INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

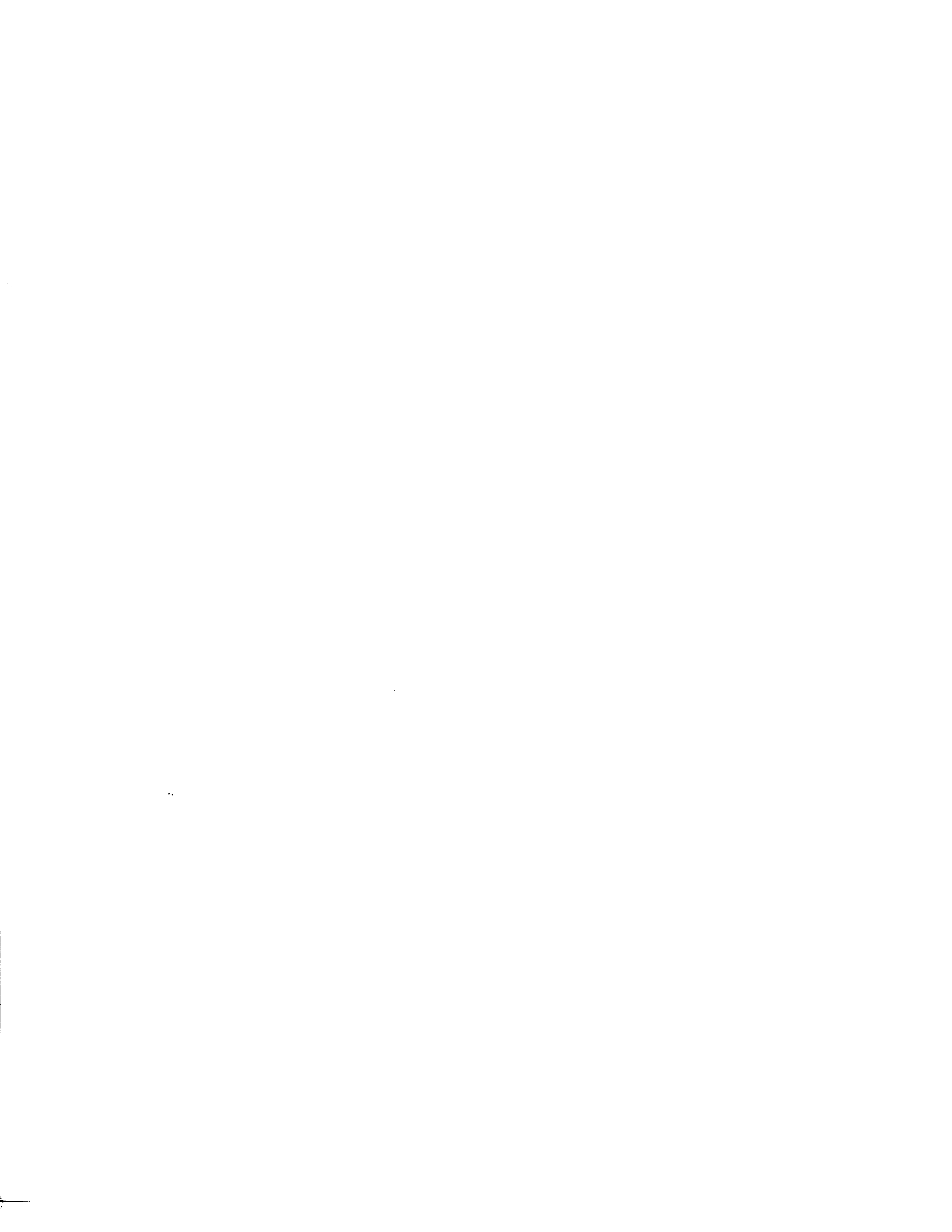
In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road. Ann Arbor. MI 48106-1346 USA
313/761-4700 800/521-0600



TOWARDS INTEROPERABILITY OF HETEROGENEOUS DISTRIBUTED SOFTWARE REPOSITORIES

by

Mindy Rosman Schreiber

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1995

UMI Number: 9530918

Copyright 1995 by
Schreiber, Mindy Rosman
All rights reserved.

UMI Microform 9530918
Copyright 1995, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© 1995

MINDY ROSMAN SCHREIBER

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

April 13, 1995

Date

Chair of Examining Committee

April 13, 1995

Date

Executive Officer

Professor Michael Barnett

Dr. Barry E. Jacobs

Professor Marsha Moroh

Professor Miriam Tausner

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

TOWARDS INTEROPERABILITY OF HETEROGENEOUS DISTRIBUTED SOFTWARE REPOSITORIES

by

Mindy Rosman Schreiber

Advisor: Professor Keith Harrow

This thesis deals with the problem of sharing software among heterogeneous distributed software repositories. Several issues must be resolved in order to successfully share software among many repositories. This thesis addresses two of these issues: a) transferring information among heterogeneous distributed repositories b) establishing a classification scheme for the purpose of storing and retrieving the software or collections of software in packages.

During the course of this research, it was determined that the use of a data independent data structure, such as MARC records, would allow the efficient transfer of corresponding information among several repositories. Tools were created to extract package information from repositories to be placed into MARC records, and also to manipulate and extract information from the MARC records, thereby completing the transfer of this information to other repositories.

Then a taxonomy for a dynamic classification scheme, capable of growing and evolving to satisfy the changing needs of the software community, was developed. The hierarchical data structure designed for this purpose is physically a network; however, it can be viewed logically as a forest of trees. Two objectives are accomplished with the use of such a data structure. First, there is no redundancy of nodes in the network; in addition, because the network can be traversed like a series of trees, there is no infinite cycling at any point during traversal.

Two methods of generating a network which can be traversed like a forest of trees are discussed in detail. The algorithms and data structures provided to describe the taxonomy contain the first method of generating this network. The second method, which is very flexible, is discussed in a section of its own. Performance issues regarding the new taxonomy are also discussed.

Further research directions regarding the automatic generation of USMARC records and the pros and cons of distributed vs. centralized software union cataloguing are discussed.

Acknowledgments

My sincerest thanks to Dr. Keith Harrow for his continuous and focused guidance, infinite patience and most valued support. The compassion you showed throughout is much appreciated.

I would also like to thank Dr. Barry Jacobs, senior scientist at NASA's Goddard Space Flight Center, for introducing me to research and for all his assistance while I was working on the HPCC project; Dr. Michael Barnett for his enlightening perspectives on software reuse; and Dr. Marsha Moroh and Dr. Miriam Tausner, for their invaluable comments and advice.

To all the members of my committee, thanks for the many hours spent reviewing this dissertation.

For their time and help in understanding how a library works, I would like to thank Howard Spivak Ph.D., Director of Academic Computing and Library Systems, and Professor Judith W. Wild, head of Technical Services at the Brooklyn College Library.

I would also like to thank my wonderful parents, Moshe and Miriam Rosman, for always being there for me; my sister Rivky Fishman and my best friend Rochelle Schilit for listening; and my children, Tova Rivka, Avraham Yaacov, Chany and Eliezer for giving me so much "nachas." You each brighten up every day.

And last, but by all means not least, my thanks to my husband Shragi, for ever encouraging me to persevere and for having more confidence in my abilities than I do myself. I could not have finished this without you.

Table of Contents

1. Introduction 1

1.1 Background 1

1.2 The Problem 4

1.3 Previous Research on Software Reuse 11

1.4 Purpose, Overview and Contributions of this Research 18

2. The HPCC Software Exchange Program 23

2.1 Introduction 23

2.2 Software Repositories 25

2.3 HPCC Software Exchange (HPCC-SE) Program 27

2.4 HPCC-SE Architectural Elements 30

2.5 HPCC Software Union Catalogue 33

2.6 Summary 35

3. An Example of a Logical Library System --

The HPCC Software Union Catalogue 36

3.1 Introduction 36

3.2 The Client 37

3.3 Searching for Software 39

3.4 Accessing the Software Repositories 40

3.5 Original and Copy Cataloguing 41

3.6 Summary 42

4. Semi-Automatic Generation of USMARC Records 43

4.1 Introduction 43

4.2 USMARC Records 47

4.3 Software Repositories 54

4.4 Generating USMARC Records 56

4.5 Summary 58

5. Indexing Schemes 59

5.1 Introduction 59

5.2 Problems 60

5.3 An Example 64

5.4 The Logical Forest 67

5.5 Inserting Package Information 75

5.6 Data Structures 77

5.7 Algorithms for Inserting New Packages 80

5.8 How the Classification Data Structure May Change in Time 88

5.9 Extracting Information from the Network 95

5.10 An Alternate Location List 100

5.11 Using the System -- An Example 108

5.12 Performance 111

5.13 Summary 113

6. Further Research **114**

6.1 Introduction 114

6.2 Automatic Generation of USMARC Records 115

6.3 Distributed vs. Centralized Software Union Cataloguing 116

6.4 Summary 121

Appendices **122**

Appendix 1: Netlib 122

Appendix 2: GAMS 127

Appendix 3: StatLib 132

Appendix 4: marclib 135

Appendix 5: marctool 143

References **154**

List of Figures

4.1.1 Sharing Software in a Distributed Information System 45

4.1.2 Sharing Software in a Distributed Information System using MARC records 46

4.2.1 MARC Delimiters, ASCII codes and Display 52

4.2.2 MARC Record in Transmission Format 52

4.2.3 MARC Record -- Decoded Contents 53

4.3.1 Example -- Transferring Fields between MARC Records 55

5.4.1 A Network Structure and Several Possible Traversals 72

5.4.2 Expansion of Node 73

5.4.3 A Node in the Network 74

5.8.1 Adding a Parent to a Node 94

5.10.1 A Network 102

5.10.2 Logical Tree Depicting the Network in Figure 5.10.1 102

5.10.3 Tabular View of Figure 5.10.1 and Figure 5.10.2 103

5.10.4 Adding a Child C to Node D in a Logical Tree 106

5.10.5 Adding a Child C to Node D in the Network--A Cycle 106

5.10.6 Adding a Child F to Node D in a Logical Tree 107

5.10.7 Adding a Child F to Node D in the Network--No Cycle 107

5.11.1 User View of Expanding Nodes 110

5.11.2 Browse Search Box after Retrieve Selected Package 110

1. Introduction

1.1 Background

Sharing or reuse of information and ideas is fundamental to the evolution of a technological society. For thousands of years reuse has helped in the development of ideas and influenced thought processes. Intellectual progress is "effected by developing and refining the ideas of others" [Bol89]. Knowledge has been and is conveyed via abstractions (e.g., alphabets and number systems). In as much as these abstractions are reused the knowledge they convey can be reused.

Studies (see, for example, [L86]) have shown that the abstractions used by a society have a strong influence on the development of the society. For instance, our phonetic alphabet has not only given us a method of reading, writing and classification but has "provided us with a conceptual framework for analysis and has restructured our perceptions of reality" [L86]. Phonetic alphabets are the most abstract alphabets in existence. They bestow upon their users a keen ability to abstract ideas and theorize. Logographic or pictographic alphabets, on the other hand, contain a unique visual sign per spoken word. According to the author of [L86], societies which use logographic alphabets tend to be more pragmatic.

