa | AddCommand |
| 48 | String | DesignExtractorImpl.mesa | Copy |
| 49 | MSegment | DesignExtractorImpl.mesa | Address |
| 50 | Runtime | DesignExtractorImpl.mesa | GetTableBase |

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.2.1 Structure Chart Graphical Output.** A complete example of the graphical output is shown in Appendix A - 2.1. A one page example of the semi graphical output is provided for illustration purposes on the next page. This table shows a semi graphical illustration of the dynamic call structure of the program. Calls between modules are clearly shown by examining the residing filename information. A very good approximation to a dataflow diagram can be understood from examining this output [Lessons #5]. The meaning of each field on the output chart is explained below.

**Line:** This field shows the numerical value of the procedure database. This database has all procedure calls and declarations entered in call order. (ie the dynamic representation of the program. Consequently as a procedure call may be repeated many times, for best representation they were suppressed [Lessons # 3]. It can be seen then that there are gaps in the sequence which represent the missing repeated procedure calls.

**Residing filename:** This field represents the File from which a procedure call is made.

**Interface name:** This field represents the interface through which the procedure is imported. The procedure may or may not be loaded in the database.

**Level:** This field represents the procedure call level. The number increments with each call and decrements with each return.

**SLC:** This field is used to represent Straight Line Code calls from the root file.

**Indented Procedure Call Levels:** This field gives the semi graphical nature of this table. For each procedure the spacing to the right is multiplied by the procedure level. This gives a very easily recognized indication of the call and return structure of the program under analysis. Although 12 is the maximum shown on the paper the maximum number of procedure call levels possible is shown in the limits section [Limit #4].

STRUCTURE CHART - SEMI GRAPHICAL OUTPUT TABLE

| Line | Called From Filename | Interface Name | Declared in Filename | Lvl | SLC | Indented Procedure Call Levels |
|---|---|---|---|---|---|---|
| 1 | DesignExtractorImpl. | FormSW | - | 1 | SLC | GenTextDisplay |
| 2 | DesignExtractorImpl. | - | DesignExtractorImpl. | 2 | | Enter |
| 3 | DesignExtractorImpl. | - | DesignExtractorImpl. | 2 | | Exit |
| 4 | DesignExtractorImpl. | Heap | - | 2 | | Delete |
| 5 | DesignExtractorImpl. | Heap | - | 2 | | Create |
| 6 | DesignExtractorImpl. | Process | - | 2 | | Detach |
| 7 | DesignExtractorImpl. | - | DesignExtractorImpl. | 2 | | TextDisplay |
| 8 | DesignExtractorImpl. | Heap | - | 3 | | Delete |
| 9 | DesignExtractorImpl. | - | DesignExtractorImpl. | 3 | | Exit |
| 10 | DesignExtractorImpl. | Process | - | 3 | | SetPriority |
| 11 | DesignExtractorImpl. | Process | - | 3 | | Yield |
| 12 | DesignExtractorImpl. | - | DesignExtractorImpl. | 3 | | log |
| 13 | DesignExtractorImpl. | Put | - | 4 | | Text |
| 14 | DesignExtractorImpl. | - | DesignExtractorImpl. | 3 | | CheckSelection |
| 15 | DesignExtractorImpl. | - | DesignExtractorImpl. | 4 | | log |
| 16 | DesignExtractorImpl. | Put | - | 5 | | Text |
| 31 | DesignExtractorImpl. | - | DesignExtractorImpl. | 3 | | CheckForAbort |
| 32 | DesignExtractorImpl. | UserInput | - | 4 | | UserAbort |
| 33 | DesignExtractorImpl. | Display | - | 3 | | Control |
| 34 | DisplayImpl.mesa | Display | - | 4 | | RawData |
| 35 | DisplayImpl.mesa | Display | - | 4 | | StructChart |
| 36 | StructChartImpl.mesa | Utils | - | 5 | | CheckForAbort |
| 37 | DesignExtractorImpl. | UserInput | - | 6 | | UserAbort |
| 38 | StructChartImpl.mesa | MStream | - | 5 | | WriteOnly |
| 39 | StructChartImpl.mesa | statsFile | - | 5 | | delete |
| 40 | StructChartImpl.mesa | - | StructChartImpl.mesa | 5 | | FormatData |
| 41 | StructChartImpl.mesa | - | StructChartImpl.mesa | 6 | | WriteFiles |
| 42 | StructChartImpl.mesa | - | StructChartImpl.mesa | 7 | | StatsWrite |
| 43 | StructChartImpl.mesa | Stream | - | 8 | | PutString |
| 44 | StructChartImpl.mesa | Stream | - | 7 | | PutChar |
| 61 | StructChartImpl.mesa | - | StructChartImpl.mesa | 7 | | CheckLength |
| 62 | StructChartImpl.mesa | String | - | 8 | | Length |
| 63 | StructChartImpl.mesa | - | StructChartImpl.mesa | 8 | | AddSpaces |
| 64 | StructChartImpl.mesa | String | - | 9 | | AppendStringAndGrow |
| 65 | StructChartImpl.mesa | - | StructChartImpl.mesa | 8 | | ReduceLength |
| 66 | StructChartImpl.mesa | String | - | 9 | | MakeString |
| 67 | StructChartImpl.mesa | String | - | 9 | | AppendSubString |
| 68 | StructChartImpl.mesa | String | - | 9 | | FreeString |
| 69 | StructChartImpl.mesa | String | - | 9 | | CopyToNewString |
| 73 | StructChartImpl.mesa | String | - | 7 | | FreeString |
| 136 | StructChartImpl.mesa | Utils | - | 6 | | log |
| 137 | DesignExtractorImpl. | Put | - | 7 | | Text |
| 138 | StructChartImpl.mesa | - | StructChartImpl.mesa | 6 | | WriteExportInfo |
| 139 | StructChartImpl.mesa | - | StructChartImpl.mesa | 7 | | CheckProcExists |
| 140 | StructChartImpl.mesa | String | - | 7 | | FreeString |
| 141 | StructChartImpl.mesa | - | StructChartImpl.mesa | 7 | | OutputExportInfo |
| 142 | StructChartImpl.mesa | - | StructChartImpl.mesa | 8 | | WriteExportTitles |
| 143 | StructChartImpl.mesa | - | StructChartImpl.mesa | 9 | | StatsWrite |
| 144 | StructChartImpl.mesa | Stream | - | 10 | | PutString |
| 145 | StructChartImpl.mesa | Stream | - | 9 | | PutChar |
| 186 | StructChartImpl.mesa | String | - | 8 | | CopyToNewString |
| 187 | StructChartImpl.mesa | Utils | - | 8 | | CardToString |
| 188 | StructChartImpl.mesa | - | StructChartImpl.mesa | 8 | | CheckLength |

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.2.2 Structure Chart Tabular Output.** A complete example of the tabular output is shown in Appendix A - 2.2. A one page example of the tabular output is provided for illustration purposes on the next page. The meaning of each field on the output chart is explained below. This presentation was the original one indicated at thesis proposal time.

| CLIENT FIELDS | |
|---|---|
| **Lvl** [Lessons #4] | This field indicates the level of the procedure, from SLC up to max. SLC is a call from the root level, level 1 is a declaration at root level. [Lessons #4]. |
| **Rep** [Lessons #3] | This field shows the repeat information on the procedure. There are three types G = Global Repeat, L = Local Repeat, T = Twin Repeat. See Appendix F for explanation of repeats. |
| **Procedure Name** | This field indicates the client procedure name. |
| **Interface Name** | This field indicates the interface through which this procedure is called, if applicable. |
| **File Name** | This field indicates from which file the procedure call resides in. |

| SUBORDINATE FIELDS | |
|---|---|
| **Lvl** | This field indicates the level of the procedure. |
| **Rep** | This field shows the repeat information on the procedure. There are three types G = Global Repeat, L = Local Repeat, T = Twin Repeat. See [Lessons #3] Appendix F for explanation of repeats. |
| **Procedure Name** | This field indicates the client procedure name. |
| **Interface Name** | This field indicates the interface through which this procedure is called, if applicable. |
| **File Name** | This field indicates from which file the procedure call resides in. |

STRUCTURE CHART - TABULAR OUTPUT LISTING

| CLIENT | | | | | SUBORDINATES | | | | |
|---|---|---|---|---|---|---|---|---|---|
| LVL | REP | PROCEDURE NAME | INTERFACE NAME | FILE NAME | LVL | REP | PROCEDURE NAME | INTERFACE NAME | FILE NAME |
| SLC | | GenTextDisplay | FormSW | DesignExtractor | | | | | |
| | | | | | 2 | LG | Enter | - | DesignExtractor |
| | | | | | 2 | LG | Exit | - | DesignExtractor |
| | | | | | 2 | G | Delete | Heap | DesignExtractor |
| | | | | | 2 | LG | Create | Heap | DesignExtractor |
| | | | | | 2 | LG | Detach | Process | DesignExtractor |
| | | | | | 2 | | TextDisplay | - | DesignExtractor |
| SLC | | ClearOutputFile | FormSW | DesignExtractor | | | | | |
| | | | | | 2 | LG | Enter | - | DesignExtractor |
| | | | | | 2 | LG | Exit | - | DesignExtractor |
| | | | | | 2 | LG | Detach | Process | DesignExtractor |
| | | | | | 2 | | ClearIt | - | DesignExtractor |
| SLC | | Run | FormSW | DesignExtractor | | | | | |
| | | | | | 2 | LG | Enter | - | DesignExtractor |
| | | | | | 2 | LG | Exit | - | DesignExtractor |
| | | | | | 2 | LG | Detach | Process | DesignExtractor |
| | | | | | 2 | | RunIt | - | DesignExtractor |
| SLC | | EnumerateFiles | FormSW | DesignExtractor | | | | | |
| | | | | | 2 | | Init | ChangeTable | DesignExtractor |
| | | | | | 2 | | MakeTable | ChangeTable | DesignExtractor |
| | | | | | 2 | G | Add | fileList | DesignExtractor |
| | | | | | 2 | G | EnumerateDirectory | MFile | DesignExtractor |
| | | | | | 2 | | AddEnumerating | - | DesignExtractor |
| | | | | | 2 | | AllEntryNames | fileList | DesignExtractor |
| SLC | | FreeNameDescriptor | FormSW | DesignExtractor | | | | | |
| | | | | | 2 | G | CleanUp | ChangeTable | DesignExtractor |
| SLC | G | GetTableBase | Runtime | DesignExtractor | | | | | |
| SLC | G | AllocateAtomStack | - | DesignExtractor | | | | | |
| | | | | | 2 | LG | Create | MSegment | DesignExtractor |
| | | | | | 2 | TLG | Address | MSegment | DesignExtractor |
| SLC | | ExternalStoreList | - | DesignExtractor | | | | | |
| | | | | | 2 | TLG | CopyToNewString | String | DesignExtractor |
| SLC | | Initialize | - | DesignExtractor | | | | | |
| | | | | | 2 | | AppendMessage | HeraldWindow | DesignExtractor |
| | | | | | 2 | TLG | Create | Heap | DesignExtractor |
| | | | | | 2 | | AddCommand | Exec | DesignExtractor |
| | | | | | 2 | | ExecutiveProc | - | DesignExtractor |
| | | | | | 2 | | Unload | - | DesignExtractor |
| | | | | | 2 | | Copy | String | DesignExtractor |
| | | | | | 2 | G | FreeString | String | DesignExtractor |
| | | | | | 2 | LG | CopyToNewString | String | DesignExtractor |
| | | | | | 2 | | MakeSWs | - | DesignExtractor |
| | | | | | 2 | G | log | - | DesignExtractor |
| 2 | LG | Enter | - | DesignExtractor | | | | | |
| 2 | LG | Exit | - | DesignExtractor | | | | | |
| 2 | G | Delete | Heap | DesignExtractor | | | | | |
| 2 | LG | Create | Heap | DesignExtractor | | | | | |
| 2 | LG | Detach | Process | DesignExtractor | | | | | |
| 2 | | TextDisplay | - | DesignExtractor | | | | | |
| | | | | | 3 | TLG | Delete | Heap | DesignExtractor |

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.3 DataFlow.data.** A Design Extractor generated example of the printed output can be seen in Appendix A.3. It should be noted that two outputs are available for DataFlow.data, the DFDVariable table and second the DFDTable. The original intent of the Thesis was to produce the DFDTable only which could be translated by hand into some kind of graphical dataflow output. However during the course of the work it was discovered that it was very difficult to conceptualize the dynamic call flow. The parsed code is obviously presented in its static order. Therefore an interim state, DFDVariable table, was produced to aid understanding of the outputs of the program, this is presented in static order [Lessons #6]. The DFDTable output however is still used for producing the final DFD graphical display. This is presented in dynamic call order.

**2.1.3.3.1 DataFlow Variable Table Output.** A complete example of a variable table output is shown in Appendix A-3.1. A one page example of the variable table output is provided for illustration purposes on the next page. This table indicates the static order of the variables in the parsed program listings. The table shows the file names, the procedure names and the variable names in order of discovery with a sequential parse. Variable declarations are not listed, only variable usage (ie Read or Write) [Lessons #7]. The meaning of each field on the output chart is explained below.

**vSP Cnt:** This field indicates the sequential count in the variable stack database.

**Program Name:** This field is reserved for filename, these are programs or monitor file names.

**Procedure Name:** This field indicates procedure names, declarations not calls.

**Proc Count:** Assigns a unique number to each procedure, helps determine scope and identify variable with procedures later in analysis.

# SECTION 2 - PROJECT DESCRIPTION

**Var Scope:** Variable Scope Indentification Character, this field identifies the resolved scope of the variables. Characters are assigned to designate the scope type. The complete list of possible scope characters are detailed in Appendix E. An example of a scope character is L = Local, i.e., declared within the current procedure, G = Global, declared within the current file etc.

**Write Count:** Each variable is only shown once within the current scope. Each time a write to this variable is done the count is incremented.

**Read Count:** Each variable is only shown once within the current scope. Each time a read to this variable is done the count is incremented.

**Var Loca:** The location of the variable in the Main DesignExtractor database.

**Decl Loca:** The location of the variable declaration in the DesignExtractor database.

**Variable Name:** The name of the variable.

**Var Type:** Variable type designation. For example Call Parameter, Database etc. A complete listing of variable types is shown in Appendix D.