Abstractions tend to simplify the conveyance of information. This holds true not only with alphabets (e.g., phonetic alphabets have between 22 and 40

symbols, whereas logographic alphabets have thousands of symbols), but in many other areas as well. In particular, we will take a quick look at how abstractions have influenced the field of software engineering.

Assembly language can be considered to be the first programming language. It made a programmer's life more productive by substituting English-like mnemonics for a series of binary digits. The success of Assembly language led to high-level programming languages (HLLs). HLLs contained constructs which created the illusion of executing one instruction, while in fact executing a number of instructions. Subroutines took abstraction one step further. A subroutine call would execute any number of statements. Libraries which came as part of some programming languages would supply a user with an assortment of input/output and mathematical subroutines. All these abstractions simplified programming, and they are all examples of software reuse. From English-like mnemonics to subroutines, each idea was used and continues to be reused.

Software reuse has been defined as "(1) The process of using pre-existing software during the development of implementing new software systems and components. (2) The results of the process in (1)" [Pe91]. IBM was encouraging customers to share know-how and programs with one another as early as December 1952. At first this idea was rejected by the mathematicians and engineers who were writing code. However, by 1955, recognition of the

benefits of sharing led to the formation of SHARE: an IBM users group preparing to install the IBM 704. Dozens of tested 704 programs were available within a year. The most compelling argument for cooperation was cost reduction [PJP91]. Furthermore, a prime reason for purchasing an IBM computer was SHARE.

Since then many more reasons for reuse have been recognized. In the following sections we will discuss various implementations of software reuse and the purpose and goal of this research.

This research brings a new level of abstraction to software reuse. Hopefully, interoperability among heterogeneous distributed reusable software libraries will be further developed and will contribute to the improvement of software engineering practices.

1.2 The Problem

This thesis deals with the issue of software reuse. Software reuse is a most promising method of reducing software development costs. At the same time, software reuse accelerates production speed and increases the reliability of the shared software. The examples of software related problems discussed below will illustrate the importance of software reuse.

Over the past 15 years, software costs have been skyrocketing in both commercial and government applications, with no relief in sight. For instance, the U.S. Department of Defense (DoD) spent over $3 billion on software in 1980 [Sk86]. By 1990 a full 10 percent of the DoD budget was spent on software. This amounted to $30 billion [Av90]. Hardware costs in 1980 for the DoD were $1.2 billion. These costs increased to $4.6 billion in 1990. When comparing these hardware cost increases with those of software, it is clear that software costs are increasing in proportions which, if uncontrolled, will limit the level and pace of technical development to which we have grown accustomed.

Software users are also troubled with delays, which in turn can lead to cost overruns. Two North American Aerospace Defense Command computer systems are examples of this problem [Av89]. The Space Defense Operations Center (SPADOC) modernization has been delayed by a minimum of 7-8 years; these delays will cost almost $150 million. Similarly, the Communications

Systems Segment Replacement (CSSR) program will cost an estimated $60 million over budget and is running approximately four years behind schedule.

One reason the cost of software has risen so markedly is that the requirements for new software systems are much more complex than ever before. Improved hardware configurations are a major cause of the demand for more complex software systems. Computing power, which has been increasing exponentially, can handle previously infeasible complex applications. Unfortunately, advances in software technology have not kept up with those made in hardware.

[Jo84] relates that a California study on commercial banking and insurance applications determined that approximately 75 percent of the functions coded were not unique. Another study mentioned in the same paper states that less than 30 percent of the source code in commercial applications deals directly with the actual problem. The remaining 70 percent of the code dealt with data validation, formatting reports and other tasks which are repeated in all types of applications. A 1983 study by the same author concluded that less than 15 percent of all code written is unique and specific to individual applications. The remaining 85 percent of all code is to some extent redundant. Japanese studies concluded that as much as 90 percent of the programs developed in any given year, especially in business applications, seemed to have been similar to other programs written in the past [Cu91].

Furthermore, economic analysis on the cost of software [B81] shows that development costs are an exponential function of the size of the software. Thus reducing new code by 50 percent will often cut costs by much more than that same amount. These facts lead us toward software reuse, in order to cut back on repeated code and thereby save on development costs.

As mentioned before, software reuse can also help alleviate the problems related to increased demands on the production of software. However, instituting the reuse of software is not a simple task. There are many psychological and technical reasons programmers don't find the idea of reusing code appealing [T87]. Among them are:

- There is a fear of losing job security when becoming dependent upon someone else's code.
- One gets satisfaction from writing the entire code alone.
- Sometimes it's easier to rewrite the code than to find it.
- Even if one wants to reuse code there may be uncertainty as to whether it is transportable.

The thought of searching for and using someone else's code may cause skeptical reusers to feel insecurity, displeasure, laziness and uncertainty.

Nevertheless, there are many reasons why these doubters should become software reusers. Included in these considerations are:

- A programmer can be much more productive when reusing code. This is obvious even when looking at simple math subroutine libraries which are part of many programming languages. Using built-in subroutines saves time needed for coding, testing and debugging. Increasing reuse can further reduce the coding effort.

- The more a piece of code has been reused, the less likely it is to be undependable. The code's reliability has been established through usage.

- When a programmer uses a set of components that the programmer has used in the past and trusts (e.g., components from a reusable software library or RSL), the resulting program's design tends to be more consistent and takes on a better form.

- When program designers use and reuse a set of components, they tend to understand them and how these elements will behave. This in turn makes a job more manageable.

- Specification and implementation of software components are likely to be completed much faster and more efficiently when standard components are used.

In short, reuse of well-designed, well-tested, well-documented reusable software components leads to more productivity, reliability, consistency, manageability and standardization [Ag88].

Specifically, software reuse has proven to be of crucial importance in situations where the failure of a software product would be disastrous. For instance, software written for highly sensitive machinery such as nuclear reactors must be totally reliable [M80]. By reusing code that has been tested and is known to be dependable major problems can be averted.

Studies show that a programmer is just 40% more productive when producing code in which 40% of the design and 75% of the code was reused [HM84]. However, the big payoff is in maintenance costs. Cost reductions of up to 90% have been reported in maintenance of software when software reuse had been used to develop new systems.

Most of the studies investigating the benefits of software reuse have been conducted about a decade ago. It is now considered an accepted fact that software reuse does pay. What remains to be accomplished is the creation of effective methods to facilitate the reuse of software.

In recent years many reusable software libraries (RSLs) or repositories have been developed. Typically, RSL users work in a single library. Within the confines of each library's limited user group the RSLs have proven to be very

effective. These RSLs are distributed throughout the country and the world, and they store their information in heterogeneous formats. We will discuss several RSLs in Section 2.2.

In a distributed information system, where information is being passed among various repositories, each repository has its own method of organizing, gathering and managing its vast collection of code. Each repository also has its own method of allowing a user to access the code, by employing some method which may or may not be user-friendly.

Potentially, quite a bit of the software can be shared among the many library systems. However, there is a serious problem: although the software is organized for the purpose of reuse within a particular repository's user-group, there is no method of transferring assets (software or related documents) among repositories.

NASA has solved a related problem with databases using the DAVID system. The DAVID (Distributed Access View Integrated Database) system was developed by Dr. Barry E. Jacobs at NASA's Goddard Space Flight Center. DAVID is a distributed heterogeneous database management system (DBMS) capable of internally storing and accessing all database types, thereby, allowing it to serve on many machines at NASA as a global data manager of distributed heterogeneous databases. Through an interface (a layer of software at each resident site) different data representations appear uniform, making

exchange of data both by query and by transaction possible, without physical conversion.

Interfaces between the heterogeneous distributed database management system DAVID and several commercially available relational database management systems have already been developed. These interfaces are operational at NASA's Goddard Space Flight Center, enabling scientists with diverse collections of astrophysical and oceanographic data to exchange information among themselves as they were never able to do before [MMH90,91].

Commercial efforts in sharing include: database systems such as ORACLE and R:BASE, which support standard SQL queries and encourage the sharing of data; and Object Linking and Embedding (OLE) which promotes the sharing of objects among various documents [PCM94].

Several factors make the time ripe for inter-library code sharing and communication. The recent explosive development of the Internet as a means of communication and distribution of information makes communication among repositories a possible task. Furthermore, programmers' reliance on object-oriented programming languages and separately compiled pieces of programs (e.g., Ada, C and new versions of Pascal) make the use of pretested, reliable pieces of code from a library a welcome method of writing code more quickly and dependably.

1.3 Previous Research on Software Reuse

Before we define precisely what we mean by software reuse, it might be helpful to look at a few success stories. Currently, most research in the field of software reuse concentrates on sharing assets within a particular organization. For example, the paper [LG84] discusses the success of reuse at the Raytheon Missile Systems Division of Information Processing Systems Organization. A study on programs written for Raytheon proved that 40 to 60 percent of the tested code was redundant. Often the benefits of reuse, however, are greater than simply the percentage of repeated code. Reuse can also save 60-80 percent of maintenance costs caused by individualized coding. A six year study on all business applications generated for Raytheon has led [LG84] to believe a 50 percent gain in productivity can be achieved with code reuse.

In [M80] and [M84], there is a discussion of a reuse method employed by the Toshiba Corporation to produce software which "cannot afford to fail." Rather than writing software, programmers "manufacture" it in Toshiba's software factory. Software reuse has increased productivity at the software factory by 14% per year. Furthermore, the reusable components have been well tested and are known to perform the tasks they were meant to achieve.

Despite these successes, there are some difficulties related to software reuse. They include:

- classifying software components
- retrieving software components
- making users understand the function of components
- connecting components to generate complete systems

The above considerations are all closely related. Good classification schemes lead to easy retrieval of components. Components whose functions are easily understood can be easily classified. Furthermore, well-classified components can be readily integrated into larger software systems.

Many classification schemes have been discussed in the software reuse literature.

- [Pr91] discusses the organization of a reusable software library using a faceted classification scheme. A faceted scheme classifies software using keywords called facets. Each facet has several terms to best describe the particular facet of the software you are searching for. For example, one facet is "entities." One of the entities a user may choose is "systems." Choosing terms, such as, "designs", "programs", "structures" etc. which pertain to "systems" will further reduce the search. Choosing the terms of each facet which most closely relate to the required software leads to the best hit(s) available in the system.

- [Br90] designed the Reusability-Oriented Parallel programming Environment (ROPE). ROPE uses a new classification scheme, called structured relational classification, which combines the advantages of hierarchical and keyword-based classification methods. ROPE's declarative hierarchy retrieves components whose attributes match those of a query. The more attributes specified in a ROPE query, the fewer components will be returned. ROPE functions within an environment called CODE (Computation-Oriented Display Environment) to help create parallel programs from reusable parts.

- The LaSSIE system is a frame-based knowledge base, which provides semantic retrieval of software [DBSB91]. The knowledge base is used as an index into the library of reusable parts. The object of the LaSSIE project was to overcome the problem of code invisibility in large software systems. This frame-based knowledge base automatically classifies software according to actions performed and provides a natural language interactive user interface.

- [MBK91] present a method of automatically generating a software library from documented software. Using the words of the documents to create a list of "lexical affinities" (words found close to each other in the document), an automatic indexing scheme is created to find the software that most closely parallels the user's requests.

Many articles advocating different methods of making software reuse work have been written. Among them are the following:

- [BCC92] outline an architecture for developing a component factory.
- [Bu87] couple a passive database with interactive design tools to make reuse an integral part of the software development process.
- [Pr91b] discusses an approach to organize and manage software for the purpose of reuse.
- [WJ90] survey current research in object oriented design, which the authors believe has a great potential for making software more reusable.

Research in integration of software components into larger systems has been developed. [N91] and [Ne84] semiautomatically construct software programs and systems using specifications and reusable software parts.

Software reuse may be implemented in a wide variety of ways. [K92] gives an overview of different types of software reuse. Some of the reuse methods mentioned are: using high-level languages, design and code scavenging, use of source code components [N91], software schemas, application generators [Cl88], very high-level languages [CLP84], transformational systems [Ch84], [Ne84], and software architectures.

Many repository systems or reusable software libraries (RSLs) are in existence today. These include the Army Reuse Center (ARC), formerly known as Reusable Ada Products for Information Systems Development (RAPID) [Ni91a,91b] [PB91] [S90], Reusable Ada Avionics Software Packages (RAASP) [We88], Reusable Software Library (RSL) [Ty86], Reusability Library Framework (RLF) and Asset Source for Software Engineering Technology (ASSET).

The research and repository systems mentioned above deal with various aspects of software reuse in a single geographic location or within a particular library. Although these systems have taken a big step toward more serious reuse of software, their advantages are somewhat limited. Sharing within any particular repository is a small solution to a much larger problem. The software crisis is a major problem, not so much because of individual companies duplicating their own code, but rather because there are vast amounts of code in the "code universe" that are duplicated many times over. Limiting sharing to within an individual library limits users to software available at a particular site. Inter-library access would open the doors to a global sharing environment and increase our ability to share reusable software components.

Several efforts have been initiated toward this direction.

The Reuse Library Interoperability Group-RIG [As91] is a working group that meets regularly to lay down requirements toward making interoperation among repositories possible. The working group has defined a Data Model Description to be used as a standard in reuse libraries. This data model contains what RIG believes to be a set of information that should accompany assets in a software library.

The STARS program [As91] [I90] is working toward the interoperation of the STARS Repository, the Unisys Ada Repository and the RAPID Center Library. A Common Data Model has been defined to enable asset information to be interchanged among these repositories.

The Defense Information Systems Agency-DISA [Me93] is also working toward the interoperability of three software repositories. Software components, which include any product of the software development life-cycle, must go through a certification process. Before being integrated into the library, software components are rated by applicability and software engineering criteria.

Both STARS and DISA are working toward interoperabilty among a small number of repositories. STARS' Common Data Model has placed restrictions on the users of the system by defining the type of data that must be stored with regard to their reusable components. By using a data-independent data structure capable of receiving and transmitting information from and to RSLs

it is our goal to make interoperability a possibility among any number of repositories. Furthermore, each reuse library need not change its structure in order to cooperate in this exchange.

The High Performance Computing and Communications (HPCC) Software Exchange is part of the Federal HPCC Program [R91]. In an attempt to vastly increase computational performance, the National Aeronautics and Space Administration (NASA) has been assigned the responsibility of coordinating the exchange of software among 11 Federal agencies. The goal of this project is to allow any number of repositories to join in this sharing effort [J92, 92a-d,93a-d].

1.4 Purpose, Overview and Contributions of this Research

Ideally, if a piece of software has already been created to do a certain task, another piece of software should never need to be created for that same purpose. Of course, in order to implement this extremely ambitious task, all software from every computer would have to be linked; therefore, this task can be considered an unreachable ideal. Restricting ourselves to software stored in RSLs, we could move closer to this ideal by joining all software repositories so that each could benefit from the others. In addition, for the system to work, programmers would have to want to reuse someone else's code and they would need to have access to all other systems. It is doubtful that even this limited goal of allowing access to software stored in RSLs will ever be achieved; however, we can hope to improve on our current level of reuse by linking many repositories together.

The initial direction of this thesis was to establish a method to help distributed heterogeneous software repositories share assets. We will discuss how this was accomplished with the use of a data independent data structure. A good portion of this research has been done in connection with the HPCC project. The HPCC project was a major research project which has made contributions in this area by several researchers.

Analyzing the contents of 12 RSLs has allowed us to determine similarities among the repositories. We determined that the use of data independent data

structures, such as MARC records, would allow the efficient transfer of corresponding information among several repositories.

In the course of our research, we designed tools to extract package information from various repositories to be placed into MARC records (see Appendices A1, A2 and A3). Then tools were created to enable the manipulation, extraction and output of information from MARC records (see Appendix A4), thereby completing the transfer of these assets to other repositories.

This research is the first to enhance (by appending new fields to the existing list of fields) and utilize USMARC as a data independent method of transferring assets from one repository to another (see Chapter 5). USMARC format is the standard for MARC records in the United States. However, MARC records are also used for cataloguing in the British Library, the National Library of Canada and other libraries wishing to participate in the OCLC (Online Computer Library Center) Online Union Catalog.

Using MARC records to transfer information among repositories only hints at the potential applications of these records. Using this work as a model, methods to promote sharing bibliographic information from books, journals and articles (plus sharing data from databases and sharing documents) can be developed. The procedures developed for this research can be seen as a generalization of the problem of transferring data among various types of

heterogeneous distributed systems, allowing many different kinds of data to be shared. More exotic uses may include sharing art and music. Virtually anything that exists on computer media can be shared with the use of MARC records.

The foundation of this thesis is based upon a logical library system (LLS) which enables users of various repositories to search, locate and extract software from other repositories. The LLS is the first attempt made to enable online sharing among an unspecified number of heterogeneous distributed repositories. It is also the first attempt to provide direct access from one repository to another. Links among repositories are established via tools from within a software union catalogue.

A software union catalogue should be a database management system which manages descriptions of items over many repositories. Just as a library's union catalogue contains descriptions of books over a set of libraries, a software union catalogue should contain descriptions of software packages (groups of related software modules) and programs over a set of software repositories. In this research we have created a union catalogue in which it is possible to update information from existing repositories and also possible to include new repositories.

A new classification scheme has been established for this project. Based on university course catalogues, keywords have been developed to classify each package contained in the system. We have identified several problems with this

scheme (see Section 5.2) which have led us toward the final objective of the research for this dissertation: designing a taxonomy for a dynamic classification scheme, using hierarchical data structures capable of growing and evolving to make software reuse effortless and convenient.

The data structure, which is physically a network, can be viewed logically as a forest of trees. The root of each logical tree will represent a general area in mathematics or statistics (or some other subject area). As the logical tree branches out, the nodes will represent increasingly more specific areas. Finally, at the leaf or bottom-level, pointers to related software will be stored. The network will function in a system that assists authors in writing useful abstracts. Retrieval algorithms will be provided to help researchers find software. We describe how this data structure is designed to evolve and help make software reuse a simple task rather than an onerous chore (see Chapter 5).

Although the RSLs which inspired this research do not currently contain object oriented software, this research complements software reuse using object oriented code. MARC records can be used to catalogue available object oriented software. Object oriented code must be retrievable in order to be used. The taxonomy described here can be used to index software in any RSL, including those which store object oriented code.

Ideas for further research on related topics are discussed in Chapter 6. Information regarding some software repositories and tools created during the process of this research are included in the Appendices.

2. The HPCC Software Exchange Program

2.1 Introduction

As part of the Federal High Performance Computing and Communications (HPCC) Program, the HPCC-Software Exchange (HPCC-SE) Program was initiated. This program was to be completed over a period of six years, starting in 1991, under the direction of Dr. Barry E. Jacobs, a senior research scientist at NASA's Goddard Space Flight Center. The HPCC-SE Program was organized in three stages [J92]:

- The HPCC-SE Experiment System is currently in operation and available on Internet.
- The HPCC-SE Prototype System was to be developed over the next two years (1993-4).
- The HPCC-SE Operational System was to be developed over the following two years (1995-6).

The HPCC-SE Experiment System helped discover some of the problems associated with sharing software in a distributed system. It is currently available only to a limited audience. The HPCC-SE Prototype System is a more extensive version, which was designed after studying the initial use of the Experiment System and building upon that version. It is capable of serving a much wider audience. However, due to budget cuts, funding for this project

ended at the end of 1993. Therefore, the HPCC-SE Operational System, which was to build upon the experience obtained from the previous two stages to create the initial Operational System, is not being developed at this time.

The goal of the HPCC-SE Program was to make inter-library exchange and reuse of software a reality. The exchange was to be done over major computer networks. As noted above, the experimental stage is currently accessible over Internet.

This chapter describes the repositories which have joined the HPCC-SE Experiment System. It gives an overview of the entire software exchange program and describes the underlying architectural elements. Finally, it states the role of the software union catalogue, whose structure and development is part of the HPCC project and a major part of this research.

2.2 Software Repositories

Four software repositories have been linked into the HPCC SE Program. They are the National Institute of Standards and Technology's (NIST) Computing and Applied Math Lab Software, which is indexed in the Guide to Available Mathematical Software (GAMS), Netlib, ELib and StatLib. Each of these repositories contains mathematical and statistical software.

The software indexed in GAMS (which we will refer to as GAMS software) is organized into "packages," each of which is a collection of programs or subprograms on a related topic. GAMS software packages are organized by a tree-structured, problem-oriented classification scheme [Bo90], [Bo89], [Bo85].

GAMS information is stored in a database, implemented on the Relational Information Management System (RIM). Both on-line and off-line guides are available to users of the system.

Netlib, ELib and StatLib store their software as text files. Users can obtain software from these repositories via e-mail. A user first requests a full index for the repository. The index includes library names (these libraries are similar to GAMS packages in that each library is a collection of related software) with descriptions of the type of software in the library. When a library which may be of interest is located, the user can then request a detailed index of that

library, followed by a request for the particular software which is required. See Appendices 1, 2 and 3 for further details on these repositories.

2.3 HPCC Software Exchange (HPCC-SE) Program

In order to implement the HPCC Software Exchange Program we must have a method of mapping all the different repositories which are to share software into a common data format. The underlying data structure for this sharing experiment is a "book." Just as a book in a library contains a title page, preface, table of contents, chapters and indices so do the books that are used in this system [J92].

For example, the logical library contains various RSLs, such as Netlib, ELib and StatLib. Each of these RSLs contains many packages and even more modules. We can consider all related data pertaining to each repository as a book in a large library. Libraries have directories (containing locations of other libraries) and catalogues (containing information on all the books in this library and all the other libraries) to help researchers find required books. Similarly, the different repository directories and catalogues will help potential users of the logical library find the software for which they are searching. Just as library directories and catalogues can be considered books in libraries, each of these repository directories and catalogues is also considered to be a book in the logical library.

A book is a collection of text files, databases and other books. Tools have been created to access, browse and retrieve elements of interest from the books.

The title page of each book gives the title, author, publisher and date of publication of the book. The preface gives an overview of the book. The table of contents is a relational database which provides a brief description of each chapter of the book. Chapters contain specific information regarding each entry in the book.

Each book will also have a set of index tools. Using the author (SUCat's author index allows searches on package authors; whereas, an author index from an RSL would allow searches on module authors), text (which compares a text input string with the abstracts to find the packages which are needed), subject (which returns the packages whose subject classification matches the one chosen by the user) or title index (which returns the package(s) with the same title as the one chosen by the user) one can build sets of entries which are of interest to the user. When the sets are built, extraction tools may be used to get copies of the required data from the sets. E-mail is one example of an extraction tool. It can be used to send a copy of a file to a user. Anonymous ftp is another method of transferring files to users.