**Procedure Call Name:** When a variable type is 'Call Parameter' then the name of the called procedure is placed here.

VARIABLE DUPUT TABLE

| vSP Cnt | Program Name | Procedure Name | Proc Count | Var Scope | Write Count | Read Count | Var Loca | Decl Loc | Variable name | Var Type | Procedure Call Name |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | DesignExtractorImpl | | | | | | | | | | |
| 3 | | | 0 | M | 0 | 2 | 480 | 478 | systemZone | DataBase | |
| 10 | | | 0 | M | 0 | 1 | 544 | 0 | window | DataBase | |
| 11 | | | 0 | M | 0 | 1 | 546 | 0 | formSW | DataBase | |
| 12 | | | 0 | M | 0 | 1 | 684 | 0 | Other | DataBase | |
| 255 | | | 0 | M | 0 | 1 | 3716 | 0 | MesaTab | Call Param | GetTableBase |
| 256 | | | 0 | M | 0 | 1 | 3761 | 3495 | MaxString | Call Param | AllocateAtomStack |
| 4 | | Enter | | | | | 482 | 0 | | | |
| 5 | | | 1 | R | 0 | 1 | 501 | 488 | ok | Return Valu | |
| 6 | | | 1 | G | 0 | 1 | 504 | 352 | running | Return Valu | |
| 7 | | | 1 | G | 1 | 0 | 509 | 352 | running | Return Valu | |
| 8 | | Exit | | | | | 517 | 0 | | | |
| 9 | | | 2 | G | 1 | 0 | 528 | 352 | running | DataBase | |
| 13 | | log | | | | | 706 | 706 | | | |
| 14 | | | 3 | M | 0 | 1 | 711 | 709 | StringProc | DataBase | |
| 15 | | | 3 | P | 0 | 1 | 718 | 548 | fileSW | Call Param | Text |
| 16 | | | 3 | M | 0 | 1 | 720 | 0 | s | Call Param | Text |
| 17 | | ExecutiveProc | | | | | 725 | 0 | | | |
| 18 | | | 4 | M | 0 | 1 | 729 | 727 | ExecProc | DataBase | |
| 19 | | | 4 | M | 0 | 1 | 736 | 0 | window | Call Param | Activate |
| 20 | | Unload | | | | | 741 | 0 | | | |
| 21 | | | 5 | M | 0 | 1 | 745 | 743 | ExecProc | DataBase | |
| 22 | | | 5 | M | 0 | 1 | 763 | 0 | h | Call Param | DutputProc |
| 23 | | | 5 | M | 0 | 1 | 771 | 0 | abort | Return Valu | |
| 24 | | | 5 | M | 0 | 1 | 786 | 0 | window | DataBase | |
| 25 | | | 5 | M | 1 | 1 | 796 | 0 | window | Call Param | Destroy |
| 26 | | | 5 | M | 0 | 1 | 809 | 0 | h | Call Param | RemoveCommand |
| 27 | | | 5 | G | 0 | 1 | 811 | 665 | execCommand | Call Param | RemoveCommand |
| 28 | | | 5 | P | 0 | 1 | 816 | 536 | tempZone | DataBase | |
| 29 | | | 5 | P | 1 | 1 | 826 | 536 | tempZone | Call Param | Delete |
| 30 | | | 5 | G | 0 | 1 | 837 | 534 | permZone | DataBase | |
| 31 | | | 5 | G | 1 | 1 | 847 | 534 | permZone | Call Param | Delete |
| 32 | | | 5 | G | 0 | 1 | 858 | 448 | atomStackPtr | DataBase | |
| 33 | | | 5 | G | 0 | 1 | 866 | 3622 | atomStackPtr | Call Param | FreeAtomStack |
| 34 | | | 5 | G | 0 | 1 | 868 | 3628 | sh | Call Param | FreeAtomStack |
| 35 | | | 5 | G | 0 | 1 | 875 | 432 | fileStackPtr | DataBase | |
| 36 | | | 5 | G | 0 | 1 | 893 | 454 | atomIndexPtr | DataBase | |
| 37 | | CheckForAbort | | | | | 917 | 706 | | | |
| 38 | | | 6 | M | 0 | 1 | 928 | 0 | window | Call Param | UserAbort |
| 39 | | | 6 | M | 0 | 1 | 932 | 0 | ABORTED | DataBase | |
| 40 | | ProcessFile | | | | | 936 | 0 | | | |
| 41 | | | 7 | M | 0 | 1 | 940 | 938 | EnumerateProc | DataBase | |
| 55 | | | 7 | M | 0 | 4 | 1126 | 1124 | Error | DataBase | |
| 56 | | | 7 | M | 0 | 1 | 1134 | 1132 | BoundsFault | DataBase | |
| 57 | | | 7 | M | 0 | 3 | 1143 | 0 | error | DataBase | |
| 58 | | | 7 | M | 0 | 1 | 1181 | 0 | release | DataBase | |
| 59 | | | 7 | M | 0 | 1 | 1204 | 0 | log | Call Param | Line |
| 60 | | | 7 | M | 0 | 1 | 1206 | 0 | name | Call Param | Line |
| 61 | | | 7 | M | 0 | 1 | 1221 | 0 | name | Call Param | log |
| 62 | | | 7 | G | 0 | 1 | 1240 | 460 | MaxFiles | DataBase | |
| 63 | | | 7 | L | 1 | 0 | 1248 | 966 | tempString | DataBase | |

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.3.2 DataFlow Tabular Output.** A complete example of the DFDTable is shown in Appendix A - 3.1. A one page example of the tabular output is provided for illustration purposes on the next page. This table was the intended output of the thesis. From this a Data Flow Diagram can be constructed manually. Section 3.2 of this thesis details how the construction of the DFD picture is achieved from this data. The chart shows the dynamic structure of the program under analysis. To achieve this table the SCTable and DFDVariable tables are combined to provide the variable information relevant in the order of dynamic call. The meaning of each field on the output chart is explained below.

| NODE FIELDS<br>The node under consideration, a file, program or procedure | |
|---|---|
| **Lvl** | The call level of the node, from SLC to max [Limit #4] |
| **Node Name** | The identifier for that node |
| **Type** | Terminal or non terminal |

| DATA FIELDS<br>An analysis of the data, basically every time a variable is used. | |
|---|---|
| **Name** | The name of the data item |
| **Type** | A type character designator, See Appendix D for a complete list of available types. |
| **Scope** | A scope character designator, See Appendix E for a complete list of available scopes. |
| **Wr** | A count of the number of times the variable was written to in the current scope. |
| **Rd** | A count of the number of times the variable was read within the current scope. |

| CONNECTION FIELDS<br>The node to which the data is connects to, i.e., the data travels between the node and the connection. | |
|---|---|
| **Name** | The name of the connection node |
| **Declared** in | The name of the file the connection node is declared in. |
| **Alias** | The correlation of the formal names with the call names. A variable used in a call is given its formal value when the procedure is used. |

DATA FLOW DIAGRAM TEXTURAL OUTPUT TABLE

| NODE | | | DATA | | | | | CONNECTION | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Lvl | Node Name | Type | Name | Type | Scope | Wr | Rd | Name | Declared in | Alias |
| | | | ok | IntCallP | Local | 0 | 1 | log | Exit | ok |
| 2 | RunIt | nonT | | | | | | | | |
| | | | background | OataBase | Public | 0 | 1 | - | - | - |
| | | | priorityBackground | IntCallP | Imported | 0 | 1 | SetPriority | RunIt | Process |
| | | | toolName | IntCallP | Global | 0 | 1 | log | Exit | toolName |
| | | | totalErrors | OataBase | Global | 1 | 1 | totalErrors | Exit | - |
| | | | totalWarnings | DataBase | Global | 1 | 1 | totalWarnings | Exit | - |
| | | | tempZone | DataBase | Public | 1 | 1 | - | - | - |
| | | | tempZone | IntCallP | Public | 1 | 1 | Delete | RunIt | tempZone |
| | | | Error | IntCallP | Imported | 0 | 1 | OpenMetFile | RunIt | MFile |
| | | | Error | IntCallP | Imported | 0 | 1 | OpenMetFile | RunIt | MStream |
| | | | aborted | DataBase | Local | 2 | 2 | - | - | - |
| | | | initialFileLength | DataBase | Local | 1 | 0 | - | - | - |
| | | | filename | DataBase | Global | 0 | 1 | filename | Exit | - |
| | | | ABORTEO | OataBase | Imported | 0 | 1 | ABORTED | - | - |
| | | | filename | IntCallP | Global | 0 | 1 | StringToHandle | RunIt | filename |
| | | | token | DataBase | Local | 2 | 1 | - | - | - |
| | | | UnterminatedQuote | IntCallP | Imported | 0 | 1 | MaybeQuoted | RunIt | Token |
| | | | s | IntCallP | Local | 1 | 2 | Empty | RunIt | s |
| | | | s | IntCallP | Local | 1 | 1 | FreeTokenString | RunIt | s |
| | | | s | IntCallP | Local | 0 | 1 | AppendExtensionIfNe | RunIt | to |
| | | | s | IntCallP | Local | 0 | 1 | FreeTokenString | RunIt | s |
| | | | aborted | IntCallP | Local | 0 | 1 | log | Exit | aborted |
| | | | log | IntCallP | Imported | 0 | 1 | LongNumber | RunIt | - |
| | | | fileCount | IntCallP | Global | 0 | 1 | LongNumber | RunIt | fileCount |
| | | | fileCount | IntCallP | Global | 0 | 1 | log | Exit | fileCount |
| | | | log | IntCallP | Imported | 0 | 1 | LongNumber | RunIt | - |
| | | | totalErrors | IntCallP | Global | 0 | 1 | LongNumber | RunIt | totalErrors |
| | | | totalErrors | IntCallP | Global | 0 | 1 | log | Exit | totalErrors |
| | | | log | IntCallP | Imported | 0 | 1 | LongNumber | RunIt | - |
| | | | totalWarnings | IntCallP | Global | 0 | 1 | LongNumber | RunIt | totalWarnings |
| | | | totalWarnings | IntCallP | Global | 0 | 1 | log | Exit | totalWarnings |
| | | | tempZone | IntCallP | Public | 1 | 1 | Delete | RunIt | tempZone |
| | | | fileSW | IntCallP | Public | 0 | 1 | ForceOutput | RunIt | fileSW |
| 2 | Init | Term | | | | | | | | |
| 2 | MakeTable | Term | | | | | | | | |
| 2 | Add | Term | | | | | | | | |
| 2 | EnumerateDirectory | Term | | | | | | | | |
| 2 | AddEnumerating | nonT | | | | | | | | |
| | | | EnumerateProc | DataBase | Imported | 0 | 1 | EnumerateProc | - | - |
| | | | name | IntCallP | Imported | 0 | 1 | Add | AddEnumerating | - |
| 2 | AllEntryNames | Term | | | | | | | | |
| 2 | CleanUp | nonT | | | | | | | | |
| | | | ok | FmlParam | - | 0 | 0 | - | - | - |
| | | | file | DataBase | External | 0 | 1 | file | ProcessFile | - |
| | | | delete | DataBase | Imported | 0 | 1 | delete | - | - |
| | | | file | IntCallP | External | 2 | 1 | delete | CleanUp | file |
| | | | ok | IntCallP | UseFmlCal | 0 | 1 | log | Exit | ok |
| | | | errors | IntCallP | Local | 1 | 3 | log | Exit | errors |
| | | | warnings | IntCallP | Local | 0 | 2 | log | Exit | warnings |
| | | | log | DataBase | Imported | 0 | 2 | log | - | - |

# SECTION 2 - PROJECT DESCRIPTION

**2.1.3.4 Generic front sheet.** In Appendix A a common front sheet is generated for all three output files. This is done to enable an easy identification of the files parsed in this session without having to plough through the detailed output. A generic example is included on the next page. The meaning of each field on the output chart is explained below.

**Index:** This is a count of the files processed.

**File Name:** This is the full name of the file processed.

**Start Index:** This field indicates the Design Extractor database starting position for the file in question.

**End Index:** This field indicates the Design Extractor database ending position for the file in question.

DESIGN EXTRACTION STATISTICS
==========

The files parsed in this session were:

| Index | File Name | Start Index | End Index | Exports to |
|---|---|---|---|---|
| 1 | DesignExtractorImpl.mesa | 1 | 3778 | Utils |
| 2 | ParserImpl.mesa | 3779 | 5558 | Parser |
| 3 | DisplayImpl.mesa | 5559 | 5725 | Display |
| 4 | StructChartImpl.mesa | 5726 | 12305 | Display |

# SECTION 2 - PROJECT DESCRIPTION

## 2.1.4. Limitations and restrictions.

**2.1.4.1 System Limitations and Restrictions.** These limits are referenced throughout the project description text and designated numbers as per below for ease in locating.

**[Limits #1]** **Data Base Maximum** = 40,000 lines. This field is specified in the Parser Interface file, designated "MaxString". DataBase name is AtomStack.

**[Limits #2]** **Missing interface information.** The Design Extractor program has been designed to parser program and module files only. Interface files are not parsed because most of the information is already available in the program files. However some database types are declared in the interface files. This information is therefore not available to DesignExtractor.

**[Limits #3]** **Max number of Source Files** = 30. The maximum number of source files that can be parsed in any one session is specified in the Parser interface, designated "MaxFiles".

**[Limits #4]** **Max number of Procedure Levels** = 300. The maximum number of procedure levels that can be analysed is specified in ProgramParserImpl, designated "maxProc".

**[Limits #5]** **Graphical representation.** As previously explained the focus of this thesis is to prove, or disprove, the theory that valid design information can be extracted from source code in the Mesa language. The thesis will work with the code until a level of understanding of this concept has been reached. The output of the software tools developed will be in the form of the textual tables illustrated above. Manual mapping of these will be done to graphically illustrate the conclusions more visibly (See Section 3]. A further course of study would be to provide a graphical engine that would automate the whole process. This activity has not been pursued in this thesis attempt.

**[Limits #6]** **Parallel processing.** The mesa language has constructs for parallel processing. These have been ignored for simplicity.

**2.1.4.2 Performance expectations**

The performance of DesignExtractor will clearly vary dependent on many factors, however the following guidelines can be used as an approximation.

# SECTION 2 - PROJECT DESCRIPTION

**Files used in sample:** Four reference files were chosen as an example set:

| | | | |
|---|---|---|---|
| 1. DesignExtractorImpl | 22883 Bytes | 768 LOC | 3778 DBase Lines |
| 2. DisplayImpl | 1489 Bytes | 39 LOC | 167 DBase Lines |
| 3. ParserImpl | 13386 Bytes | 300 LOC | 1779 DBase Lines |
| 4. StructChartImpl | 46478 Bytes | 1534 LOC | 6580 DBase Lines |
| TOTALS | 84236 Bytes | 2641 LOC | 12305 DBase Lines |

**Parse function:** Time to parse above files: 7 Minutes

Therefore average performance: 5 seconds per 1000 bytes OR

16 seconds per 100 LOC OR

3.5 seconds per 100 DBase Line

**Display function:** Several modes were chosen as follows:

Time to Produce SCGraph and DFDTable - Same timing as for Parse Function

Time to produce SCTable and DFDVariable - Same timing as for Parse function

Time to produce Extractor.data output = approx 20 Minutes
Average performance: 14 seconds per 1000 bytes OR
45 seconds per 100 LOC OR
10 seconds per 100 DBase Lines

### 2.1.5 System files.

The Xerox Mesa language environment runs on XDE, a comprehensive set of tools and utilities to enable development. Debug, analysis, compile and run tools are all provided with the system. The thesis project should therefore work properly within this environment. The thesis project was developed in "Tajo" an implementation of XDE that allows the environment to coexist with the suite of Viewpoint programs.

# SECTION 2 - PROJECT DESCRIPTION

## 2.2 System Specification

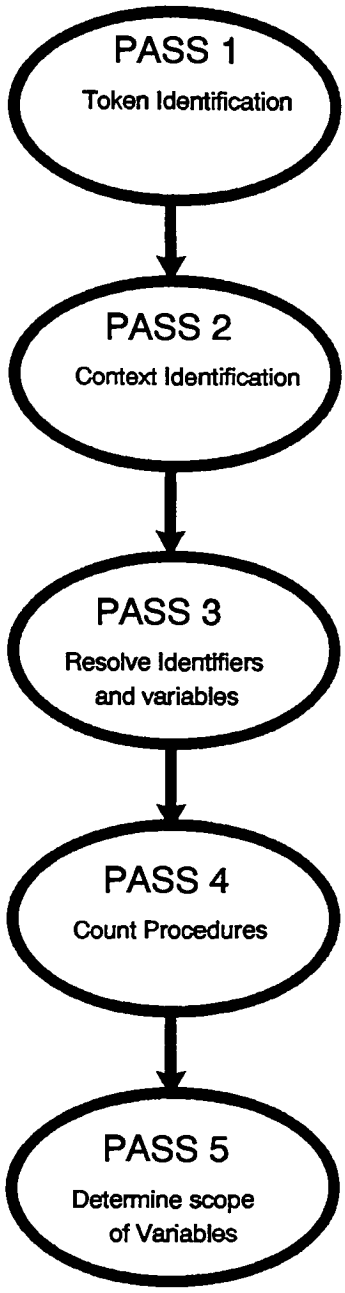
The software development that was attempted for this thesis is detailed in this section. The foundation design philosophies are discussed in Section 2.2.1. The Data Flow Diagrams are shown in Section 2.2.2 and the Structure Chart in Section 2.2.3. Brief discussions of the equipment configuration and required implementation tools are detailed in Sections 2.2.3 and 2.2.4.

### 2.2.1 Foundational Design Philosophies.

DesignExtractor was constructed as a parser and display program. Each file as previously discussed runs through a routines that strip out relevant data and organize it into usable fields in the program databases. The diagram on the next page [Figure 2.2.1] illustrates the basic functions of the parser.

The parser has five passes, Token identification, Context identification, Resolving identifiers and variables, Counting procedure declarations and Determining the scope of variables. Data is placed into the main program databases [Figure 2.2.2] and can be displayed in its entirety through RawData as described in Section 2.1.3.1. The database is always active whenever the tool is running.

# SECTION 2 - PROJECT DESCRIPTION




| Operation | Data produced |
|---|---|
| Pass1Impl/Atom: This section parses the raw source code files and sets the main program database (atomStack). | Extracts each token entity<br>Allocates a main database location<br>Determines PGS grammar #, App C<br>- Mesa keywords<br>- all identifiers<br>- all punctuation<br>quoted strings |
| ProgramParserImpl/Pass2This section parses the main program database. Identifying contexts and recording the result in the atom stack database. | Mark all BEGIN END blocks<br>Mark all statement boundaries<br>Mark all procedure boundaries<br>Mark all list boundaries<br>- Directory - Call parameter<br>- Import - Using<br>- Export - Other<br>- Parameter<br>- Return |
| ProgramParserImpl/Pass3: Each identifier has a value 1 at this stage. After parsing the main database the idMark field is updated to state the usage for each identifier. | Filter out non spec keywords<br>Mark procedure declarations<br>Mark procedure calls<br>Mark program and monitor<br>Mark variables plus read / write operation<br>Mark variables in lists |
| ProgramParserImpl/Pass4: Parses the main program database and uniquely numbers each procedure in static order. | Mark each procedure body with unique number<br>Mark nested procedures |
| ProgramParserImpl/Pass5: Parses the main program database determining the scope of each variable. Explained in more detail in Appendix E. | D = Dereferenced arrays fields<br>E = External database<br>G = Global Variable<br>I = Instantiated variables<br>L = Local, i.e., confined to this procedure<br>M = Imported<br>P = Public<br>R = Return parameter. |

**Table 2.2.1 - Overview of the parser**

# SECTION 2 - PROJECT DESCRIPTION

The display options use the main program database, atomStack, to produce the various outputs. Several passes of the database, either in a relevant local section or in total, are done while producing display outputs. The output formats are described in Section 2.1.3 and the internal working of the output programs are detailed later in the Structure Chart section, Section 2.2.3. The databases FullStack [declared in Display.mesa], VarStack[local to DataFlowImpl] and DFStack [ local to DataFlowImpl] are temporary databases created during display production and deleted when complete.

The outputs can either be viewed on screen or via printed output. As previously described, three output files are produced, Extractor.data, StructChart.data and DataFlow.data. These files remain in the current working directory even after the tool has been unloaded. The printed output shows in landscape form. Further details of the output formats are shown in section 2.1.3.

## 2.2.2 System Data Flow Diagrams.

The design of the system is modeled below in Data Flow Diagram form. Starting at the Context Diagram and working down through two levels of detail in the areas where this needs to be shown. Further levels of refinement have not been detailed here although in some cases they are available. However the design information at the more detailed levels is best produced by examining the tool outputs. See Appendix A and the Conclusions in section 3.0

# SECTION 2 - PROJECT DESCRIPTION

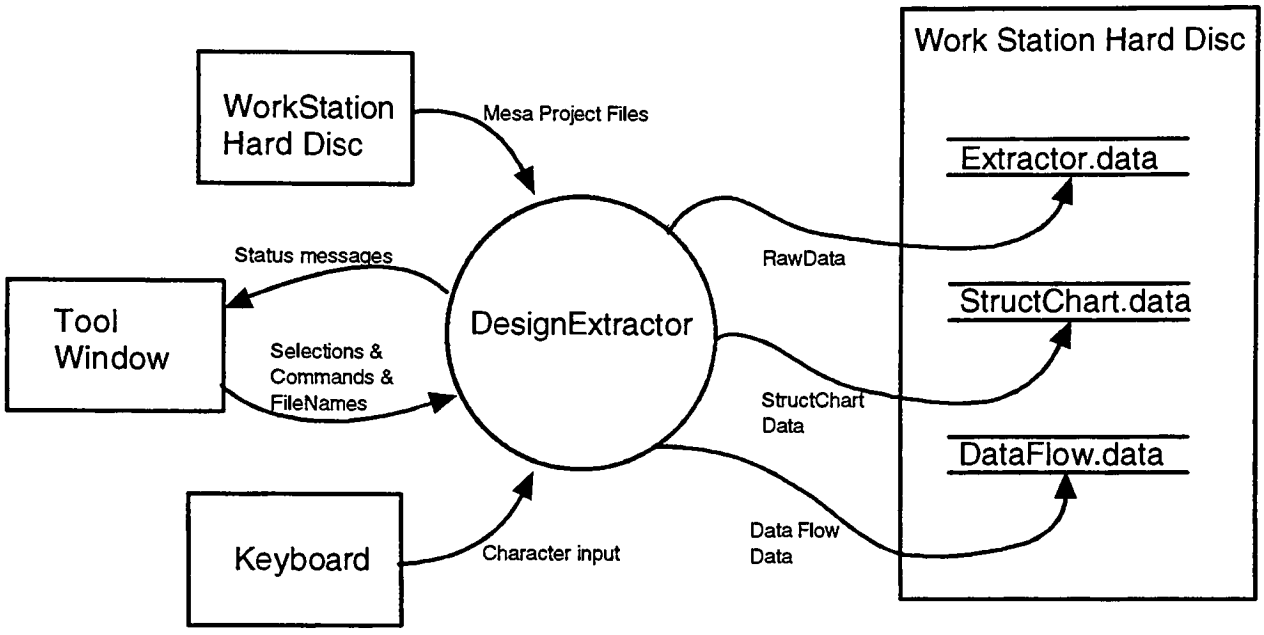



**Figure 2.2.2.1 - DesignExtractor Context Diagram**

The diagram above shows the context of the system. File name characters are typed in and input from the system keyboard to instruct the "DesignExtractor" System as to which Mesa project files to parse. The characters appear in the appropriate place in the tool window as file names. The tool window deals with four items. Status messages, which are interactively displayed as the tool is used. Selection options (eg DFDTable, SCTable etc - see section 2.1.1 describing the user interface). Commands that request action such as 'Parse' or 'Display'. Finally file names which are displayed as they are typed in to register which files to parse. The Mesa files are stored on the workstation hard disc and operated on by DesignExtractor according to the

# SECTION 2 - PROJECT DESCRIPTION

selections made. The three output files are produced as shown in the diagram. Formats of these outputs are discussed in section 2.1.3.

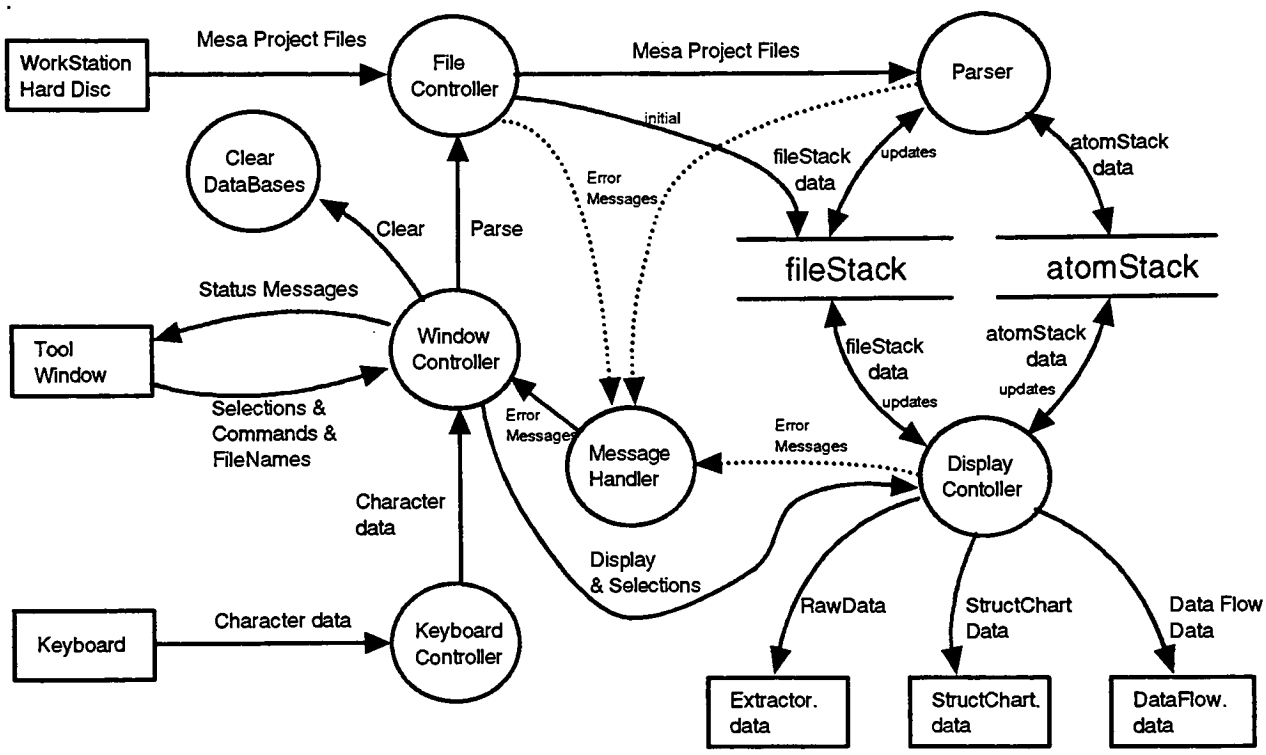



**Figure 2.2.2.2 - DesignExtractor Level 0 Data Flow Diagram**

The Level 0 diagram above breaks down the Context diagram a step further.The same inputs and outputs are detailed as for the top level but the single process bubble are now broken down into intermediate operations. Each process is described below.

**2.2.2.1 Keyboard Controller.** The Keyboard Controller is the method to input characters. The routines for this are embedded in the XDE environment and no special applications were developed for this program. The Keyboard controller inputs characters

to the control window (Section 2.1.1.1). The keyboard also has some control functions such as abort which is monitored throughout program operation. Abort will stop processing. The mouse operation is of course covered under this module, again XDE environment controlled.

**2.2.2.2 Window Controller.** The Window Controller sets up the User Interface for the tool. The layout of the window and the interactions with the tool are described in detail in section 2.1.1 The User Interface. Basically this window handles all interactions with the user with the exception of keyboard entry.

**2.2.2.3 File Controller.** This process deals with the mesa project files that the DesignExtractor program operates on. The handler waits for a Parser command then selects from the hard disc the files that have been keyed into the space for files in the User Interface window. See Section 2.1.1.3 for the control window interactions.

**2.2.2.4 Clear Database.** This process can be invoked to clear all data from the main atomStack and fileStack databases. If the information required from a particular session is no longer needed and a new set of files is to be parsed, the clear function should be invoked. After clear is completed atomStack and fileStack has null or default values in all locations. The contents of the output files Extractor.data, StructChart.data and DataFlow.data retain their data without modification.

# SECTION 2 - PROJECT DESCRIPTION

**2.2.2.5 File Parser.** The File Parser is responsible for controlling the Mesa Project Files, extracting the appropriate data from them and building the Parsed Data File. Mesa Project Files are stored on the workstation hard disc. When the file controller is instructed to bring in a file it is automatically parsed. The Parsed Data File contains all the required lexical symbols to complete the Textural Creations later on. Table 2.2.1 above gives a detailed description of the functions of each of the 5 passes within the parser algorithm. Figure 2.2.2.3 below shows the next level dataflow breakdown level for the Parser process.

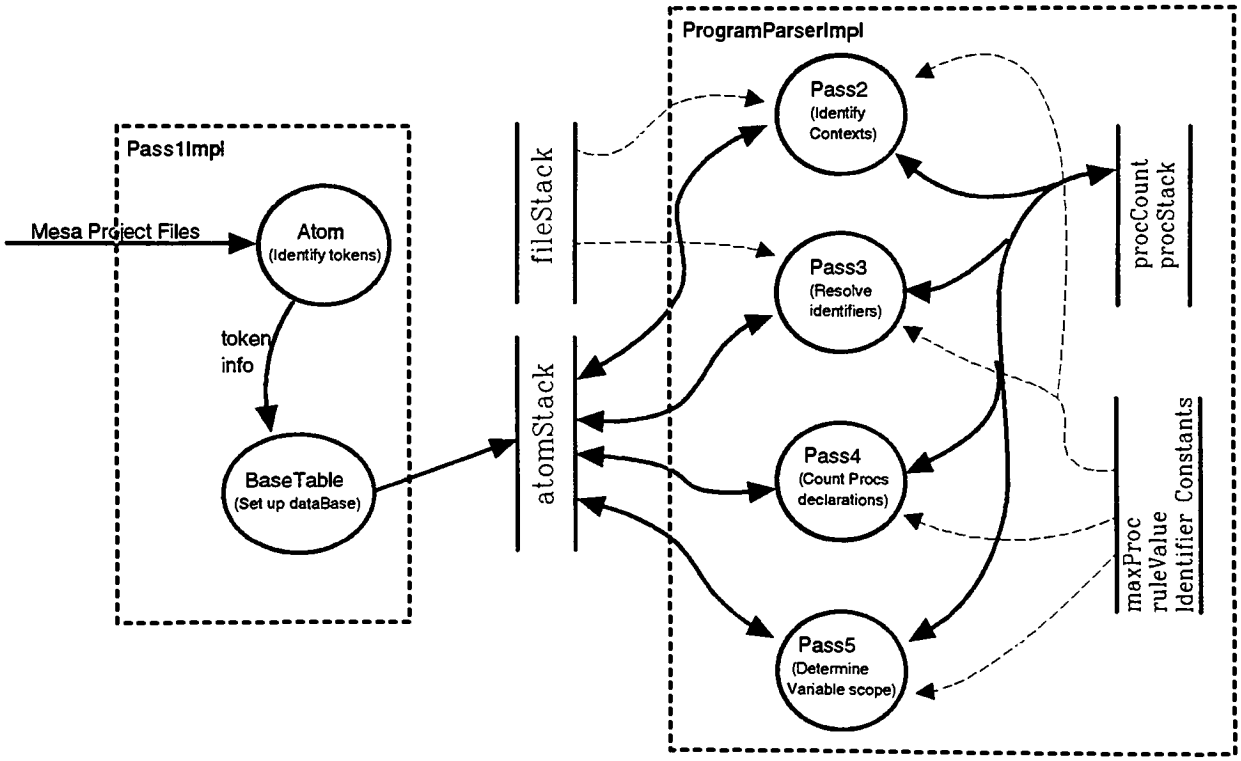



**Figure 2.2.2.3 - File Parser Level 1 DFD**

# SECTION 2 - PROJECT DESCRIPTION

The mesa project files are parsed one at a time. The first file is fed into atom (in Pass1Impl) and a token is created for every unique item identified. The unique items are then placed into the atomStack database through the procedure BaseTable. After this point operations are done solely on the atomStack database and the files are no longer referenced. The order of operation is Atom, BaseTable, Pass2, Pass3, Pass4, Pass5. Interim databases are used and discarded after processing is complete. These are indicated on the right hand side of Figure 2.2.2.3.

**2.2.2.6 Display Controller.** The Display Controller is the final top level module. It is responsible for control of the interactions with the output files. The structure of the output information is detailed in Section 2.1.3 Outputs. Figure 2.2.2.4 below shows the next level of dataflow breakdown of the Display Process.