The basic idea is to view all of the Internet as part of one large "logical library." For the purpose of sharing software, we can consider each software repository and each repository directory as separate items on the shelves of a large library.

Continuing the analogy, we note that researchers check catalogues and directories to find appropriate books for their research. The books they find will then supply the researchers with the required facts. In a logical library, we would look into a repository directory to find the appropriate repository for a certain problem. To get the actual solution to the problem we would then use the information supplied by the directory to look into the desired repository.

Thus, using methods similar to those used for the purpose of finding and sharing books in a consortium of libraries, the HPCC Software Exchange Program will help users of the system find and retrieve available software.

2.4 HPCC-SE Architectural Elements

Before describing the architectural elements used in the system we must first define some key concepts.

- The user's incentive for searching for software is the problem.
- A self-contained piece of software is a program.
- A collection of modules is a package. This is analogous to a journal in a library which is a collection of articles.
- A smaller piece of software and/or documentation within a package is called a module. This is analogous to a journal article.

In order to help a researcher store, search for and locate software, certain architectural elements must be developed.

- Repository Management assists a researcher in identifying and locating software programs, packages and modules within a specific software repository. The software repository can be at a single site or distributed over multiple sites. This is analogous to a library system that manages books, journals and articles; the library can store its resources in one place or in several places.
- Submission Standards and Procedures inform a researcher how items were entered into a repository. This information contains one or more

paragraphs describing the software repository. It then describes the exact format of the included information. This helps the researcher extract software in as useful a form as possible. This is analogous to a set of standards for publishing books or submitting articles to journals.

- Repository Directories help researchers identify and locate software repositories that are relevant to their research. This is analogous to a database which has descriptions of the contents of many research libraries in the United States.

- Uniform Access provides researchers with a uniform method of accessing heterogeneous distributed RSLs. A researcher can use a common interface in searching heterogeneous repositories such as Netlib, GAMS, StatLib and ELib rather than having to learn each repository's interface.

- Software Union Cataloguing assists researchers in locating software packages and programs over a set of heterogeneous distributed repositories. The Software Union Catalogue is analogous to a union catalogue for a consortium of libraries; for each published book or journal, this catalogue lists all the libraries which have copies.

- Module Cross Indexing allows users to identify sets of software modules over a collection of packages. Once the modules are located, the researcher may use a local software repository or the researcher may use

the Union Catalogue and then a Repository Directory to extract the actual software.

There are numerous ways of finding software in the HPCC Logical Library. Section 3.3 deals with searching for software using the HPCC Software Union Catalogue.

2.5 HPCC Software Union Catalogue

As previously mentioned, the Software Union Catalogue (SUCat) allows users to locate software packages over a set of heterogeneous distributed repositories. The SUCat book's components include, as with all other books, a title page, preface, table of contents, a file called "How to Use this Book," indices and chapters. There are four indices: text, author, title and subject. The chapters of this book include the following components for each package of every repository: USMARC records, Abstracts, RIG Yellow Pages, Description of Special Applications or Subject Classification, Locations of Software, HPCC Directory of Software Repository Entry and Alls.

- USMARC records are a data independent method of storing information about the different packages in each repository. USMARC records will be discussed in detail in Chapter 4.

- Abstracts contain a description of the type of software that is included in this package. The abstracts are also essential for the text index. This indexing tool compares the text in the abstracts with the user requests to build a set of useful packages.

- RIG Yellow Pages follows the RIG model of describing and cataloguing each package. A tool reads the MARC record of the appropriate package and outputs the RIG model of that record.

- Description of Special Applications or Subject Classification is a keyword classification scheme. Each package in the SUCat is classified by one or a group of keywords. This classification scheme is used for the subject index.

- Locations of Software contains the id of the repository in which each package is located. Once this information is found a user can directly access the repository by using the next chapter of the book.

- HPCC Directory of Software Repository Entry gives a user direct access to a given repository.

- Alls gives the user access to all the aforementioned chapters at once.

2.6 Summary

In this chapter we discussed the goals of the HPCC Software Exchange Program. Then a description of software repositories linked to the HPCC-SE Program was given, followed by a description of the architectural elements needed for the Software Exchange Program. Finally, the components of the Software Union Catalogue (SUCat) were described.

This chapter described some of the repositories that we have dealt with in the course of this research. Part of the research for this dissertation deals with the movement of software from one repository to another using a data independent data structure. This chapter outlined a software exchange program. In Chapter 4 we will discuss the underlying data structure, the MARC record, which makes software exchange possible. Before our discussion on MARC records, however, we would like to take a brief look, in the following chapter, at how users can access software from different repositories using the SUCat.

3. An Example of a Logical Library System-The HPCC Software Union Catalogue

3.1 Introduction

A software union catalogue should contain a method of retrieving packages (or package information) from various repositories. It must also have a user interface to help the user obtain this required information.

This chapter describes one example of how the user interface of the system, called the client, can be structured. It gives a description of how users can access software repositories. Finally, this chapter discusses original and copy cataloguing with MARC records.

This chapter gives a user's view of a union catalogue, which allows sharing among various repositories. The underlying data structure and tools which make sharing possible in the Logical Library System are discussed in Chapter 4.

3.2 The Client

A client is a program that interacts with another computer program, called a server, to obtain services. The client supplies users with a graphical user interface to help them obtain the information they are seeking. Once a book is made known to the users of the system, it is made available through interactive book clients. The client program for the HPCC SUCat book has system level access to the book server, thereby providing users with requested information from the client menu.

Below is a description of the prototype Book Client used in the HPCC SUCat's client-server system. We will describe it via a screen-by-screen walk-through. The first screen to appear shows the following headings:

- Title: When a user clicks on "Title" a text file with the title page of the current book appears. The title page provides the title, author, publisher, date, etc. of the book.

- Preface: When a user clicks on "Preface" a text file with the preface of the current book appears on the screen.

- Pages: When a user clicks on "Pages" a hyper-table containing a summary of each page (or package) appears. For each package, this table contains the package's name, a brief description and its entry-id. The components

of "Pages" appear as icons on the left hand side of the hyper-table browsing tool. The components are:

summary - produces, as the default component, a brief summary of the selected package.

descr - provides a more detailed description of a selected package. A user choosing "descr" has the option of choosing (on the top of the screen) tools-Browsing tools. If the user makes this selection, a little tool box appears allowing the user to choose from 1: Submission Form (default) Browsing Tool, 2: Rig Yellow Page Browsing Tool, and 3: Readable Marc Browsing Tool. These three choices give the user the ability to view information on the selected package in different formats.

locations - provides direct access into a selected repository, thereby providing access to actual code.

- Indexes: When a user clicks on "Indexes" a list of four available indexes appear in a new window. These indexes are:

abstr-txt - which uses text to retrieve relevant packages.

title - uses program/package title to find relevant package(s).

author - uses author name to find relevant package(s).

subject - uses subject classification to find relevant package(s).

3.3 Searching for Software

When conducting a search in SUCat one finds the RSL which contains the information being sought. Direct access is provided to enter the RSL. Then once inside the RSL a similar search must be repeated in order to get to the actual software.

This method of retrieving software requires redundant searches (once inside SUCat and then repeated inside the RSL) and can be improved. Direct access into the package in the chosen software repository should be provided from within SUCat. Then the user can just examine the modules within the package and pick the one(s) of interest. In Chapter 5 we will discuss how the method proposed in this thesis will lead to finding software and retrieving it with a single search.

3.4 Accessing the Software Repositories

It is quite simple to access the software repositories that are hooked into the system from the Software Union Catalogue. Once a repository book is published, that repository's information is made available to the SUCat book.

After determining the package(s) a user is looking for, the system provides a hyper-table containing the names of the repositories which contain these packages. Choosing a particular repository name accesses the repository.

3.5 Original and Copy Cataloguing

Each package in the SUCat book must have a corresponding MARC record in order for the package to be in the Software Union Catalogue. MARC records, which are data independent records capable of storing package information from any RSL, will be discussed in detail in Chapter 4.

Original cataloguing is done when a MARC record must be created for a package. See Appendix A4 for tools available to generate the original MARC records.

One of the benefits of using MARC records to catalogue packages in any system which functions as a union catalogue of heterogeneous distributed RSLs is the ease of including an already catalogued package into another repository, or for that matter updating the union catalogue. For example, when SUCat determines that a new package has been included in a member repository, it copies the MARC record of that package from the repository. SUCat is automatically updated when the new package's MARC records are included in SUCat. Copy cataloguing is the function of copying an existing MARC record from one book to update another.

3.6 Summary

This chapter described the interface, called the client, which helps users locate software packages and programs over a set of software repositories.

An explanation was given of how different software repositories are made available to the HPCC SUCat. Finally the benefits of copy cataloguing were discussed.

The following chapter deals with semi-automatically generating MARC records.

4. Semi-Automatic Generation of USMARC Records

4.1 Introduction

The USMARC format was developed to enable the Library of Congress to transmit its catalogue records to other libraries and agencies. It provides a data independent method of transferring information from one system to another [By91]. In other words, regardless of how the information was stored in individual systems, the data could be transferred to USMARC and then shipped to and read by other systems.

It is precisely this attribute, data independence, that makes the USMARC format a most effective tool to transfer package information from one RSL to another.

In a distributed information system, where information is passed directly among various RSLs, an interface is needed for each pair of different repositories. If there are n repositories, we need $n(n-1)$ or roughly n^2 interface modules (see Figure 4.1.1). Furthermore, each new RSL would need $2n$ connections to the existing system of n repositories.

One way to reduce the number of modules is to have a common structure through which all data will pass. One fundamental structure into which all information could be transferred is the USMARC format. By passing all information into the MARC format when exporting package information from

an RSL, and out of the MARC format when importing package information into an RSL, we reduce the number of required modules to *2n* (see Figure 4.1.2).

Thus, each of the *n* RSLs, rather than needing a separate transformation to export and import asset information to each of the *n-1* other RSLs, needs just one transformation to export to a MARC record and another to import from MARC. Ordinarily, each time a new RSL joins the system of *n* repositories, *n* new interfaces must be created for the new RSL. In addition, for each of the previous member RSLs, someone will be inconvenienced by having to write a new interface to include the new member. This would probably lead to reluctance from both new and old members to having new repositories join the system.

As the number of RSLs involved increases, the savings become more dramatic. For example, if 5 RSLs were sharing package information, with standard interfaces we would require 20 interfaces, as opposed to the 10 interfaces required when using MARC as a transfer mechanism. However, if 50 RSLs were sharing package data, using standard interfaces we would require 50x49 or 2,450 interfaces, as opposed to 50x2 or 100 interfaces needed when using MARC. As the size of *n* increases, the difference between *2n* and n^2 becomes much greater. This would make the potential savings even more significant.