# SECTION 2 - PROJECT DESCRIPTION

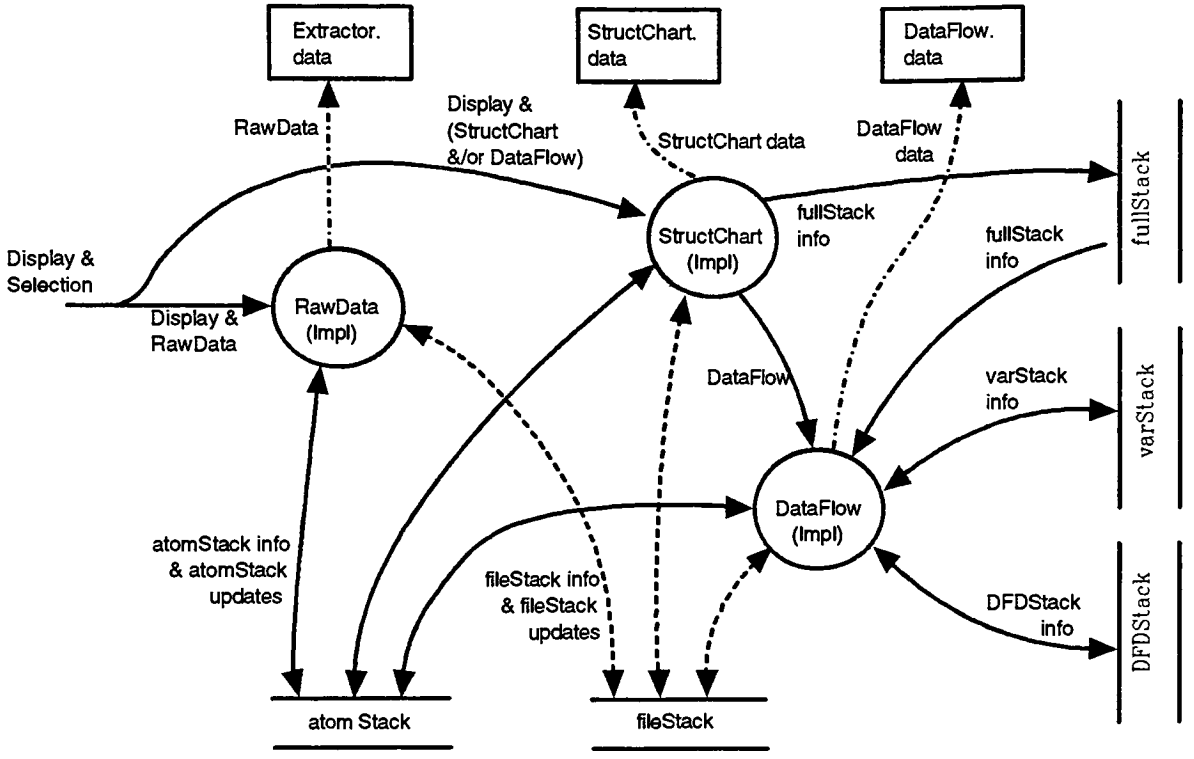



**Figure 2.2.2.4 - Display Level 1 DFD**

There are four interactions shown in the above diagram. First the input section which has the Selection and the Display command on the left hand side. The command selection determines which internal process is invoked. The database interaction is shown at the bottom of the diagram. Again the main source of data is the atomStack and fileStack databases. Information is mainly read from these databases, however some information about the relationship between procedure calls and declarations (for example) is only determined at this point and written into the atomStack. Internal databases are shown on the right. These are created during processing of a display, then discarded on completion. The final interaction is with the output files themselves, these processes are

responsible for opening, formatting and deposition of data and closing the appropriate files. The processes clear and overwrite any preexisting data in the output files.

If RawData is selected then only the RawData process is invoked. This produces a dump of the atomStack to the output file Extractor.data (See section 2.1.3.1). If StuctChart or DataFlow options are requested then StructChart is invoked. Even if no Struct Chart data is required for output the procedure has to be called to compile the structural information that is built on further in the dataflow program. If no dataflow options are requested then only Struct Chart needs to be invoked. Selection of Stuct Chart and Data Flow options will produce two output files during processing. Selection of RawData and other options are incompatible.

**2.2.2.7 Message Handler.** All messages are output to the Feedback Window of the user interface. This is achieved via a utility interface that is called whenever status or error messages are required to be sent. All messages from all modules are sent via this utility.

**2.2.2.8 Outputs.** The outputs have been adequately described previously. Section 2.1.3 goes into the detail of each output format produced. Files Extractor.data, StuctChart.data and DataFlow.data are the final representations of the DesignExtractor.

That completes the discussion of the Data Flow Diagrams.

# SECTION 2 - PROJECT DESCRIPTION

## 2.2.3 System Data Dictionary

The table below illustrates the data items used in the Data Flow diagrams of section 2.2.2. Note these are not all the data items used in the program.

| Data Item | Declared In | Data Elements | Meaning or Usage |
|---|---|---|---|
| atomStack Data [40000] | Parser | BELevel | - Cardinal, counts the Begin and End occurrences. |
| | | declLocation | - Cardinal, contains the declaration location of procedure calls, variables and dereferenced items. |
| | | fileNumber | - Cardinal, contains the file # of current location |
| | | fileType | - Defined, Program, Monitor, Configuration or Definition |
| | | idMark | - Cardinal, number representing resolved identifiers. |
| | | lEatom | - Long String, The import/ export token identifier |
| | | lEmark | - Character, I for import E for Export |
| | | Index | - Cardinal, Sequential count of database entries |
| | | InputString | - Long String, the current token |
| | | listMark | - Cardinal, A number representing the list type |
| | | Plevel | - Cardinal, Count of procedure levels |
| | | readWrite | - Character, R for Read and W for Write |
| | | ruleValue | - Cardinal, not used |
| | | sCount | - Cardinal, a count of Mesa statements |
| | | symbolNumber | - Cardinal, the PGS grammar number, see Appendix C |
| | | type | - Long String, the type of a variable |
| | | typeBase | - Long String, not used |
| | | varMark | - Cardinal, tracks variables to procedures |
| | | varScope | - Character, scope type, described in Appendix E |
| | | values | - Long String, values a variable may hold |
| Character data | | Any keyboard char | |
| Clear | DesignExtractor | Clear | - Command to erase all data in databases |
| Commands | DesignExtractor | Clear OR | - Command to erase all data in databases |
| | | Parse OR | - Command to initiate file parsing |
| | | Display OR | - Command to initiate display processing |
| Data Flow Data | | Character data | - Any keyboard character arranged and formatted appropriate to the data flow output requirements |

**Table continued on next page**

# SECTION 2 - PROJECT DESCRIPTION

Continued/-

| Data Item | Declared In | Data Elements | Meaning or Usage |
|---|---|---|---|
| DFD Stack info [1000] | DataFlowImpl | context | - Boolean, True if this is at context level |
| | | SLC | - Boolean, True if this is at Straight line code level |
| | | level | - Cardinal, count of procedure indent level |
| | | nodeName | - Cardinal, pointer to the nodeName in atomStack |
| | | nodeType | - Defined, Dummy, Non Terminal or Terminal |
| | | nextHIndex | - Cardinal, pointer to calling nodeName in atomStack |
| | | dataName | - Cardinal, pointer dataName in atomStack |
| | | dataType | - Defined, CallParam, DataBase, DataFlow, Dummy, External, Formal Param, IntCallParam, ReturnValue, FormalReturn. See Appendix D for further details. |
| | | parameterCount | - Cardinal, registers count of parameters in a proc' |
| | | connDone | - Boolean, True if connection location found |
| | | connIndex | - Cardinal, pointer to connecting node in atomStack |
| | | alias | - Cardinal, pointer to instantiated name in atomStack |
| | | declIn | - Cardinal, pointer to declaration location in atomStack |
| | | varCount | - Cardinal, the count of location used in the database |
| Display | DesignExtractor | Display | - Command to initiate display operation on atomStack |
| Error Messages | All modules | | - Too numerous to mention, error messages are transmitted whenever thresholds are exceeded or confiicts arise in selections |
| Filenames | DesignExtractor | Character data | - Any list of filenames without punctuation, each file separated by a space. The .mesa extension optional |
| fileStack Data [30] | Parser | fileName | - Long String, name of the mesa file |
| | | fileCount | - Cardinal, count of files processed |
| | | exportName | - Long String, Name the file it exports too. |
| | | startIndex | - Cardinal, pointer to atomStack start location |
| | | endIndex | - Cardinal, pointer to atomStack end location |
| | | procCount | - Cardinal, count of procedures in this file |
| | | prevLevel | - Cardinal, Not used |

Table continued on next page

# SECTION 2 - PROJECT DESCRIPTION

Continued/-

| Data Item | Declared In | Data Elements | Meaning or Usage |
|---|---|---|---|
| full Stack info | Display | callLocation | - Cardinal, pointer to procedure call in atomStack |
| | | callLevel | - Cardinal, register of procedure level of this call |
| | | declLocation | - Cardinal, pointer to declaration in atomStack |
| | | fileNumber | - Cardinal, pointer to current file number in fileStack |
| | | SLC | - Boolean, true if proc call is in Straight Line Code area |
| | | firstOfTwin | - Boolean, true if first occurrences of two adjacent calls of the the same procedure |
| | | firstOfLevel | - Boolean, true if first occurrences of a procedure that is called more than once at this call level |
| | | firstOfGlobal | - Boolean, true if first occurrences of call within this file |
| | | twinRepeat | - Boolean, true if second occurrences of twin |
| | | levelRepeat | - Boolean, true if > first occurrences of call at this level |
| | | globalRepeat | - Boolean, true if > first occurrence of call globally |
| | | vSPIndex | - Cardinal, pointer to variable stack database, locks procedure calls in each database together |
| Mesa Project Files | Found on hard disc | Compiled character | - Regular files, only guaranteed to work if they have first been successfully compiled. |
| Parse | DesignExtractor | Parse | - Command to initiate parse operation on mesa files |
| RawData | | Character data | - Any keyboard character arranged and formatted appropriate to the RawData output requirements |
| Selections | DesignExtractor | SC Graph | - Boolean, True if Structure Chart graph output req |
| | | SC Table | - Boolean, True if Structure Chart table output req |
| | | DFD Variable | - Boolean, True if Data Flow variable table output req |
| | | DFD Table | - Boolean, True if Data Flow table output required |
| | | Repeats | - Boolean, True if repeat procedure calls are required to be shown with the above output selections |
| | | RawData | - Boolean, True if dump of atomStack required, cannot request above selections if RawData selected |
| Status Messages | All modules | Character data | - Maintains feedback to user of operation of the tool. Messages sent to the feedback window of the UI |
| StructChart Data | | Character data | - Any keyboard character arranged and formatted appropriate to the Structure Chart output requirements |
| var Stack info | DataFlowImpl | index | Cardinal, pointer to variable location in atomStack |
| | | readCount | Cardinal, count of the number of read occurrences |
| | | writeCount | Cardinal, count of the number of write occurrences |

**Table 2.2.3.1 - DesignExtractor Data Dictionary**

# SECTION 2 - PROJECT DESCRIPTION

## 2.2.4 System Organization Chart

The diagram that appears on the next page is the Structure Chart for the system. This diagram does not require much explanation as it is directly derived from the DFD's of the last section. Extra modules have been added in certain areas to aid modularity and consistency. Also, Structure reflects some of the mesa operating environmental aids and constraints.

DesignExtractorImpl is the root program implementation file. This is where control resides of the operation of the User Interface window and consequently all use invoked commands and selections. The user interface is operated through the "Tool" window interface, a mesa template tool provided for the user to customize to specific needs. The Parse operation are all confined within the Parser Interface and the Display operation within the Display interface. The "S" designation is for Stub file. This files were available in a previous implementation which are not required here. The program modules have no active operation in DesignExtractor. The "U" operations are for utilize. This interface is called from all modules as required. Provides operation such as, changing cardinal values to strings, or logging feedback messages to the feedback window etc.

# SECTION 2 - PROJECT DESCRIPTION

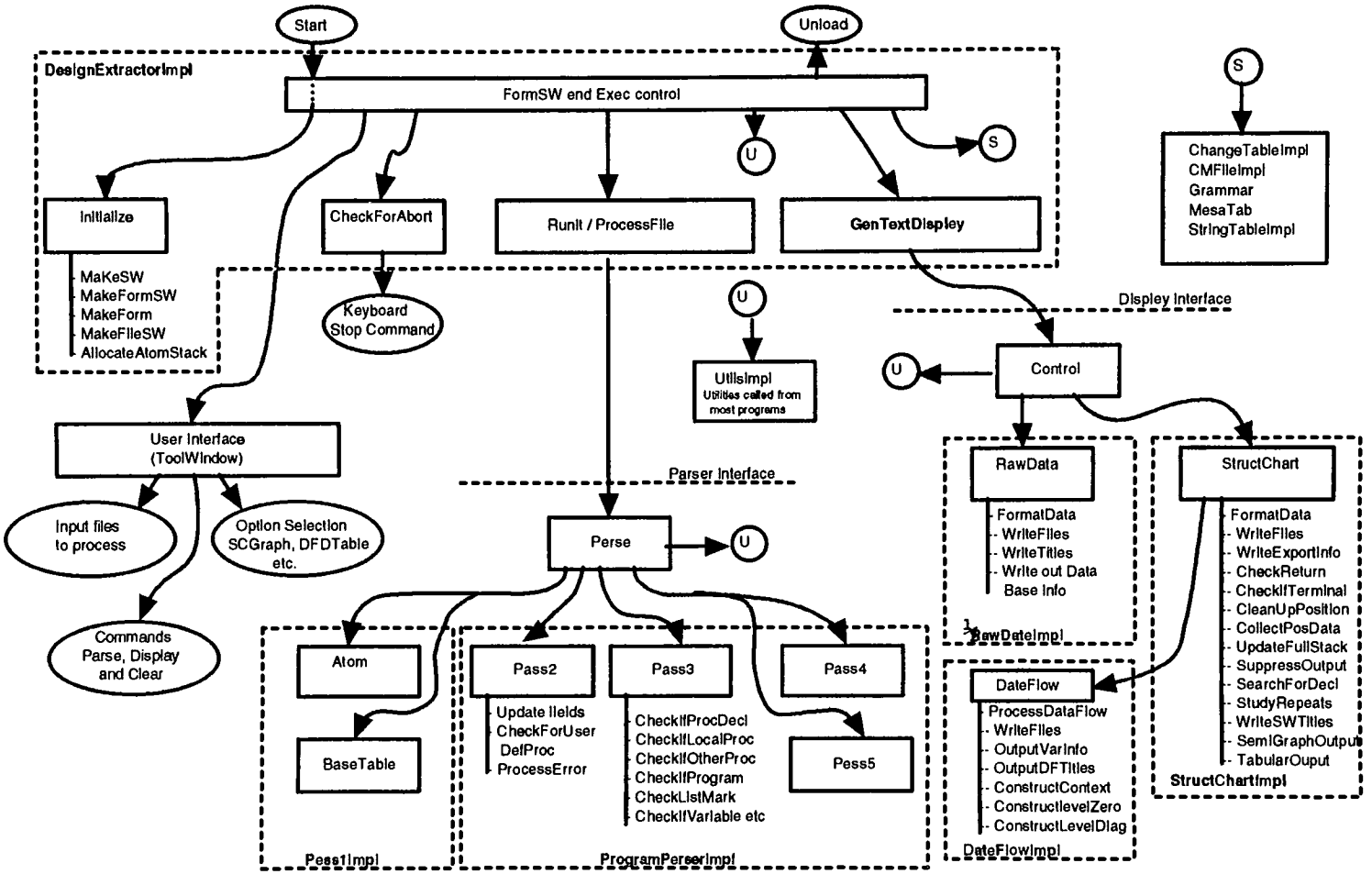



**Figure 2.2.4.1 - Structure Chart of Design Extractor**

# SECTION 2 - PROJECT DESCRIPTION

## 2.2.5 Equipment configuration

The following define the equipment configuration to be used for this thesis project:

| | |
|---|---|
| Workstation: | Xerox 6085 with 80Mbyte hard disk |
| Network: | Ethernet using Xerox XNS protocols |
| File Servers: | Interactive use of Project files from remote file server |
| Printer: | Network connected laser printers |

## 2.2.6 Implementation tools.

The following define the software implementation tools to be used:

| | |
|---|---|
| Environment: | Xerox Development Environment |
| software language: | Mesa |

## 2.2.7 Known bugs

1. Extra variable in formal parameter list. eg StructChart created 13 times. Actually is counting the number of variables in the call list.
2. IString not recognized, just looks for string.
3. NIL handling
4. statsfile summary at context level
5. Some import designators missing in connections field.

# SECTION 3 - CONCLUSIONS AND FINDINGS

## 3.0 CONCLUSIONS AND FINDINGS

Section 3 starts with a short introduction then, a study of each of the DesignExtractor program outputs, then compares the findings to the theory of Section 1. This spurs a number of additional findings that have been categorized under the titles Observations, Advantages to the Maintenance programmer, Comparisons to the XDE environment capabilities, Enhancements that may be useful and a final Concluding paragraphs. .