SHARING SOFTWARE IN A DISTRIBUTED INFORMATION SYSTEM

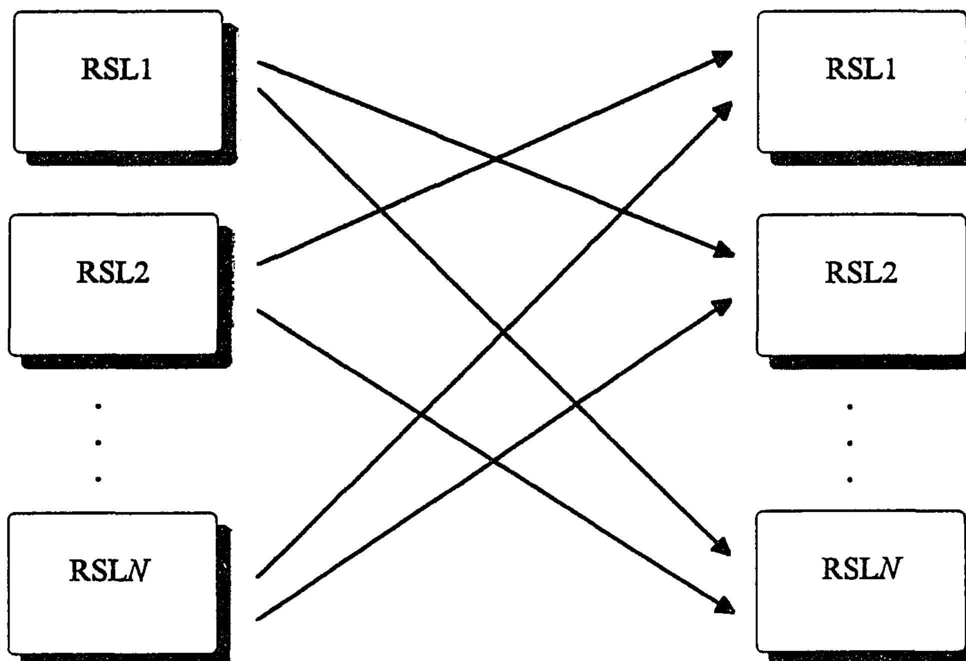


Figure 4.1.1

SHARING SOFTWARE IN A DISTRIBUTED INFORMATION SYSTEM
USING MARC RECORDS

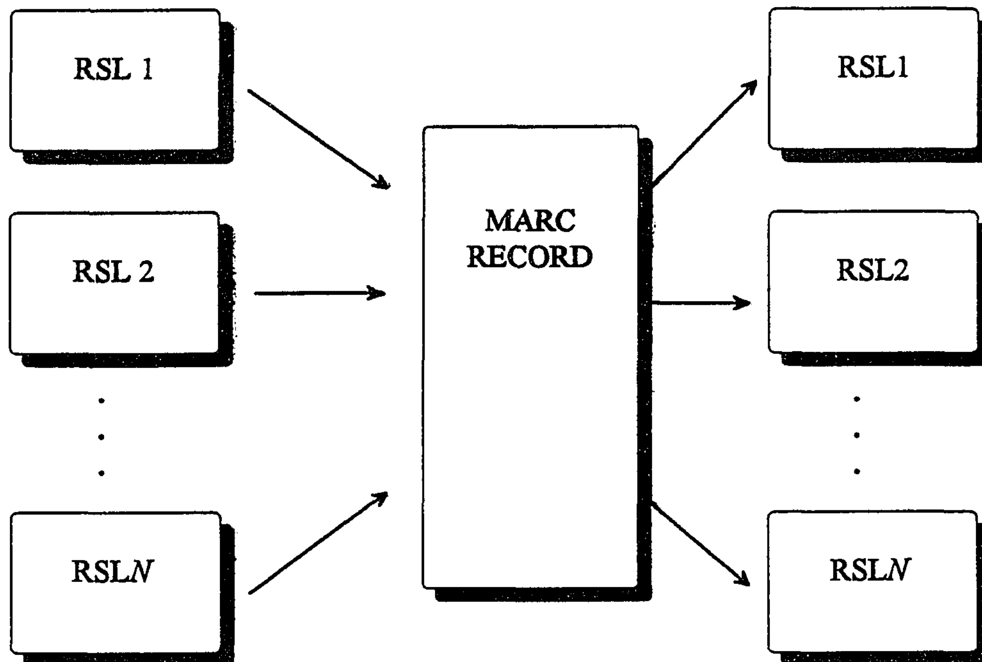


Figure 4.1.2

4.2 USMARC Records

Essentially a MARC record is made up of leader information, a directory and data values. Below is a description of a standard MARC record, developed at the Library of Congress in the late 1960s [Onl89].

The first 24 characters of a MARC record is the leader. As you will notice in our examples of SUCat's MARC records, many parts of the leader are currently not in use. The pieces of information contained in the standard leader are as follows (each character is stored in a single byte):

CHAR DESCRIPTION

- 00-04 Logical Record Length:

  The first 5 characters of all MARC records contain 5 numeric characters, representing the total length of the MARC record. The length is right justified in the field of 5 characters and zero-filled. For example, if the length of the MARC record is 123 bytes, it would be stored as 00123.

- 05 Record Status:

  This character defines the status of the record. The most common values of this field are "n" (new), "c" (changed) and "d" (deleted).

- 06 Type of Record:

  The format of the record can usually be determined by the code in this field. For example, an "e" in this field suggests that this is a bibliographic record for printed or microfilmed maps.

- 07 Bibliographic Level:

  This field is only defined for bibliographic formats.

- 08-09 Undefined:

  This field is undefined in USMARC.

- 10 Indicator Count:

  USMARC always has 2 indicators, which further describe the field. Therefore, this value is always 2. Indicators supply information about the data in the field which is often significant for online operation (such as how the field is indexed) and for card production (such as whether the field prints on catalogue cards). This field remains blank in SUCat.

- 11 Subfield Code Count:

USMARC always has two-character subfield codes: the delimiter "|" and a single character identifier. The subfield code count is always 2.

- 12-16 Base Address of Data:

  This field is a five-character, right-justified, zero-filled number, representing the offset at which the first data character is found.

- 17 Encoding Level:

  This field identifies the degree of completeness of the record. Blank represents "full." It is blank in SUCat.

- 18 Descriptive Cataloguing Form:

  This single character represents the form of descriptive cataloguing used in the record. Blank means no information was given, "a" means the book was catalogued using AACR2 provisions, "i" means ISBD (International Standard for Bibliographic Description) was used. This field is blank in SUCat.

- 19 Linked-Record Code:

The two possible values in linked-record code are: blank, which indicates that no related record is needed in order to fully process the record; and "r", which indicates a related record is required to fully process this record. This field is blank in SUCat.

- 20 Length of Field-Length portion:

  The field length of each tag field is always 4, which is always the value in the 20th byte of the leader.

- 21 Length of Starting-Character-Position portion:

  The length of the starting character position is 5, which is the value given to this item.

- 22 Length of Implementation-Defined portion:

  There is no implementation defined portion; thus a 0 value is given to this item.

- 23 Undefined:

  A 0 is placed in position 23 as a placeholder.

The directory consists of a series of entries, one for each field in the record. These entries directly correspond to the fields of the repository which exported an asset (package and/or related information) into this MARC record. Each entry contains a tag, a field length and the address of the starting position of the field's information in the MARC record. The entries are placed in ascending order in the directory, using the tag as the key.

The data values correspond to the fields in the repository from which the package is exported. In a MARC record the data is labeled using three-letter tags. The tag is stored in the directory entry for the field, not in the data entry.

As mentioned before, the directory is made up of fixed length entries whose first three characters are the tag number. These tags are placed in numerical order into the directory. This makes searching for an appropriate data field very efficient: search the directory for the appropriate tag; find the starting position and length of the corresponding data field; get the information (see Figure 4.2.2 and 4.2.3).

A very short MARC record is given below together with a description of its contents. First we define the delimiters found in the MARC record's transmission format. Please note that the display of the actual delimiters would not print here, therefore, a different set of display characters were chosen. The

actual MARC records contain the characters that correspond to the ASCII Code column in Figure 4.2.1 below.

Delimiter Character	**Displays here**	**ASCII Code**	**Definition**
Record Terminator	x	0001 1101	It is used to terminate a MARC record.
Field Terminator	#	0001 1110	It is used to terminate a field within a MARC record.
Subfield Delimiter	=	0001 1111	Delimiter for subfield code. It is followed by a lower case letter or digit.

MARC Delimiters, ASCII Codes and Display

Figure 4.2.1

00242n 2200085 4500001001100000204001100011520002600022 650009600048901001200144#ZIB_000007# =ablas3 # =amatrix * matrix BLAS # =aNumber Theory; Computer, Mathematical, and Physical Sciences; Mathematics; Number Theory; # =apackage #x

Example of MARC Record in Transmission Format

Figure 4.2.2

Bytes	**Field**	**Contents (~ indicates blank)**
00000-00023	leader	00242n~~~~2200085~~~4500
00024-00083	directory	<tag> <length > <start position> 001 0011 00000 204 0011 00011 520 0026 00022 650 0096 00048 901 0012 00144
Base address (starts at Byte	**Tag** 00085)	
'00000	'001 Asset ID	ZIB_000007#
'00011	'204 Title	~~=ablas3~#
'00022	'520 Abstract	~~=amatrix~*~matrix~BLAS~#
'00048	'650 Subject Class.	~~=aNumber~Theory;~Computer,~Mathematical, ~and~Physical~Sciences;~Mathematics;~Number ~Theory;~~#
'000144	'901 Type	~~=apackage#

Marc Record -- Decoded Contents

Figure 4.2.3

4.3 Software Repositories

If all repositories stored identical information about the packages located in them, interfaces to send the data from one repository to another would be unnecessary. However, this is not the case. Repositories have heterogeneous formats. For instance, the fields corresponding to package information in NIST's GAMS repository are: Package #, Package Name, Type, Port, Description, Language, Development, Citation, Distribution, Computer #, Computer Name, Support Level, Access, Version #, Version Name, Library Documentation, Module Documentation, Sample, Source, Tests, Date Introduced and Date Superseded. The fields in DOE/UT@K's Netlib repository are: Object ID, Library ID, Type, Title, Author, Version, Library Entry Date, Abstract, Subject ID and Keyword. Clearly, a good portion of the field names do not match; nevertheless, some key information must be passed between these repositories in order for users to know which package(s) (or library) to examine.

To insure that as much useful data as possible will be transferred from one repository to another, MARC must have a tag field to correspond to every possible field in each repository. Therefore, the union of all fields would actually be very large. Figure 4.3.1 is a simplified example. The fields M0 through M9 in Figure 4.3.1 correspond to a union of all fields in the repositories linked into this simplified system. Assume that repository A (RA) has fields A1 through A6 and repository B (RB) has fields B1 through B7. For

RB to import a package's information from RA, RA must export that package's information into a MARC record. In doing so, RA will export all its fields; however, RB will read only the fields that are of interest to it.

MARC tag fields	*RA fields*	*RB fields*
M0	A5	B7
M1		
M2	A2	B6
M3		
M4	A1	B4
M5	A4	B3
M6	A3	B2
M7		B5
M8	A6	
M9		B1

Example -- Transferring Fields between MARC Records

Figure 4.3.1

For instance, in this example, when RA exports fields A5, A2, A1, A4 and A3 they will be read into RB's fields B7, B6, B4, B3 and B2, respectively. Note that RA's field A6 has no corresponding RB field. Therefore, A6 will be disregarded by RB. Furthermore, RB's fields B5 and B1 have no corresponding fields in RA. These fields will remain empty.

4.4 Generating USMARC Records

In this research, USMARC records are generated in two stages:

- First the information to be put into the MARC record must be read and placed into the MARC generating program's internal USMARC record format. This information must be retrieved from the repository which stores the package. Thus, retrieving this data is totally dependent upon how the repository stores its information.