This thesis project was considerably larger than expected. The information available for the Mesa parser was scant and difficult to understand. A study of the MESA parser was undertaken revealing that some of its elements could indeed have been used for the construction of the project code for this thesis. Also, MESA is a complex and unfamiliar language to me, and XDE (the development environment) was new. Given these constraints the design / programming phase of the project began in June of 1991 and completed in December of 1993. This represents approximately 2000 hours of work given time off for vacations and fatigue. This is approximately 1.25 manyears of work using a 40 hour week measure. Even with this investment in time and painful thought the program only represents a window into the opportunities presented. A more familiar programmer may have been able to cut this time down, and a member of the MESA compiler design team [Enhancements 1] could probably have made the implementation more universal. Nevertheless this project remains a major accomplishment in my life. Representing many lessons in endurance, thoughtfulness, frustration and joy as implementation barriers were faced and overcome.

# SECTION 3 - CONCLUSIONS AND FINDINGS

## 3.1 A study of the outputs

Section 2 details the top level project description. Some of the deeper levels of the program are discussed below. The Project Description of Section 2.2 was similar to the level of abstraction achieved at initial design time. The deeper levels were not understood during the initial proposal phase, to any significant level of appreciation. The abstraction detailed in the proposal was helpful but insufficient.

During the study of the outputs it may be beneficial to refer to section 2.2, especially section 2.2.4 where the overall system organization is described. A great deal of the program detail is left out, the purpose here is to look at the output formats and discuss these in relationship to the program implementation. Appendix A has been constructed to enable this method of study. The outputs in Appendix A are a parse of the DesignExtractor program itself. Thus thesis efficiency is maximized by discussing the outputs and the program in tandem. All the modules of the DesignExtractor program cannot fit into the database as it is rather large. Therefore a selection of 4 of the key program modules have been parsed, these are the most relevant to aid our discussions.

### 3.1.1 Study of SCGraph

DesignExtractorImpl kicks off the processing. A file or files are input to the user interface window and on the selection of the "Parse" command the program starts to run. The first thing the DesignExtractor program does is parse the code. The sequence is described in detail in section 2.2.1 and 2.2.2.5. The output listing of Appendix A.2.1 shows the SCGraph output format of the

# SECTION 3 - CONCLUSIONS AND FINDINGS

DesignExtractor tool working on itself. The rest of this section follows along with the SCGraph output in the Appendix.

The first thing you notice, on page 4 (of Appendix A.2.1), about the structure is the root procedures, marked SLC. These procedures are never called. As DesignExtractor is the control procedure for the DesignExtractor configuration file these are indeed the entry points to the program [ Advantages 1]. Each one of the procedures marked SLC can only be accessed from XDE controls such as the display or the keyboard. For example line 1 "GenTextDisplay" is accessed by the "Display" command, "ClearOutputFile" line 1355 is initiated by the "Clear" command and "Run" line 1381 is accessed by the "Parser" command on the User Interface display window. Each of these SLC (for Straight Line Code) procedures is a root procedure. Further SLC procedures are present, these deal with the startup/initialization portions of program setup.

Continuing with the parse (Run line 1381) track let's explore where it goes. Traverse down the track to "Parser" line 1457, it can be seen that we are still in DesignExtractor by examining the "Called from filename" field. We are now at procedure call level 4, RunIt line 1385 and ProcessFile line 1411 are the main control procedures. Many procedures are called from RunIt and ProcessFile, you may notice that some of these do not have "Declared in" fields. That is because these procedures have been imported through an interface. Notice the "Interface Name" field which shows you from where. If the program files associated with these interfaces where loaded in the parse session then the "Declared in Filename" field would then hold the appropriate called from information. As they are not loaded the display shows only the interface

# SECTION 3 - CONCLUSIONS AND FINDINGS

name as described. It is easy to see then from this which are the locally declared procedures, which are declared in other program modules and which are imported.

The next item of interest is at line1458 where we finally traverse an interface/program boundary. Because of the comprehensive parsing algorithms of DesignExtractor no reference to the interface files themselves are required. As the program looks for all procedures loaded in the parse run (all files) if the procedure is present then it is displayed, if not the interface name only is displayed. Also note that the list of missing interfaces at the beginning, pages 2 and 3, can be used as a quick reference to locate the missing interfaces.

Line 1458 "ParseInit" then is found in ParserImpl. Notice the nice banding here [Advantages 2]. Best seen if you look at page 8 and 9 together. It can be seen that the earlier procedures are predominantly found in DesignExtractor and the latter procedures (post line 1458) are in ParserImpl. The clarity of the banding of course breaks down if the program thrashes between program modules. The example in Appendix A.2.1 does not show this. However a previous run with the ProgramParserImpl showed that the program continually switched between DesignExtractor and ProgramParserImpl. To examine these areas could yield performance enhancement opportunity in a tightly time sensitive implementation [Advantages 3]. If calls to other program modules are frequent like this it may be better to hold that function locally to prevent having to bring in code from slower memory. Of course this is dependent on many factors, such as program size, memory size, cache capabilities etc.

# SECTION 3 - CONCLUSIONS AND FINDINGS

Continuing down the listing to line1490, Pass 2, 3,4 and 5 can be seen. These procedures are not loaded in the system. If the file ProgramParserImpl was loaded (referenced through the Parser interface) and another parser and display run done then this section would break out into all the procedure called by Pass2, 3, 4 and 5. This file was not loaded on purpose as it was very large, expanding the Appendix section too far.

If one wanted to do a study of ProgramParserImpl the procedures declared in that module would be clearly shown. A study of a single file could be done just as well as a system [Observations 8]. In this case the root procedures would be those that would be called by a parent if the parent were loaded. In our example above ProgramParserImpl would show root procedures Pass2, Pass3, Pass4 and Pass5 for example.

I have taken you down a single path, but dependent on the maintenance programmers interest he or she could examine any path of interest. He may also have elected not to load these files and perhaps taken a different route. There are endless possibilities. However, I believe it has been demonstrated, that following the trail of a complex program is much simplified by using DesignExtractor. For a multiple interface, multiple program module implementation examining the structure is greatly simplified. [ Advantages 4]. Indeed the architectural structure is detailed by the output listing of Appendix A.2.1.

If it has not become obvious already, it should be noted that we are looking at the dynamic structure of the program [Advantages 5]. The output listing of SCGraph shows all the calls in their order of call occurrence. This is much preferable to the normal methods of code reading,

# SECTION 3 - CONCLUSIONS AND FINDINGS

done in the static or declarative order. This is a major advantage to the maintenance programmer who is often overwhelmed with the volume of code. SCGraph is only a few pages long yet clearly shows the relationship between procedure and program modules. And, study of deeper levels can easily be done by adding additional program files and parsing them.

Call level of procedures is known from this graphical representation at all times. [Advantages 6]. It would be possible to spot program anomalies where excessive call levels were used with this chart. Scanning the "Lvl" column shows that the deepest level for this run appears to be 11, see line 660. Note the paper is not quite wide enough for long procedure names at the 11th call level, thus a little output distortion has occurred. A wider piece of paper would solve this. [Enhancements 9]. Recursive calls are not handled and none seem to be present. The DesignExtractor program would need a method to handle these if they were present [Enhancement 2]. Currently the program overflows the line endings spoiling the output format.

Repeated procedures on the same level are also suppressed. You may have noted while studying the SCGraph output that the line numbers are not always sequential. This is because a procedure call is repeated, maybe several times, at this point where line numbers are skipped. The output has been produced with repeats suppressed [ Advantage 8]. If the selection "Repeats" where selected at the User Interface prior to operating the "Display" command then the line numbers would come out sequentially and of course all the repetitive calls would show. The output however is much uglier and harder to understand.

# SECTION 3 - CONCLUSIONS AND FINDINGS

## 3.1.2 Comments

A program extraction of comments was not done in this thesis implementation. However there is a clear application of comments at this time. While traversing SCGraph as we did in the last section we may want to understand a little more about a procedure. The procedure name may not be terribly explicit, even though MESA is generous with naming conventions (up to 32 characters). This is a good application for the concept of HyperText.

If while examining the SCGraph listings a particular area needed more examination to understand, a HyperText box could be used to contain a description of the activities performed by a particular procedure. [Advantages 7, Enhancement 3]. A maintenance programmer could click the mouse on the procedure in question and if the correct references where held in the file then the comments immediately preceding the procedure could be displayed. Consider Figure 3.1.2.1 below:

# SECTION 3 - CONCLUSIONS AND FINDINGS

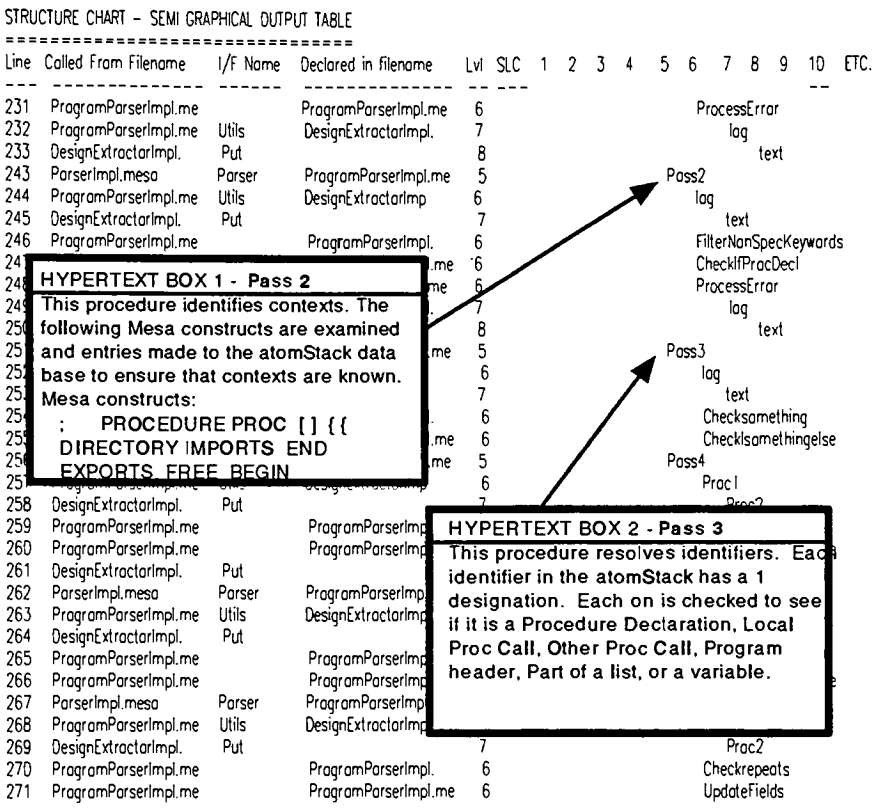



**Figure 3.1.2.1 - Example of comments with HyperText**

This kind of representation would be very useful. Provided the comments are in the code and correctly designated by delimiters then they could be extracted for use later on. The DesignExtractor does not provide this function. The Pass1 function of DesignExtractor uses the MESA parser which provides active tokens only, text comments are deleted. Therefore to provide this function an enhancement to the MESA parser would be required [ Enhancement 4]. The parser would be required to provide a link, in this case, between the procedure declaration and the associated comment field.

### 3.1.3 Study of SCTable

Appendix A.2.2 contains the output listing for SCTable. Section 2.1.3.2.2 contains the description of the output file for this table. The SCTable is really the Structure Chart output expected when the thesis began. However as the SCGraph structure was invented during the implementation phase SCTable becomes much less interesting.

The table is basically the same as that presented for SCGraph except the output is in a sequential tabular form. The output starts, see page 4, at the SLC level on the client side, then displays the called procedures on the right hand side, the subordinates. The display counts through each level of client showing subordinates for each until all levels have been displayed. Correlation of the table levels is 1:1 with the SCGraph table.

This table additionally shows repeat levels, however the number of repeats is not shown [Enhancement 5]. Repeat level function is described in Appendix F. Calls crossing file boundaries cannot be seen well here as the "fileName" field shows where the procedure is called. The example given above where Run called Runit called ProcessFile called Parse called Pass2 etc can be tracked through this table. However it is difficult and the graphic enhancement is lost. Remember the file boundary was after Parse, up to then we were in DesignExtractor then we transitioned to ParserImpl. Page 7 shows ProcessFile and page 8 Parse, examining the Subordinates - Filename field does indicate that transition has occurred but all the banding advantages cannot be seen well.

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.1.4 Study of DFDTable

This table was the ultimate expected output of the thesis. Appendix A.3.2 contains an output listing of DFDTable. Section 2.1.3.3.2 explains the meaning of each of the fields for this output format. This output can be used to create the graphical display that correlates to a Data Flow Diagram.

It should be noted here what the program is attempting to do. Earlier we studied SCGraph which showed the procedure call outputs in a sequential manner according to the dynamic structure of the program. That is the call structure rather than the declarative structure. The DFD table takes advantage of this same structuring and then adds the variable information. The order of processing then is the dynamic one. Note a semi graphic tool option that would be very useful for this display would be a HyperText option similar to that shown in the comments section 3.1.2 above. A pull down window could show the Local, Global, Imported, External variables appropriate to a particular procedure. This would be an excellent maintenance tool showing the structure of the program while allowing further examination by Hypertext windows. [Advantages 9, Enhancements 3]. See figure 3.1.4.1 below:

# SECTION 3 - CONCLUSIONS AND FINDINGS

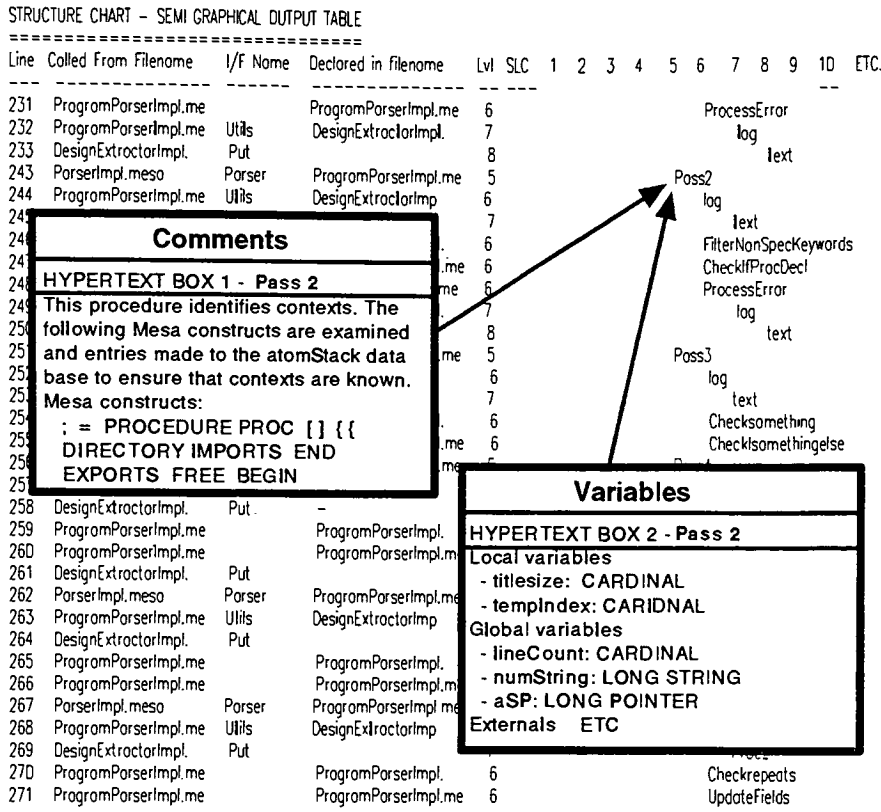



Figure 3.1.4.1 - Hypertext variable pullout

**Context Level:** The context level is indicated on page 2 of Appendix A.3.2, with the first line designated "C". The context level uses the node name of the control file (specified in the configuration file). This file is by default always the first file in. If another file is parsed first then it will be designated the context level. Then the program looks for external references, inputs and outputs, these are the context level constructs.

Files are located using the "External" notation [Enhancement 5]. Similarly for Outputs. The "External" designation helps determine the output conditions. Of course examination of the table

# SECTION 3 - CONCLUSIONS AND FINDINGS

shows that reads are equivalent to input and writes to output. The external files are designated in Pass5 of the parser. A depth search could be conducted for these, indeed this would have yielded statsfile which is shown in later procedure treatments.[Enhancement 8]. However in this case External references at the top level only have been addressed. The external reference "file" is shown.