- Then, the information which has been placed into the program's internal MARC record format, together with appropriate delimiters (i.e., end of field marker, end of record marker and subfield indicator), must be used to create a standardized external MARC record.

For this research, four software repositories, GAMS, Netlib, ELib and StatLib, have been investigated for the purpose of extracting information necessary to produce MARC records. Because each repository stores its data in different formats, programs used to extract information from each repository differed. A suggestion for further research in this area would be to create a versatile program capable of handling the extraction of information from any repository.

The internal MARC record format is data independent. Therefore, the same procedures can be used to create external MARC records from the internal

MARC records, regardless of the internal MARC record's origin. See Appendices 1, 2, 3 for more details on creating MARC records.

For this research, tools have been created to read MARC records and print them as Rig Forms, Submission Forms and Readable MARC records. These marctools can also read one field of a MARC record and display it. Some examples of MARC records and how they are displayed using the different marctools are shown in Appendix 5.

In the spirit of this research, a reusable software library, marclib, was created. Marclib contains reusable procedures, used to read from and write to MARC records. Details can be found in Appendix 4.

4.5 Summary

This chapter shows the advantages of using MARC records as a data independent structure used in the transfer of information among repositories. It describes the structure of MARC records, how the fields may differ in various repositories and how this affects the MARC records. The process of generating MARC records is then discussed. An example of a MARC record is given.

In the following chapter we will discuss some problems that have been found in the current SUCat indexing schemes. We then propose a new taxonomy for classifying software.

5. Indexing Schemes

5.1 Introduction

A new taxonomy for classifying software is proposed in this chapter. Sections 5.2 and 5.3 give an example and discuss some problems that have been encountered. The goal of solving these problems has led to the research in the following sections.

In Section 5.4 we discuss the network and data structures which will serve as the basis for a new keyword-based dynamic classification scheme for software reuse. Sections 5.5, 5.6 and 5.7 explain how to insert package information. They describe the data structures to be used and give an algorithm for package insertion. In Section 5.8 we demonstrate how the system may change in time. Section 5.9 describes how to retrieve information from the network. Section 5.10 gives an example of how to use the system. Finally, Section 5.11 discusses the issue of performance, highlighting the benefits of this proposed system.

5.2 Problems

Every software library must contain a system of indexing and retrieving software. It is the effectiveness of this system (as well as the software contained in the RSL) that determines how well the RSL will respond to its users.

Four indexing schemes have been chosen for the SUCat project. They are: title index, author index, subject index, and abstract text index. As mentioned before, the title index is used to find locations of software based on the software package's name. Similarly, the author index finds packages based on the name of the author of the package. The subject index is a keyword search based on a subject classification scheme to be discussed later in this section. Each package has a number of keywords which classify it. Similarly, each keyword has a list of packages associated with it. A subject search on a keyword returns the associated list of packages. Finally, the abstract text indexing tool allows users to enter text strings. This text is compared to the abstract of each package. Only packages with exact matches in their abstract text are returned.

Each of these strategies for finding software utilizes an existing thesaurus to help users choose appropriate terms for locating software. The thesaurus contains a list of terms that are used by the particular indexing scheme.

The first two indexes, title and author, are straightforward and can be used to extract the requested packages. However, as discussed below, problems have been found with SUCat's subject (keyword) and abstract text indexing schemes which must be resolved.

Determining an optimal set of subject keywords is difficult. The decision to use keywords based on the different subjects in a university catalogue had been made for the SUCat project. The logic behind this choice was that there is a parallel between the areas in which software is written and the courses given in colleges. For example, some of the keywords which correspond to college courses are Applied Mathematics, Computer Science, Digital Image Processing and Electrical Engineering. Once the set of initial keywords was chosen, a list of keywords was associated by hand with each package, and stored in a file. Similarly, a list of packages was associated with each keyword. In this way, a search on a particular keyword would return the list of packages with software pertaining to that keyword. However, we have determined that in many cases the chosen keywords return either too many or too few packages.

The keywords that have been chosen (for the subject search) to classify the packages in SUCat return anywhere from one package (which is less than one-half of one percent of all packages) to 213 packages (which is more than 97% of all packages). Technically, given our search parameters, these are the

correct number of packages that should have been returned in each case; but these results indicate that this is clearly not an optimal set of keywords.

We propose a hierarchical approach to alleviate the problem. A hierarchical approach would subdivide the domain of keywords returning a large number of packages into smaller domains, each with a corresponding keyword. This would give the user a choice of searching a number of new keywords, each with a smaller domain than the original keyword and likewise fewer packages. Properly classifying a collection of software is essential to making reuse an attractive alternative [PF87]. A highly functional keyword-based system needs a good set of keywords which will cover all the domains of the packages and still return a workable number of packages.

One solution to correcting the set of keywords is related to the abstract text indexing scheme. We will discuss the problems related to abstracts and conclude with a possible solution to improving our keyword search.

When a system uses an abstract text indexing scheme there must be a description of the component (by component, we mean either module or package, whichever is appropriate), or an abstract, in order to retrieve the component. Obviously, the better the description the more likely the software is to be found by other users.

Ideally, when authors write code they should include a well documented abstract which will inform users about the subject area as well as the exact functionality of the code. (Other details should be included as well, but not necessarily in the abstract.) One method of retrieving appropriate software is searching the software abstracts for text that best matches a user's request. Browsing through the text thesaurus could help a user choose appropriate text.

5.3 An Example

A problem with the abstract text index is illustrated in the following actual example. When a user wants to find software to use the "Gaussian elimination" method of solving linear systems, the user can employ the abstract text index. A search for "Gaussian elimination" on SUCat, which contains the union of all packages in Netlib, GAMS, ELib and StatLib, returns one single package. When doing the exact same search on Netlib alone, nine packages are returned!

This discrepancy is found to stem from the fact that the SUCat search uses only package-level abstracts, whereas the Netlib search uses module-level abstracts. When abstracts were needed for the abstract text indexing scheme, they were created on the package level (one abstract per package). Unfortunately, some of these abstracts do not give a clear picture of the type of software contained in the package. Here are some examples:

library name: c++

abstract: miscellaneous codes in c++

library name: go

abstract: Golden Oldies: widely used, but not in standard libraries

library name: misc

abstract: various stuff collected over time

All the text searches that have been examined during the course of this research have searched only abstracts not the code itself. When the search for "Gaussian elimination" was made, only one package's abstract mentioned this term, whereas at least nine packages (those found in Netlib) contained code relating to this topic. There are two possible methods of correcting this problem.

First, this problem can be ameliorated by making a union catalogue's abstract index scheme distributed. Instead of searching the vague package abstracts of the union catalogue, a search should be conducted by issuing requests to each of the relevant repositories. The server will get a list of packages from the individual repositories. The union of these lists will be sorted, with duplications omitted, and returned to the user. Searches in repositories are and will continue to be performed on the module level. The module abstracts contain more information than the package abstracts; therefore more of the needed package ids would be returned.

Second, the problem can be corrected at its source. The search was performed on vague package abstracts, as opposed to more detailed module abstracts. Abstracts should not be written in vague generalities. Without useful descriptions, software written for the purpose of reuse will never be reused. Whether on the module level or the package level, hazy descriptions are

worthless. Specific terms (e.g., "linear equations", "sparse matrix multiply", "Gaussian elimination", "fast fourier transforms", "eigenvectors"--as opposed to "stuff", "widely used" and "miscellaneous") which would help a user locate required software should be used to describe the package or module.

A "rule of thumb" for writing abstracts should be: Make sure your abstract supplies the terms a potential user might use to find this software.

Once useful abstracts are written, we can solve a problem related to the subject classification index. Terms to be used as keywords can be extracted from the abstracts. These keywords can be used to help classify packages in a dynamic classification scheme. A dynamic classification scheme capable of evolving to satisfy the changing needs of the software community would be most valuable.

5.4 The Logical Forest

We propose a taxonomy for a dynamic classification scheme for storing and locating software. The scheme is capable of evolving to satisfy the growing needs of the software community. The ideas and algorithms that follow are to be incorporated into a system designed for the purpose of storing and retrieving software from distributed heterogeneous software reuse libraries.

Physically, the data structure behind the classification scheme is a network and not a series of trees. However, in the context of a person using the system, it can be viewed as a forest. A forest consists of zero or more trees, each of which consists of one or more nodes. The user will never see the entire network structure, but rather a portion of it that looks like a tree, and can be traversed in a tree-like manner. Figure 5.4.1 shows an example of a network structure and several possible traversals. Each node in the network will contain child pointers, which will allow tree-like traversals to occur starting downward from any node. Upward traversals will be limited to one parent node, as will be described later in this chapter.

A network root is a node in the network that has no parents and serves as a root to a logical tree. A terminal network node is a node in the network which has no children, but rather points to the packages which pertain to its keyword. Once we traverse from a node, say node *a*, to its child, say node *b*,

node *a* will remain the parent of node *b* during this entire traversal of a logical tree, despite the fact that node *b* might have several parents in the network.

Each node will contain a keyword, which may be a word or a phrase descriptive of the software modules emanating from this node. For example, when dealing with mathematical and statistical software, each keyword would be a mathematical or statistical subject category, such as Number Theory, Integral Transforms, Data Summarization and Elementary Statistical Graphics.

The keyword of a node which will serve as a root within the network will be a most general topic. No other category or keyword would point to this node (its parent pointer would be NULL); however, it would point to other categories. As can be seen in Figure 5.4.1, root nodes, intermediate nodes and terminal network nodes may serve as roots to logical trees.

The keyword of a node which will serve as a terminal network node (or a leaf of a logical tree) will be a most specific category. Other categories would point to it. This terminal node will not have child pointers, but rather, package pointers, pointing to a list of packages which contain code pertaining to its keyword. If a user starts a search using the keyword of a terminal network node, the node will serve as both the root and the leaf of the logical tree, and it will lead directly to software packages.

Intermediate nodes will contain categories which are pointed to by other more general categories, and also point to more specific categories. There may be zero or more intermediate nodes between root and terminal nodes.

Expansion of any node will produce a logical tree. For example, expanding the node (or browsing down in the logical tree) whose keyword is "Integral Transforms" (a small piece of GAMS' hierarchy) would yield the logical tree shown in Figure 5.4.2. In this example, Integral Transforms is a root node, and one-dimensional-FFT-complex is a leaf node, with several nodes in between these two. Fast Fourier Transforms may also be an intermediate node on some other tree, with a different root. A user starting a tree traversal at Integral Transforms will be unable to see the part of the network containing the other root. Note that this system avoids redundancy because nodes, such as Fast Fourier Transforms, physically appear just once in the network, no matter how many logical trees they may belong to.

Each node's parent and children pointers will determine its position in one or more logical trees. Since this structure is, in fact, a network, each node may be pointed to by more than one parent. The benefit of this structure is that it will allow any node to be part of more than one logical tree, without the problem of redundancy.