The next nine inputs can really be seen as the SLC level calls from the Structure Chart Graph. Remember these procedures were not called at all and therefore they must be inputs [Advantages 1]. Specifying that these are keyboard input commands or display commands is not possible at this level, this level of abstraction is missing in automatic extraction (See Observations) . However the fact that these procedure calls are entry procedures is very valuable for the Maintenance Programmer in analyzing the purpose of these procedures [Advantages 1]. Also if at design time the implementor put appropriate comments in to say what were the input functions of these procedures, the Hypertext concept could be used to easily display this information upon request. See Figure 3.1.4.2 for the Context diagram.

# SECTION 3 - CONCLUSIONS AND FINDINGS

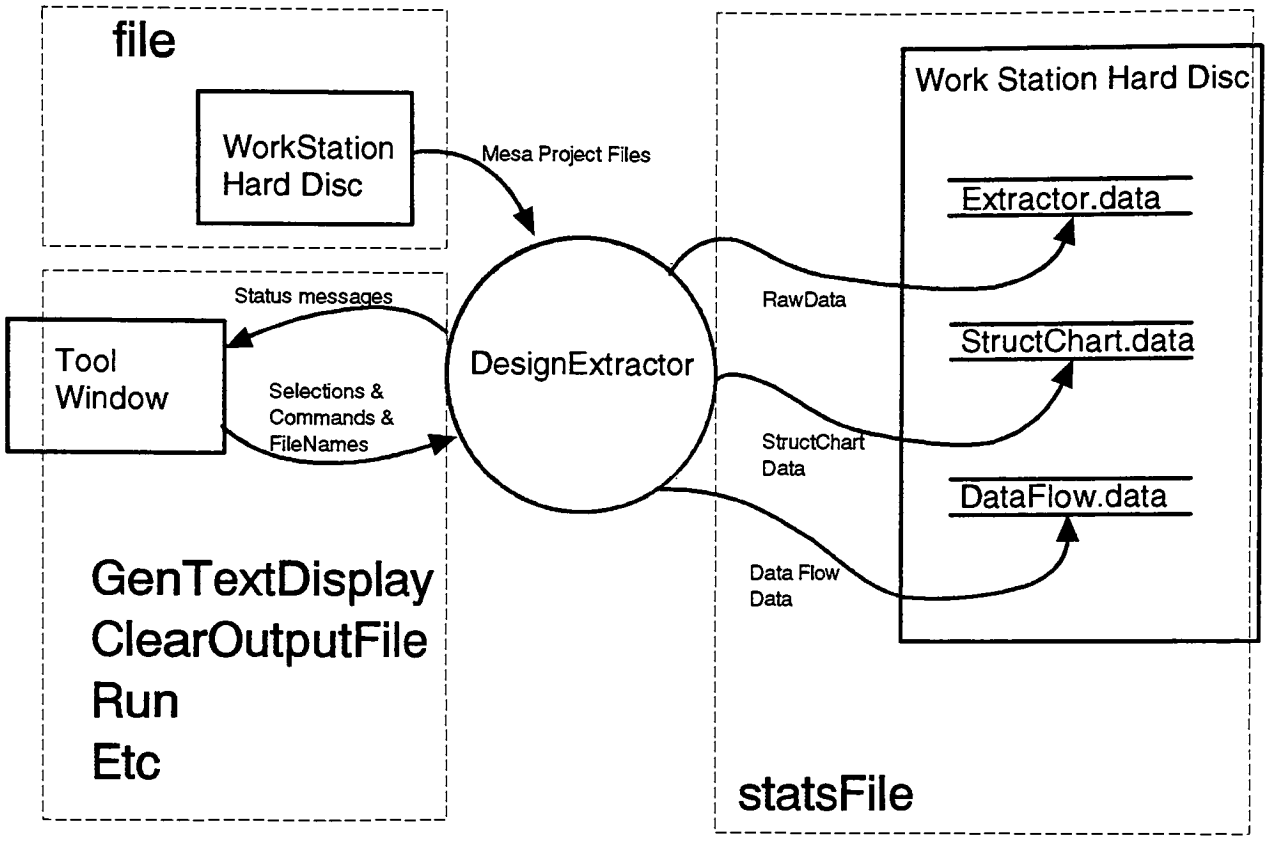



Figure 3.1.4.2 - Context level DataFlow Diagram

**Level 0 data flow diagram:** The next level of Data Flow Diagram is indicated by the SLC level and the level 1 procedures. This is because the level 1 procedures are the entry procedures as described earlier. Each one of the level 1 procedures starts something off and calls all the other procedures that follow. Globals, imported variables, external designations are all shown as appropriate. See Appendix E for a treatment of the variable scope designators. To avoid duplication the level 0 data flow diagram is not drawn below, however to make sure a deeper level function is addressed a level 1 diagram has been chosen in the next paragraphs..

# SECTION 3 - CONCLUSIONS AND FINDINGS

**Level 1 Data Flow Diagram:** Traversing down the listing in Appendix A.3.2 we get to the Procedure call for StructChart at level 4. The diagram below,figure 3.1.4.3, illustrates a level 1 example of the Data Flow chart that could be dawn here by examining the DFDTable output. To ensure that the correlation between the diagram and the DFDTable listing is understood, we will spend a little time on this one going through each element in detail.

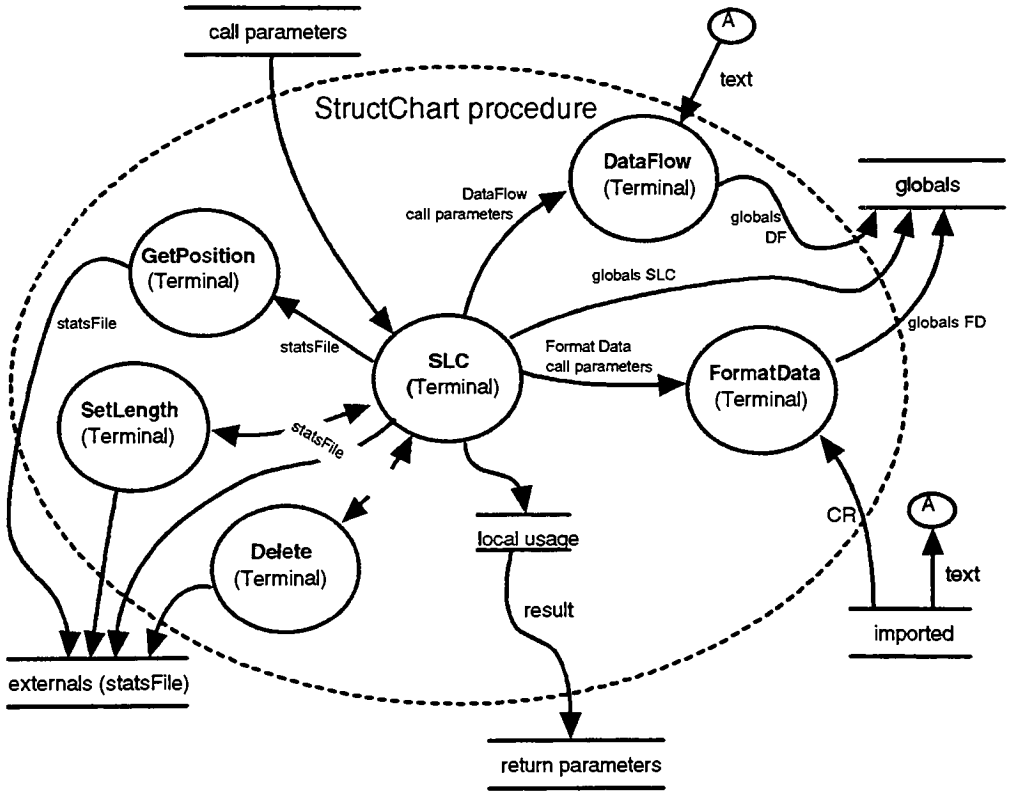



**Figure 3.1.4.3 - Level 1 Data Flow diagram example**

# SECTION 3 - CONCLUSIONS AND FINDINGS

**Node:**The procedure call level is at 4, the node is StructChart (Last line of page 8 Appendix A.3.2).

**Formal Parameters:** The first item bracket of variables we see are named FmlParam, these are the formal parameters for the StructChart procedure. eg created, aStackPtr, aIndexPtr through repeats. Note that these may not be the same variable names as the calling procedure. The name correlation of calling procedure to formal parameter is done in the "alias" column when a call is made. We will see that later on when we get to the FormatData call at the base of page 12.

**DataBase - Global:** The next four variables are Global, eg posIndex, fullCount and positionStackPtr and fullStackPtr. The connection fields hold the root name, which in this case is the same, and the Program name where the variables were declared, in this case all in the StructChartImpl file.

**DataBase - Imported:** The variable name is shown in the data name field and the interface name of the import is shown in the connections name field.

**DataBase - UseFmlCall:** This is a formal parameter passed in to StructChart the value of which is being read, i.e., aIndexPtr. Note the declaration field is not filled in because this item is a passed in parameter. Similarly a little lower down for SCGraph and SCTable.

# SECTION 3 - CONCLUSIONS AND FINDINGS

**DataBase - External:** Again this is an External type statsFile.

**IntCallP-UseFmlCall:** The designation internal call parameter signifies that this variable is in the field of a procedure call. The connection field name shows which procedure. As explained above the formal calls (UseFmlCal) of StructChart are passed on as call parameters to FormatData. A little later the same situation applies for the call to DataFlow. Note the **Alias field,** this field is used to indicate the formal parameter identifier used by the called procedure. For example with the FormaData procedure the calling variable aIndexPtr is instantiated to the called procedure as aIP. Therefore if you examine FormatData later on in the listings you will note that aIP is used throughout. [Enhancement 7].

**IntCallP - Global:** The call parameters for positionStackPtr, fullStackPtr, fullcount are all Global variables passed in to Display.

**IntCallP - External:** The call parameters for SetLength, GetPosition and delete are all designated the same. Statsfile is an External database that is used as a call parameter for each of these.

**DataBase - Local:** The variable result is declared within the procedure StructChart and is therefore designated as local.

**RetValue - Local:** The return value for StructChart is "result" is a local variable.

**Parsing downwards:** If you examine the Figure 3.1.4.3, the level 1 DataFlow diagram of StructChart there is one further detail that needs to be explained. Namely the dataflows emanating from the procedure bubbles FormatData and DataFlow. These values where found by examining the procedure listing in the dataflow table and realizing that Global calls are made in their respective procedure bodies also. This leads to the observation of [Enhancement 8].

### 3.1.5 A study of DFDvariable

This table is contained in Appendix A.3.1. and the detailed description of the field is contained in section 2.1.3.3.1. This table contains exactly the same details as in DFDTable, with a few extra fields. However the order of presentation is completely different. This table runs in declarative order. That is each file and procedure appears in the order they were parsed and placed into the atomStack database.

The relationship between the variable table and the other tables is indicative of the architecture of the DesignExtractor program itself. SCGraph shows the dynamic order, DFDvariable the static order but with all the variables delineated. The program combines these two tables to achieve the DFDTable final output of section 3.1.4. DesignExtractor uses SCGraph as the template then overlays the data in the variable stack that represents DFDVariable. This table is not output as a default. It is probably more interesting as a debug tool, although it does allow a study of the static flow if desired.

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.1.6 A study of Extractor.data

The output listing of Extractor.data is shown in Appendix A.1, a detailed explanation of the fields is explained in section 2.1.3.1. This output is very time consuming to produce. Every 100 lines takes about 10 seconds to produce. Therefore a large dump of say 10,000 entries will take some 1,000 seconds, that's over 15 minutes. A full database dump would take in the order 45 mins.

However Extractor.data is an excellent debugg tool. It is easy to see if the program is working correctly as each token is individually displayed as are the parsed relationship identifiers. The designations of variable types, module types, list types, scopes etc can all be examined as required. Also the relationship of calls to declarations are all held in the field of this table. This output is extremely powerful and represents the main source of information that could drive a graphics engine if created [Enhancement 13] This is not a particularly efficient way to make links and connections, however it is very readable and helpful when trying to trace problem areas.

## 3.2 Comparison to the theory

This section refers to the original propositions of the thesis and determines how well the final conclusions matched to the expectations. Four areas of interest are explored, first comparison to the IEEE standards detailed on page 16, second against the rules for Structure Chart creation detailed in Table 1.3.1 on page 61, third the rules for Data Flow diagram creation detailed in Table 1.2.1 on page 38 and briefly the output of Extractor.data as it compares to Table 1.3.4 on page 66.

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.2.1 The IEEE standard

Table 1.1.4 page 16 makes claims that Design information could be extracted automatically from code. The table is repeated below with the added explanation (Result) to substantiate the original claims.

| Design attribute | Extractable | Original Claim | Result |
|---|---|---|---|
| Identification<br>The name of the entity | Yes | Mesa code has distinct syntax for procedures, implementation modules and configuration files. These can be extractable automatically. | Yes, demonstrated as fully extractable |
| Type<br>A description of the kind of entity | Yes | Procedures and modules are clearly defined in Mesa. The interfaces will help us differentiate between subprograms, full modules, procedures, processes and data stores. | Yes, this structure is well understood from our output listings.. SCGraph clearly shows that procedures can be dynamically represented and the banding of modules is clear. |
| Purpose<br>A description of why the entity exists | Partial | The intent here is to study the content of the comments section of each module and interface. This should yield information that is relevant to the details of function, performance and special requirements. Perhaps a good Reverse Engineering methodology would include code commenting recommendations to facilitate automated identification of relevant fields. | Yes, this has been demonstrated by the discussions of section 3.1.2. However comments have not been extracted because the Parser algorithms remove them. See [Advantages7]. |
| Function<br>A statement of what the entity does | Partial | Similar methodology comment as for purpose. Code writers generally comment fairly well in program headers on the type of transformations performed. However, this is not usually well controlled or consistent in level of content. | Same comments as for Purpose above |
| Subordinates<br>The identification of all entities composing this entity | Yes | Parent / child relationships are well documented in Mesa source code. Extraction of this information enables the construction of structure charts. | Yes, fully demonstrated by SCGraph |
| Dependencies<br>A description of the relationships of this entity with other entities | Partial | Structure chart information can help with this. The intention is to extend this to Data Flow Diagramming. This involves the definition of a number of rules and paradigms. Thase will be detailed in section 1.3 where the theoretical base for this will be developed. Real-time control information will be ignored for this thesis. See section 1.1.5 were the scope is discussed. | Well, the dynamic relationship has been demonstrated however the level of abstraction that would help understand how modules (or why) they interact is not really very clear from any of the extraction. However good commenting and Hypertext facilities would greatly aid this. |

# SECTION 3 - CONCLUSIONS AND FINDINGS

| | | | |
|---|---|---|---|
| Interfaces<br>A description of how other entities interact with this entity | Yes | Mesa has strong interface structure which should help yield this kind of information. However, complexity will be introduced by the concurrency factor. Fork, Join and the semaphore operation will greatly complicate the control implications. Again, these interactions will be ignored as detailed in section 1.1.5 where scope is discussed. | Yes, the structural and procedural interactions are well demonstrated. |
| Resources<br>A description of the elements used by the entity that are external to the design | Yes | The prognosis at this time expects that much of this can be captured. The inputs and outputs from any particular module are identifiable. In some cases, partial understanding of the resource implication may also be available from the code declarations. CPU cycles are not relevant at this level. | Not really, from the perspective of which variable and types are used yes the extraction is good. However appreciation of system resources is not well seen by any of the extracted data. |
| Processing<br>A description of the rules used by the entity to achieve its function | Partial | This kind of information is often missing at the code level. The design determinations were made to structure the design. The information then rapidly gets out of date as the software maintenance complexities rise. | No, none of this information was readily extracted. |
| Data<br>A description of the data elements internal to the entity | Partial | Much of this information can be captured. The intent in this thesis is for the tools to produce 'data dictionaries' detailing this content. However it would not be expected that graphical representation of the data base structures and data handling algorithms could be gleaned at this stage. | Yes, the data items have been extracted and demonstrated. The data dictionaries were not constructed however it is clear from the level of implementation completed that this could be done fairly well. |