In order to avoid looping when producing a logical tree, each node will have location markers, showing a subtree (or tree id) and a level number indicating

the location of this node (level 0 being a network root) and a boolean called terminal_node used to show if the node is a terminal network node. Using this information, the system will make sure that nodes cannot become their own descendants (or ancestors). Each node will contain a list of locations, indicating the logical tree(s) originating from network root(s) containing this node and on what level of the logical tree the node is located. The location of each node is determined by that of its parent(s). For example, if a parent is located in *tree a* at level 2 the child is located in the same tree at level 3. Our algorithm prevents a node from being located in the same tree at different levels. However, in another tree the node can be located at a different level. In this way, since each node is always located on one level within any logical tree, a node cannot become its own ancestor or descendant.

In Section 5.10 we will discuss the possibility of keeping more detailed location markers to allow more flexibility in placing nodes within a logical tree.

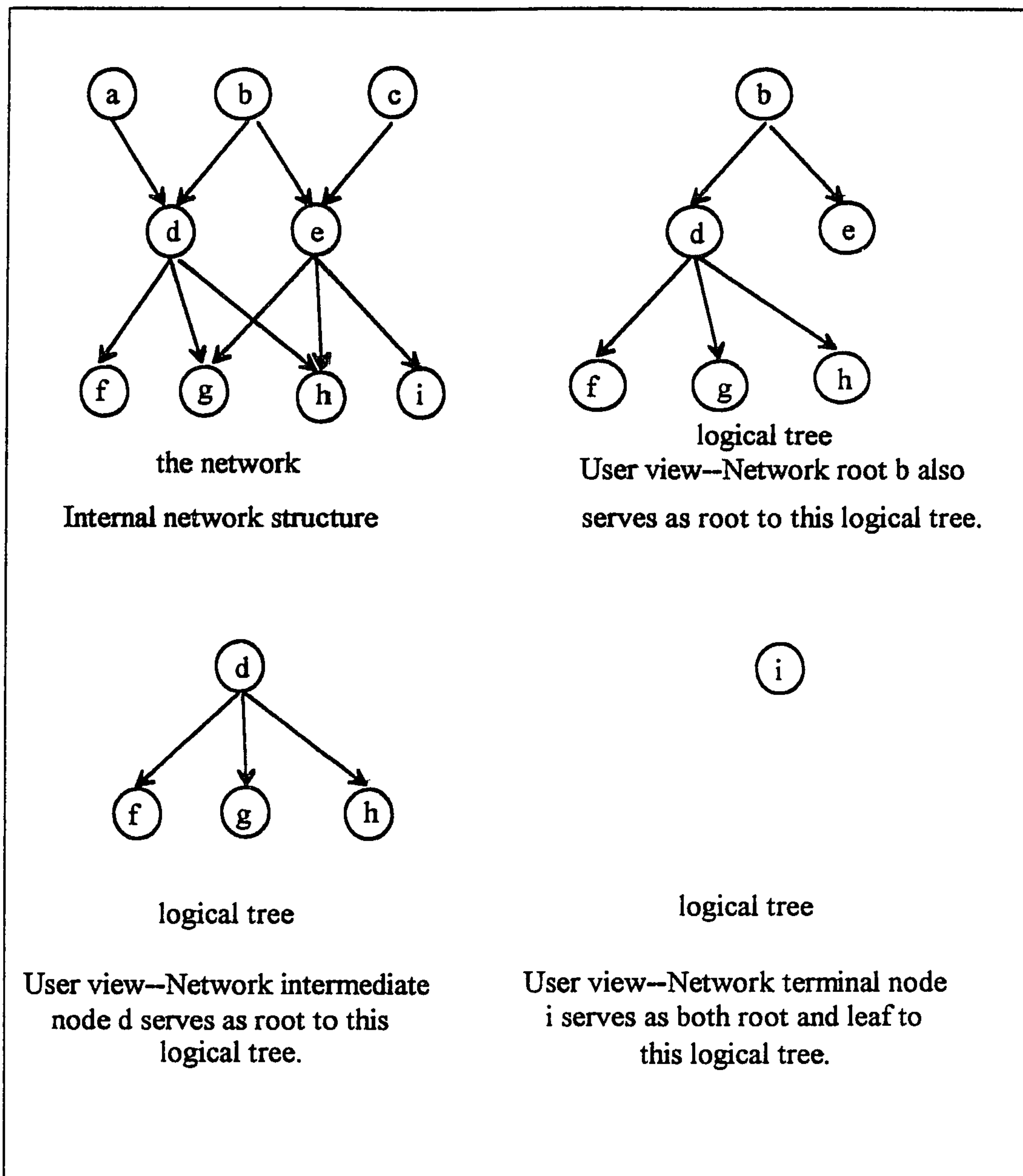
An initial set of keywords will be chosen from GAMS' hierarchical classification scheme. GAMS' tree-like structure can easily be adapted to be suitable for our network.

Of course, it would be unrealistic to believe that the first set of keywords chosen will be perfect for the job, in the sense that users will always be able to find exactly the packages they need. The semantics of the titles chosen will

sometimes result in the user choosing the wrong logical tree. Suggestions for more and/or better keywords to find the software will be made by those who insert the software, as well as by those who will be searching for existing software. Achieving the goal of attaining available software easily will be a dynamic process. The major component of this process will be a dynamic list of nodes containing keywords, which will grow and shrink based on recommendations from users and creators of software.

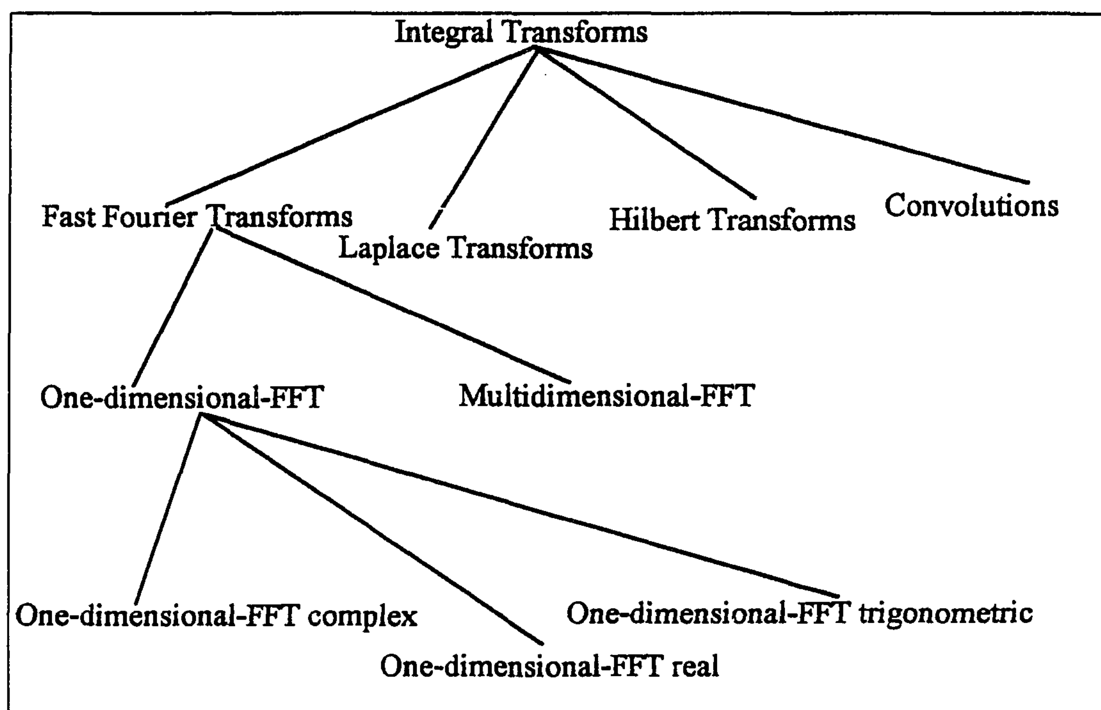
There will be a table (which we will refer to as Node Table) consisting of all the existing nodes ordered alphabetically by the node's keyword. Each node will contain a keyword, a boolean which will indicate whether or not this is a terminal node (pointing to packages), pointers to the list of the node's parents and the node's children, and pointers to a dictionary definition, a synonym list and to a list of location markers (see Figure 5.4.3) which indicate the location(s) of each node in the logical trees.

A thesaurus will aid in locating packages which emanate from a node with a keyword that corresponds to a user's search word. The thesaurus will contain a list of synonyms for each keyword that has one or more synonyms. For instance, suppose "Fast Fourier Transforms" is a keyword. Synonyms for this entry may be "FFT" and "fft". A keyword search for "Fast Fourier Transforms" will automatically search for "FFT" and "fft" as well.



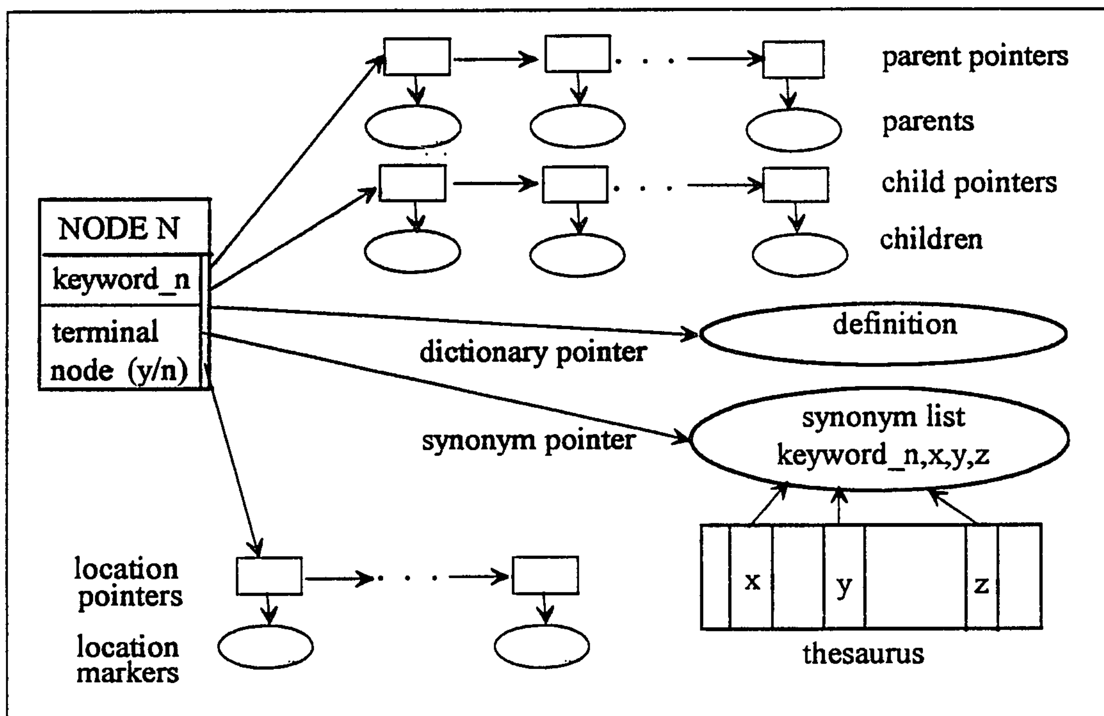
A Network Structure and Several Possible Logical Trees

Figure 5.4.1



Expansion of Node

Figure 5.4.2



A Node in the Network

Figure 5.4.3

5.5 Inserting Package Information

It is necessary to assist authors in properly classifying their packages so that they can be located by users. For this purpose, we propose an interactive tool which will prompt the author for various pieces of information. The interactive tool could be used to gather cataloguing information for the SUCat as well as information for inserting new packages into the classification data structure. An algorithm for inserting new packages into the classification data structure appears in Section 5.7. The minimal information needed to insert new packages is the location of the package and an abstract. A list of keywords will be optional.