**Table 3.2.1.1 - Comparison table to IEEE standards**

# SECTION 3 - CONCLUSIONS AND FINDINGS

## 3.2.2 Rules for Structure Chart creation

Table 1.3.1 on page 66 makes claims that Design information could be extracted automatically from code. The table is repeated below with the added explanation (Result) to substantiate the original claims.

| Rule | Description | Result |
|---|---|---|
| 1 | **Procedure calls.** Map all procedure calls throughout the complete software system to obtain the full client / service relationship tree. This includes all calls embedded in procedures and within straight line code.<br>OR alternatively map all procedures below the dummy client of Rule 5. | Yes, demonstrated by SCGraph and SCTable in Appendix A |
| 2 | **Multiple calls.** If multiple calls to the same procedure are made from within a parent procedure or segment of straight line code then a separate Structure Chart node should be allocated for each call. | Yes, all procedure calls are mapped into the database. However to improve the display quality the Repeats function was added. When repeats are masked then one child typed only is displayed per parent. |
| 3 | **Scope of search.** The scope of the search for clients can be limited to the modules that have export access to the impl's | Yes, each modules is indicated with the client/subordinate relationship. Where interfaces are not loaded they are referenced. |
| 4 | **Imported Interfaces.** A subordinate should be created for each imported procedure each time it is called from a unique client. | Yes, same as for Rule 2. |
| 5 | **Dummy client's.** A dummy client can be inserted at any configuration file boundary to limit the scope of the SC. | Yes, The implementation chose to make configuration boundaries the lowest level for this implementation. Normaly the number of files and lines of code under a configuration are large and plenty for the tool to work on. Because interfaces define relationships the external looking calls are also seen in the interface field. |
| 6 | **Straight line code.** A portion of straight line code in an impl should be represented by a Structure Chart node. | Yes, each structure chart node is represented by a procedure call. Each line procedure call is shown as the client. Then moving to the DFDTable the SLC actions are shown as database operation within the body of the procedure. |
| 7 | **Control impl.** The control impl clause of the configuration file defines the starting point of code execution. | Yes. This has been used throughout as the first file in. Then the tool knows the root calls at that configuration boundary [Advantages 1] |

**Table 3.2.2.1 - Comparison of Structure Chart creation rules**

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.2.3 Comparison of data flow extraction to theory

Table 1.2.1 on Page 38 makes claims that Data Flow information could be extracted automatically from code. The table is repeated below with the added explanation (Result) to substantiate the original claims.

| Rule Set | Sub Rule | Description | Result |
|---|---|---|---|
| Rule 1 SC's | 1.1 | **SC Modules.** Modules activated by more than one superior are duplicated. | Yes, demonstrated in SCGraph,all procedure calls are mapped into the database. However to improve the display quality the Repeats function was added. When repeats are masked then one child typed only is displayed per parent. |
| Rule 2 - DFD to SC | 2.1 | **DFD / SC correlation.** DFD hierarchy levels coincide with each respective similar level of the SC. | Yes, demonstrated in DFDTable |
| Rule 3 - Data | 3.1 | **Data.** Static analysis should yield all constant, variable and formal parameters. | Yes, demonstrated in DFDTable |
| | 3.2 | **External outputs.** Static analysis should yield all candidates for output data stores. These outputs are toward file systems or standard I/O | Yes, demonstrated in DFDTable. However this topic is further discussed in [Enhancement 6] |
| | 3.3 | **External inputs.** Static analysis should yield all candidates for input data stores. These inputs originate from file systems or standard I/O | Yes, demonstrated in DFDTable. However this topic is further discussed in [Enhancement 6] |
| | 3.4 | **Internal outputs.** Static analysis should yield all candidates for output buffers. These outputs are toward other modules. | Yes, demonstrated in DFDTable where outputs to the procedures outside the current procedure are known. However grouping could be enhanced. |
| | 3.5 | **Internal inputs.** Static analysis should yield all candidates for input buffers. These inputs originate from other modules. | Yes, demonstrated in DFDTable where outputs to the procedures outside the current procedure are known. However grouping could be enhanced. |
| | 3.6 | **Non formal parameters.** Static analysis should yield all non formal parameters from each module and every activation of other modules within it. | Yes, demonstrated in DFDTable. Correlation of Call parameter to Formal parameter is done in the Alias field |
| Rule 4 - Nodes | 4.1 | **Terminal modules.** A terminal node in the SC is always represented as a transformation at the same level in the DFD. | Yes, all terminal and non terminal nodes are shown in DFDTable |

**Table continued on next page**

# SECTION 3 - CONCLUSIONS AND FINDINGS

Continued/-

| Rule Set | Sub Rule | Description | Result |
|---|---|---|---|
| | 4.2 | **Non terminal modules.** When represented in the DFD these modules are split into two representations.<br>***1 At the same level as the SC.*** The node includes the transformations within the body of the node plus all the subtree node transformations. i.e., Covers the scope of effect of the node.<br>***2 At the level below the SC.*** This is at the same level as the subtree nodes. The purpose of this inclusion is to show the transformation within the node body on the same level as the child nodes. | Partial, all the local uses of variables within the procedure bodies are shown. However the actual transformation information is only understood by examining the code. Also see the depth search comments [Enhancement 8]. |
| Rule 5 - Repositories | 5.1 | **Repositories.** Identify the data repositories at each level of the DFD. Completed bottoms up with the following subphases: | Partial, the local data repositories as well as Global are shown in DFDTable. However see the depth search comments [Enhancement 8]. |
| | 5.2 | **Replacement** of all formal parameters of candidate variables with the actual parameters. Thus indicating the actual parameters used at the level of the module being represented.<br>***a) Multiple calls.*** Several actual parameters could be produced if multiple calls to a procedure are made from a parent procedure.<br>***b) Constants.*** In every replacement, all actual parameters that are constants are suppressed.<br>***c) Expressions.*** In every replacement, all actual parameters that are expressions are replaced only by the operands.<br>***d) Global.*** If a variable is global to a called module then the variable name is replaced by the formal parameter of the module that called it. This process works its way up the SC until the highest level of its global effect is found. | Yes, demonstrated in DFDTable. Correlation of Call parameter to Formal parameter is done in the Alias field.<br>Yes, if repeats were unsuppressed then each call would show instantiation in the alias field.<br>No, all call parameters and formal parameter are shown<br>No, not implemented [Enhancement 15]<br>Yes, this is achieved by the instantiation to the formal values, however the bottom up view is only achieved by manual inspection. |

Table continued on next page

# SECTION 3 - CONCLUSIONS AND FINDINGS

Continued/-

| Rule Set | Sub Rule | Description | Result |
|---|---|---|---|
| | 5.3 | **Reduction** of the variable list to the actual variables that certainly contribute to the formation of repositories.<br>***a) Terminal modules.*** All the local variables declared in the module are removed from the list. These clearly only have a scope or effect internal to that transformation node.<br>***b) Calling module body.*** The variables that are purely local to the body are eliminated. Again these are local to the node and need not be shown.<br>***c) Non terminal module.*** The union of all variables in the body of the module plus the directly called child module variables, then subtract the variables declared in the called modules. | Yes, terminal modules do not show any variable data information<br>No, these are shown<br>No, union of variables is only done manually. Note Figure 3.1.4.3 shows a level 1 DFD where the manual process has been applied |
| | 5.4 | **Construction** of the repositories.<br>***a) Input repositories.*** If variables appear in the same input list at all times they can be replaced with a composite identifier. This allows us to construct the concept of 'data stores'.<br>***b) Output Repositories.*** If variables appear in the same output list at all times they can be replaced with a composite identifier. This allows us to construct the concept of 'data stores'. | No, algorithms not written to achieve this function See [Enhancement 15] |

**Table 3.2.3.1 - Data Flow theoretical rule comparison.**

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.2.4 Comparison of raw data extraction to theory

Table 1.3.4 on Page 66 claims that Data Flow information could be extracted. The table elements are repeated below with the added explanation (Result) to substantiate the claims.

| Column # | Field | Result |
|---|---|---|
| | Transformation nodes | |
| 1 | - Client Procedure | Yes completed |
| 2 | - Subordinate procedure | Yes completed |
| 3 | Terminal v non terminal | Yes completed |
| | Data | |
| 4 | - Data item identifier | Yes completed |
| 5 | - Interface imported from | Yes completed |
| 6 | - Data values used | Yes, stored in database but not displayed |
| 7 | - Type imports | Yes completed |
| 8 | Type | Yes completed |
| 9 | Internal v external | Yes completed |
| 10/11 | - Read v write | Yes completed |
| 12 | - Parameter call | Yes completed |
| 13 | - Return parameter | Yes completed |
| 14 | - Enable clause | Partial, [Enhancement 16] |
| 15,16,17 | - Scope local, global, ext | Yes completed |

**Table 3.2.4.1 - Comparison of data extraction with theory**

Missing Page

# SECTION 3 - CONCLUSIONS AND FINDINGS

from module to module occurs on an every other call basis for example then this may be a problem. This effect was seen during a parser of DesignExtractor and ProgramParserImpl where continual thrashing from module to module occurred. This effect may reduce software performance in a time critical system.

**[Advantages 4]. Program architecture.** The output of SCGraph Appendix A.2.1 is certainly one of the most useful formats produced. At a quick study the architectural structure of the system is apparent. Where the call structure moves from module to module, which are the entry procedures and the call levels are all easily identifiable. This output is the dynamic structure of the program rather than the declarative.

**[Advantages 5]. Dynamic representation.** The SCGraph and DFDTable represent the dynamic structure of the program rather than the declarative as mentioned above. This representation is lost when studying reams of code which jump from place to place with often very little positional cohesion.

**[Advantages 6]. Call level.** SCGraph additionally indicates the call level of procedures. This output can be used to understand whether call level problems are present in the program. At this time recursive calls are not handled. [Enhancement 2].

**[Advantages 7]. Comments and Hypertext.** As described in section 3.1.2 the SCGraph format could be an excellent tool for comments display. As the dynamic structure of the program is clear the next level of information is, what do the procedures do. If a Hypertext pull out were

Missing Page

# SECTION 3 - CONCLUSIONS AND FINDINGS

**[Advantages 11]. Variable scope designations.** The variable output tables of Appendix A.3 are very valuable in helping programmers understand the scope of the variable set used within any particular procedure within the implementation. Local, global and imported conditions are clear.

**[Advantages12]. Variable usage.** The variable output tables are constructed to show variable usage not declaration. Therefore the programmer can study the utility of a variable within a procedure. Local variables that are not used will not be listed. An enhancement could be to choose a variable and mark the SCGraph with where that variable is used. [Enhancements 8].

**[Advantages 13]. Locates user declared procedure types.** Mesa allows programmers to declare their own procedure types. The code gets complicated with these, however DesignExtractor captures these just as any other procedures.

**[Advantages 14]. Unused databases.** DFDTable shows occasions where Formal Parameters are passed into the procedure then not used in the body or used to make subsequent calls. These are candidates for reduction of complexity. Also some are simply passed through the procedure, i.e., formal call to internal procedure call. These too may be candidates for deletion if the internal call procedure does not used them either.

**[Advantages 15]. Variable read write.** Helps understand the variable usage.

**[Advantages 16]. Flexible file parsing.** The file input schemes are discussed in Section 2.1.2. This is a major feature of the program. A programmer can study one file at a time if they wish, then pull in other files as they see fit. Or simply parse the whole lot at once. This provides maximum flexibility to the file study session.

### 3.3.2 Observations

This section discusses some of the observations that occurred during implementation.

**[Observation 1]. Abstraction.** The level of abstraction achieved during this thesis implementation does not approach that of a real Data Flow diagram. When the Data Flow modules are implemented in code the structure tends to break down somewhat based upon the needs of the particular language of implementation. Mesa helps this if the programmer can arrange his modules and interfaces in a logically coherent way. However the traverse backwards up the Re Engineering curve of Figure 1.1.4 is obviously incomplete. The code clearly does not yield specific actions. The best that can be achieved at this time is an architectural representation of the implementation. This gets close to the boundary of the specification section of system engineering but is sadly lacking in content.

**[Observation 2]. Missing MESA keywords.** The PGS grammar of Appendix C shows all the parsing function keys used in the DesignExtractor implementation. I was surprised to find that certain keywords in the Mesa language were not included in the Pass 1 token parser supplied from the Mesa compiler. For example UNWIND, FREE, ABORTED etc were not included. A procedure was added into ProgramParserImpl to catch these keywords.

# SECTION 3 - CONCLUSIONS AND FINDINGS

**[Observation 3].** Workstation disc space. It was originally thought that files would have to be brought onto the workstation, parsed, then deleted to conserve space. However it proved that the space limitation was in the virtual memory that tended to overload with greater than 30,000 atomStack lines. Thus all parsed files can easily be held on the workstation.

**[Observation 4]. Configuration files and interfaces.** It was originally proposed that interface and configuration files should be parsed also. However during implementation it was discovered that the configuration files showed only a collection of files and a starter file. This was easy to understand and drove the rule that for a configuration run starting with the starter file was sufficient to produce the output. Similarly after compilation the program modules contained sufficient information to demonstrate that interface and procedure locations were adequately understood, with the exception of type definitions held at the interface file level.

**[Observation 5]. Import and export list.** Similarly to observation 4. Import and export lists are also easily available at the head of each program module. No further clarification is required.

**[Observations 6]. Turnaround time.** Debug of this program, in this form, was becoming increasingly difficult because of turnaround time. When a bug was found, a fix implemented, often the cycle included ViewPoint reboot, restart XDE through the loader, reinitialize DesignExtractor, Parser then Display maybe several options. All this could take some 20 minutes depending on the problem under investigation. This made debug extremely unproductive as the system size grew and problem subtlety increased.

**[Observations 7]. Documenting code.** Throughout this endeavor I noticed how much it seemed that code was understood at the time of writing then forgotten when revisited. Consequently comments were sparse. A good rigor would have been to have commenting rules that expressly encouraged commenting in a certain fashion. In this way revisiting code would be simplified and the HyperText opportunities previously discussed could be well utilized.

### 3.3.3 Enhancement opportunities

Throughout the implementation an eye was kept on opportunities for improvement. The maxim of build one then throw one away would work well here. Some of opportunities listed below fit into that category and some simply are opportunities for further study.

**[Enhancement 1]. Use of the Mesa Parser.** The Mesa parser is a six pass parser that implements many of the steps attempted for DesignExtractor. Pass 1 of the parser was used in the implementation. However the later levels could not be used because the structure quickly became more complicated than I was able to decipher. A poll of folk within Xerox yielded the result that not much expertise was around on the parser and nobody understood the data structures. However there would have been clear advantages of pursuing this route. Mesa BCD files include the lexical tokens to enable the debugger to locate the source locations from the executable code. If this structure could be cracked them much of the parse time of DesignExtractor could be reduced. The the challenge would be to produce display options only. This would greatly speed up processing time and make the tool universal to all parser Mesa code.

# SECTION 3 - CONCLUSIONS AND FINDINGS

**[Enhancement 2]. Recursive calls.** The DesignExtractor program was not designed to handle recursive calls. However that would not be that difficult. Simply to keep a track of loops in the dynamic call structure would quickly yield the required result.

**[Enhancement 3]. Hypertext on Structure Chart.** As explained in the advantages the SCGraph output display has excellent utility. If enhanced with a Hypertext function for both variables and comments this utility would be excellent. It could also be extended to include code segments. For example, imagine having the SCGraph available during debug sessions, programmers could hone in on areas where the problem lies, study the comments, examine the variable then current the code right there. These modifications of course represent considerable investments in time and resources.

**[Enhancement 4]. Mesa Parser to Parser comments.** To implement the requirements of Enhancement 1 then the Mesa parser initial pass would need to be modified.

**[Enhancement 5]. Count repeats.** The implementation of the DFD charts showed the level of repeat information for procedure calls. It would be a nice feature to show the count of how many times each procedure is called. This in not currently implemented.

**[Enhancements 6]. Specification of externals.** The method of locating external variables is currently hard coded into DesignExtractor. To enhance the utility a user interface should be constructed that allows the user to specify particular external variables. The program could suggest alternatives and allow the user to select preferences.

# SECTION 3 - CONCLUSIONS AND FINDINGS

**[Enhancement 7]. Variable alias tree.** A really neat feature would be to add a variable alias tree. Many variables are passed from procedure to procedure with a changed name at each call interface. A graphic that displays all the procedures that use that particular variable and its alias at the time would enhance productivity enormously.

**[Enhancement 8]. Depth search facility.** Similar to enhancement 7, variables need to be understood from a depth search perspective. For example in the context level all externals should appear whatever the low level procedure that called the external database. Similarly for each level, the lower level activities should be captured. The thesis display options did not lend themselves well to this. A study could be undertaken to discover a satisfactory display method.

**[Enhancement 9]. Wider paper.** SCGraph started to lose its formatting capability at about the 12th call level because of the 8.5x11 inch paper output constraint. A 14inch ouput option could be added as a feature to allow greater depths to be displayed.

**[Enhancement 10]. Language comment features.** Features should be built into programming languages to ensure better code commenting. This could be implemented with pull out windows that appear each time the language determines a comment is appropriate. This may be at file boundaries, procedure boundaries, decision boundaries. Now if these were indexed like a word processor the comments could be displayed in Hypertext modes or simply a narrative. This would enforce a rigor on the programmer, make maintenance programming much easier and provide a quality check of programmers work during implementation. This would aid code reviews, management inspection, and document why a certain code segment is implemented.

# SECTION 3 - CONCLUSIONS AND FINDINGS

**[Enhancement 11]. Single variable study.** The display of SCGraph could be enhanced to contain an asterisk to show that a particular selected variable is operated on. This is similar to the depth search concept. Then the maintenance programmer could look for operation on any item and study it as necessary. This could avoid some side effect problems that often happen when old code is changed by inexperienced staff.

**[Enhancement 12]. Performance.** This program was not constructed to produce optimum performance. If a new program was attempted I would strongly recommend studying the optimum performance capability. Maintenance programmers will not use slow processing algorithms with any enthusiasm. The ideas in Enhancement 1 could be the answer.

**[Enhancement 13]. Finer repeat selection**. SCGraph can either show all repeats or no repeats. A useful enhancement would be to allow just the repeats under one procedure to be shown.

**[Enhancement 14]. Finer parser selection**. As per 13 a good utility would be to allow a parse level to be pre specified. For example parser to level 5.

**[Enhancement 15]. Expression in call parameter.** DesignExtractor did not deal with this case. Enhancements would be required if this function proved to be a useful function.

# SECTION 3 - CONCLUSIONS AND FINDINGS

### 3.3.4 Concluding paragraphs

The DesignExtractor program has now been developed to a point where it shows the utility of data extraction from Mesa code. It is by no means comprehensive and the output formats are not conducive to immediate adoption of the tools by the MESA programming teams. However the SCGraph output is clearly a winner. This concept of Dynamic representation of the program architecture has been demonstrated, in a format that is concise and easy to read.

As discussed in the sections preceding, there are many advantages to the maintenance programmer if these tools, plus the noted enhancements, were adopted by a language and built into the parsing algorithms. Together with windowing utilities, that encouraged programmers to input comments at procedure call times, program decision points, strategic algorithm locations and other important points, there is good potential for reducing maintenance costs. Additionally the suggestions for Hypertext pullouts based on the SCGraph display format would enhance system readability, and if extended to code Hypertext, then debug sessions could become interactive with the system architectural structure. The key is a windowing environment where input is made easy and precise.

Although the work here represents good success with the concepts discussed, it is clear that the level of abstraction will not reach that of the original Data Flow designs. The code constrains the architecture to the character of the working system and does not display the reasons why design decision were made. If in the future, language designers build language systems that are intended to go both ways, forward and reverse engineering, then maintenance budgets could be reduced, whilst improving the reliability of system enhancement.

# SECTION 4 - BIBLIOGRAPHY


1 Ambr Ambrosio, Johanna: Finally coming into it's own: reverse engineering technology makes headway into IS organizations. Computer World Oct 22 1990. P25 & 26. BRIEF ARTICLE

2 ANSI Ansi standard flowchart

3 Anto Antonini, P; Benedusi, P; Cantone, G; Cimitile, A: Maintenance and reverse engineering: Low level design documentation production and improvement. IEEE SW, Feb 1987, P91-100.
Investigates systematic approaches to SW maintenance. Concentrating on code extraction techniques and possible graphical representation schemes.

4 Aoya Aoyama, Mildo; Mizomoto, Kazuyasu; Murakami, Noritoshi; Nagano, Hironobu; Old, Yoshihiro: Design specifications in Japan, Tree structure charts, IEEE SW March 1989, P31- 37
Describes the methodologies being used in Japan for design representation. The method is called Tree Structured Design, providing the normal elements of decomposition. Widely accepted in Japan.

5 Bach Bachman, Chalie: A CASE for Reverse Eng, Datamation July 1988. P49- 51.
This article investigates the need for CASE tools to address Reverse Engineering. Most current vendors have developed only Forward Engineerig capability. The author makes a strong argument for both features to make CASE tools successful and more widespread.

6 Bene Benedusi,P; Cimitile, A; De Carlini, U: Reverse Engineering methodology to reconstruct hierarchical data flow diagrams for SW maintenance. IEEE Comp Society conf on SW Maintenance, Miami Oct 1989. P180 - 189.
A detailed treatise of an experiment in Reverse Engineering. The article describes the extraction of design information from a current Pascal code system. Production of DFD's and SC is the main focus.

7 Bige Bigelow, James; Hypertext and CASE, IEEE SW March 1988, P23 - 27.
Describes the elements of a Hypertext database and how critical this is to CASE applications. Tackles large scale projects and the importance of coherent documentation and code.

8 Bigg Biggerstaff, Ted J: Design recovery for re-use and maintenance, IEEE Computer July 1989. P36 - 49.
A pictorial explanation of the elements of Reverse Engineering. Spanning the whole SW lifecycle from code to the highest levels of abstraction Develops the domain model and describes the content in detail.

# SECTION 4 - BIBLIOGRAPHY


9 Boeh Boehm, Barry W; TRW Defence Systems Group. A Spiral model of SW development and enhancement. May 88 IEEE 0018-9162. Pages 61 - 72.
This article explains the benefits of the Spiral model of SW development as opposed to the 'all at once' waterfall model.

10 Broo Brooks, Frederick P Jr; The Mythical Man Month - Essays on SW Engineering. Addison- Wesley, original copyright 1972.
A series of Essay describing the development problem set of programming in the large. A very entertaining but very pessimistic view of the difficulties involved. Categorises the effort as similar to a tar pit, where developers get stuck maybe producing a system finally but never reaching schedule and reliability goal expectations.

11 Cadr Cadre Technologies Inc:Teamwork/ SD user Guide Release 3.0.3 , Pages 1-2 to 3-21
Describes the design representation schemes for the CADRE Teamwork CASE tools.

12 Cadr Cadre Technologies Inc: Teamwork Ada User Guide Release 3.1, Pages1-2 to A-18
Describes the use of Ada design representation schemes in the CADRE teamwork CASE tools set.

13 Chad Chaddock, Roger; Shaw, Andrew; Vinea, Vladimir: Trends in REAL TIME CASE Accesssion #: 36082 ,Stanford Research Institute, June 1990, P1 - 25.
Discusses the long term and short term directions in CASE tool development. Plus Information Modeling and structured analysis.

14 Chik Chikofsky, Elliot J; Cross, James H; Reverse engineering and Design Recovery: A Taxonomy, IEEE SW Jan 1990, V7 P13- 17.
An attempt to clarify the terms used in the Reverse Engineering discipline. Uses hardware paradigms as comparisons and develops definitive use of SW terms.

15 Choi Choi, S.C; Scacchi, W: Extracting and re-structuring the design of large systems.Vol 7 P66 - 71.ISSN 0740 7459
A brief article on graphical representation of design constructs from a code base. Introduces concepts of structure, dynamic content, and behavioral content. Uses NuMil processing environment.

16 Coff Coffee, Peter: Analysis tools make source code more meaningful. PC Week Aug 20th 1990. P 38. BRIEF ARTICLE

# SECTION 4 - BIBLIOGRAPHY


17 Colb Colbrook, A; Smythe, C: Retrospective introduction of abstraction into SW, IEEE conf on SW Maint. Oct 1989. P 166 - 173.
Studies the result of design abstraction from code. Concerns itself with the problems of poorly structured data and suggests ways to retrospectively introduce abstract data types into existing systems.

18 Cuso Cusomano, Michael: Toshiba's Fuchu software factory: Strategy, technology and organization. Pages 1-70 Xerox Accession #: 34837. MIT 10 Oct 1987.
Describes the importance of modularity and factory operational styles for the production of SW. Japanese companies are using these techniques extensively to promote the principle of re-use.

19 DeMa DeMarco, Tom: Structured Analysis and System Specification. Yourden Inc. 1978. P1 - 164.
Builds on the Yourdon Constantine volume on structured design. Introduces more on analysis but the strength of this book is it's simple straightforward explanitory style making it an excellent companion to the afformentioned.

20 Enos Enos, Judith C.;Van Tilburg, RL: Software design - Tutorial # 5, Computer Feb 1981,Vol14, P61- 82.
Describes the history of Structured Design methodologies and computer science in general. Describes the methods and constraints on systems though the sixties, seventies and early eighties.

21 Fairley Fairley, R.E: Modern SW design techniques, Symposium of computer SW engineering, April 1976 P11 - 29.
Describes the modern software design techniques of Structured Design. The paper surveys some of the techniques and methodologies popular at the time of the article.

22 Falk Falk, Howard: CASE tools emerge to handle real time systems, Computer Design Jan 1988, P53 - 74.
Describes the advances in CASE tools in recent years with particular interest in graphical representation and real time elements.

23 Gilf Gilford, M: The Z-A of systems design, CASE magazine May 1988,V16 P89- 90.
A look at Reverse Engineering. The typical approach to engineering is starting from the requirements and working through to code. This article investigates the advantages of going the other way.

24 Hann Hanna, Mary A: Defining the 'R' words for automated maintenance: reverse eng,reusable pkg etc. SW magazine Aug 1990. P41 - 48.
A good discussion document on Reverse Engineering. Introduces Vendor capabilities and a list of their products.

# SECTION 4 - BIBLIOGRAPHY


25 IEEE IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std 830-1984, P 1- 24.
Describes the industry standard minimum needs for SW requirements specification documentation.

26 IEEE IEEE Recommended Practice for Software Design Description, ANSI/IEEE Std 1016-1987, P 1-15.
Describes the industry standard minimum set of documentation required to describe SW designs.

27 Mann Mannino, Phoebe; Stoddard, Bob; Suddith, Tammy: The McCabe Software Complexity Analysis As a Design and Test Tool. Texas Instruments Technical Journal, March/April 1990. P41- 53.
A thorough analysis of the McCabe Techniques.This is probably the most detailed article I have seen spanning complexity, test and battlemap representations.

28 Mart Martin, James: Making the CASE for true reverse engineering tools. PC Week April 23rd 1990. P52 BRIEF ARTICLE

29 Mart Martin, James: The beauty of re-engineering: continual enhancement. PC Week, April 23rd 1990. P 62 BRIEF ARTICLE

30 McCa McCabe, Thomas: Battle Map, Act show code structure and complexity, IEEE SW, May 1990, P62.
A quick reference guide to the concept of the McCabe BattleMap

31 McCa McCabe, Thomas J, Butler, Charler W: Design complexity measurement and testing. Communications of the ACM Dec 1989. P1415 1425.
Descibes the McCabe complexity metrics and their application. Shows the graphical representation schemes chosen by McCabe

32 McCa McCabe, Thomas J: The BattleMap Analysis Tool. McCabe Associates product brochures 1990.
Describes the BattleMap representation scheme for intermodule calls. The intent is to display on a single page as much structural information as possible.

33 McCl McClure, Carma: CASE is SW automation, Prentice Hall 1989, P183 -193
Describes the SW lifecycle and the current thinking of moving effort into the earlier stages. Goes on to show how CASE technology will further enhance the requirements, design activity and reduce coding and testing times.

# SECTION 4 - BIBLIOGRAPHY


34 O'Br O'Brien: Run time Reverse Engineering speeds SW troubleshooting. High performance systems, Nov 1989. P41- 48.
This article investigates the strengths of Reverse Engineering in the runtime environment. This is radically different to the usual case where only static parameters are considered. Aids in the analysis of interrupt operation and behavioral anomalies.

35 Pete Peters, L. J.; Trip, L.L.: SW Design representation schemes, Symposium of computer SW engineering, April 1976, P31- 55.
Discussions of modern design techniques and their advantages over the older techniques of flowcharting. Emphasis the importance of graphical and textural schemes.

36 Raj Rajlich, Vaclav: Vifor Transforms code skeletons to graphs, IEEE SW May 1990, P60
A quick introduction to Vifor, mentioned in detail elsewhere in the references.

37 Rajl Rajlich, Vaclav; Damskinos, Nicholas; Limos, Panagiotis; Silva, Joao: Visual support for programming in the large,IEEE SW March 1988. P92 -95
Describes visual representation schemes for program code and the interconnections between the modules for FORTRAN. Brings in a case study of a system Reverse Engineered and displayed using the VIFOR system. VIFOR = Visual Interactive FORtran.

38 Ryme Reymer, John R: Visual programming; is a picture worth 1000LOC. Seybold's office computing report Feb 1990.
The article talks to the uses of graphical programming. How to create program tailoring to the users need by easy step graphical interfaces. Blurs the boundary between users and professional programmers.

39 Schw Schwartz, Karen D: Analysis tool assists Navy in reverse engineering, Government Computer News Aug 20th 1990. P 50. BRIEF ARTICLE

40 Snee Sneed, Harry M; Jandrasics, Gabor: Inverse transformation of SW from code to specification. IEEE SW Mar 1988. P102 - 109.
This article works with a COCOL Reverse Engineering technique. The idea is to extract information from the code in two steps with the intent to build an entity relationship model of the system under review

41 Stev Stevens, W.P; Meyers, G. J; Constantine, L.L: Structured design, IBM Systems Journal May 1984, Vol 13, P216- 224.
Describes the notorary work of Constantine and Myers describing the principles and methods of Structured Design. A detailed article with much helpful instructional information on coupling, cohesion etc.

# SECTION 4 - BIBLIOGRAPHY


42 Taji Tajima, Denji; Matsubara, Tomoo; Inside the Japanese SW industry, IEEE Computer, March 1984, V17 P34-43.
Describes the activities in SW development within the Japanese industry.

43 Ulri Ulrich, William: Re- Engineering versus Reverse Engineering, SW Magazine Sept 1988. P9
A brief article discussing the differences between the two views.

44 Wild Wilde, Noman; Thebaut, Stephan: Maintenance assistant, Work in progress. Systems and SW Journal Jan 1989. P3-17.
Studies on the usefulness of SW tools for the maintainer.

45 Xero Xerox Corporation: Mesa Course, September 1988, 610E00230, P1.1 - 15.6

46 Xero Xerox Corporation: Mesa Language Manual, Oct 1988, 610E00171, P1.1 - 9.19.

47 Xero Xerox Corporation: XDE User's Guide, Oct 1988, 610E00201, P1.1 - 40.6.

48 Xero Xerox Corporation: Mesa Programmer's Manual, Sept 1985, P1.1 - 40.5.

49 Your Yourden, Edward: Techniques of Program Structure and Design. Prentice Hall 1975. P36 - 195.
Early writtings of one of the key pioneers of the structured design and programming methodologies. Detailed and extensive.

50 Your Yourdon, Edward; Constantine, Larry L: Structured Design. Yourdon Inc 1976. P17 - 82 & 221 - 391.
Classic volume on Structured Design. This could be considered a bible on the techniques of Top's Down Structured Design, Data Flow Diagramming, Cohesion, Coupling, Data Flow Diagramming. Two of the foremost authoritative authors of the 70's through to current day.

51 Zelk Zelkoaitz, Marcia; Yeh, Raymond; Hamlet, Richard; Gannon, John; Basili, Victor: Software Engineering practices in the US and Japan, IEEE Computer June 1984, V17 P57-66
Describes the activities in SW development within the Japanese industry.