When the author finishes entering the information, the abstract is searched for strings or phrases which match existing keywords. Pointers to nodes whose keywords match words extracted from the abstract will be stored either in the package's parent list (for leaf level keywords) or a temporary list (for non-leaf level keywords) (see Section 5.7, Insertion Algorithm, step 2). At this point three possible things may occur: matches may be found in leaf node keywords; no matches may be found; or some matches may be found (however, not in leaf node keywords). It is our goal to find matches with leaf node keywords or their synonyms, since only leaf nodes will contain pointers to packages pertaining to their keyword.

If no matches are found, the author will be prompted to examine the list of synonyms or the alphabetized list of keywords in the Node Table for keywords to find an appropriate keyword and insert the corresponding node into the parent list (see Section 5.7, Insertion Algorithm, step 4). The author may not find any existing keywords that apply to the package and may believe there are other keywords which would be appropriate, but as yet are not in the Node Table, or the author may feel an existing keyword should have a synonym. In either case, the author may choose the option of entering a suggestion (see Section 5.7, Insertion Algorithm, step 5). Suggestions will be saved for consideration by the administrators of the system. These suggestions will be evaluated and will be partially responsible for the evolution of the system.

If a match is made to a non-leaf node word, the author will be shown a logical tree which starts at the given word and may be expanded to all the leaves of that segment of the logical tree. The author will then be asked to choose all keywords from terminal nodes that appropriately classify the package. Pointers to nodes whose keywords match the words chosen will be placed in the package's keyword list. If none are appropriate the author may make requests for new synonyms or keywords when suggestions are solicited to (1) add a new synonym (2) add a new keyword (see Section 5.7, Insertion Algorithm, step 3). Adding synonyms and/or keywords is performed via the system administrator.

5.6 Data Structures

Before outlining the algorithm, we will describe and/or illustrate the data structures involved.

node

keyword	parent list	child/package list	dictionary pointer	synonym pointer	location list	terminal node flag

keyword is a character string containing a general category description of all nodes emanating from this node (e.g., Number Theory, Statistics and Probability).

parent list is a list of parent pointers.

parent pointer

pointer to parent node	next parent pointer

child/package list is a list of child/package pointers.

child/package pointer

pointer to child node or package	next child pointer or package pointer

dictionary pointer is a pointer to a character string defining the node's keyword.

synonym pointer is a pointer to a character string containing a list of synonyms of this node's keyword from the thesaurus. The list includes this node's keyword.

location list is a list of location markers.

location marker provides the name of a network root (which is the tree_id) and the level number of this node beneath the network root.

tree id	level number	pointer to next location marker

terminal_node flag is a boolean. Value TRUE means that this node is a terminal network node and its children are packages. Value FALSE means this node is not a terminal network node and its children are other nodes.

Node Table is an ordered list of pointers to all the nodes in the network. The list is ordered alphabetically using keyword as the key.

pointer to node	next node pointer

package's parent list is a list of pointers to nodes whose keywords describe the package.

browse list is a list of browse nodes. When traversing a logical tree a browse list will be formed. The browse list contains the current parent of each node in

the list, thereby making browsing up a logical tree possible. Algorithms in Section 5.7 will use the browse list.

browse node

pointer to node	bn_parent	next browse node

bn_parent is a pointer to the active parent of a node in a browse list.

5.7 Algorithms for Inserting New Packages

The following algorithms are the building blocks of the algorithms which follow, both here and in the rest of this chapter.

- **expand(*pointer to node* n)** creates a logical tree from a node pointed to by n and makes that node's children available to the user.

```
for each i that is a child of the node pointed to by n
    display i in tree;
```

- **create_list(*pointer to node* n, *pointer to list of leaf level node pointers* leaf list)** recursively creates a list of pointers to all leaf-level nodes emanating from the node pointed to by n. The author of the package will use this leaf list to choose parent nodes to add to the package's parent list .

```
if the node pointed to by n is a leaf node {
        insert n into leaf list;
        return;
}
for each i that is a pointer to a child of the node pointed to by n
        create_list(i, leaf list);
```

- **synonym(*pointer to browse node* bn)** displays the character string pointed to by bn->synonym pointer.

    display contents of memory pointed to by bn->synonym pointer;

- **definition (*pointer to browse node* bn)** displays the dictionary definition pointed to by bn->dictionary pointer.

    display contents of memory pointed to by bn->dictionary pointer;

- **get_selected_packages (*pointer to browse node* bn)** calls create_list to create a list of all terminal network nodes emanating from the node pointed to by bn->pointer to node. Then, for each package pointed to by each terminal network node in the list, the user may choose to retrieve or not to retrieve the package.

```
create_list(bn->pointer to node, list);
for each terminal network node, tn, in list
/*terminal network nodes point to packages */
        for each package pointed to by tn
                allow user to choose to retrieve or not to retrieve package;
```

- **browse_down (*pointer to browse node* bn)** browses down the logical tree.

```
/* the user has selected browse_down (bn) in browse_options */
        expand (bn->pointer to node);
        user picks chosen_child, a pointer in
                bn->pointer to node->child/package list;
        allocate a browse node, set ccptr to point to it, and place it into
                browse list;
        bn->next = ccptr;
        ccptr->pointer to node = chosen_child;
        ccptr->bn_parent = bn;
        ccptr->next = NULL;
        return (ccptr);
```

- **browse_up (*pointer to browse node* bn)**

```
/* the user has selected browse_up (bn) in browse_options */
        if bn->bn_parent = NULL {
                print (bn->pointer to node->keyword, "is root of logical tree");
                return (bn);
        }
        return (bn->bn_parent);
```

- **add_to_list (*pointer to list of pointers* list, *pointer to node* np)** places np into list.

    allocate new node, set nn to point to it;

    locate last node in current list, let lp point to it;

    lp->pointer to next node = nn;

    nn->pointer to node = np;

    nn->pointer to next node = NULL;

- **browse_options (*pointer to browse node* bn)** allows a user to choose from: viewing bn->pointer to node's synonyms; viewing bn->pointer to node's definition; browsing up the logical tree from bn->pointer to node; retrieving selected packages which emanate from that node; exiting the subroutine; or if bn->pointer to node->terminal_node flag shows that it is not a terminal network node, the user may browse down the logical tree.

choose from: {

(a) synonym (bn)

(b) definition (bn)

(c) bn = browse_up (bn)

(d) get_selected_packages (bn)

}

if bn->pointer to node->terminal_node flag = FALSE

/*not a terminal network node*/

then add choice below

(e) bn = browse_down(bn)

return (bn);

Algorithm for inserting new packages into classification data structure:

(1) Author completes form for new package. The information includes an abstract and an optional list of keywords.

Allocate space for a package node pointed to by new_pkg_ptr.

(2) The abstract is analyzed: words matching keywords (or their synonyms) of existing leaf-level nodes are extracted from the abstract. A parent list, which is a list of pointers to terminal network nodes whose keywords classify this package, is created. Pointers to non-leaf-level nodes whose keywords match keywords (or their synonyms) in the abstract are inserted into temp_list, a temporary list. The list of keywords from (1) are analyzed in the same way as the abstract and taken care of in the same way.

(3) nodeptr = pointer to first node in the temporary list, temp_list,

```
        whose node's keyword matches one of the non-leaf-level keywords;
    while (nodeptr != NULL) {
        allocate a browse_node, set bn to point to it, and insert into
                browse list;
        bn->pointer to node = nodeptr;
        bn->bn_parent = NULL;   /*root of logical tree*/
        continue_browsing = TRUE;
        while (continue_browsing = TRUE) {
            bn = browse_options(bn);
            choose from: (a)continue_browsing = TRUE;
```

```
                    (b) continue_browsing = FALSE;
        } /*  end while continue_browsing */
        if bn->pointer to node->terminal_node flag != TRUE {
            /* not a leaf-level node */
            create_list(bn->pointer to node, list);
            allow user to choose from list of returned leaf-level nodes;
            for each cp, which is a pointer to chosen leaf-level node
                add_to_list (parent_list, cp);
        } /* end if */
        else  /* node is leaf_level */
            add_to_list (parent_list, pointer to node);
        nodeptr = nodeptr->next;
    } /* end while nodeptr != NULL */
```

(4) If pointer to package's parent list = NULL allow user to search all

keywords from Node Table;

insert pointers to nodes (whose keywords were chosen) into the

package's parent list;

Choose one or more of the following:

(a) add new keyword(s) (via system administrator).

(b) add new synonym(s) (via system administrator).

(c) exit this step.

(5) If pointer to package's parent list != NULL

```
/* For each parent node pointed to in the parent list, a pointer to the
 new package must be included as one of the parent's packages. */
kw = a pointer to the first element in parent list;


while (kw != NULL)   {
        allocate a new package pointer, set npp to point to it, and place
                into kw->pointer to parent node->package list;
        npp->pointer to package = new_package_pointer;
        kw = kw->next parent pointer;
}
else    /* keyword list = NULL and step 4 did not correct the situation*/
        reject package; /* send information to system administrator */
author may select
                i) add new synonym(s) (via system administrator).
                ii) add new keyword(s) (via system administrator).
```

5.8 How the Classification Data Structure May Change in Time

When a user inserts or extracts information from the Node Table, the nodes appear to be part of a logical tree. However, as mentioned earlier, the actual structure of the nodes is a network. Therefore, nodes may contain a list of parent pointers rather than just one parent. We must limit traversals to create a tree-like structure (e.g., traversal up the structure must be made via the same nodes which led down the structure initially--this is assured with the use of the browse_node parent).

Suppose an author finds a node described by (and therefore containing) the keyword, word_a, whose parent pointer is parent_a, which is also a node in the tree. The author feels the node containing word_a should also be pointed to by parent_b, another node in the tree. A second parent pointer may be added to a node as long as the new parent, parent_b, is not a descendant of the original node, described by keyword word_a. (With this precaution no looping will occur when traversing down the tree.) In order to make traversing up the logical tree possible when in browse mode, it will be necessary to keep track of the current parent of each node (namely, the parent node that led to this node). After adding parent_b to the node's parent pointers, parent_b's children list will be extended by appending a pointer to the node whose keyword is word_a. Figure 5.8.1 shows the new pointers which are necessary when adding a new parent to a node.

The algorithms below will be referenced in the algorithm for inserting new keywords which follows:

- **check_parent_location** (***pointer to node*** **temp_parent,** ***pointer to parent pointers list*** **parent list**) returns a flag which indicates whether or not temp_parent may become a new parent pointer in parent list. In this algorithm, a new parent to a node will be accepted only if there is no case where another parent to this node exists in the same tree but at a different level.

```
i = pointer to first element in temp_parent->location list;
while (i != NULL) {
        p = pointer to first parent in parent list;
        while (p != NULL) {
                j =  pointer to the first element in parent->location list;
                while (j != NULL) {
                        if i->tree id = j->tree id and
                              i->level number != j->level number
                        then return (FALSE);
                        j = j->next;
                } /* end while j */
                p = p->next;
        } /* end while p */
        i = i->next;
```

```
} /* end while i */
return (TRUE);
```

- **check_child_location (*pointer to node* temp_child, *pointer to node* parent)** returns a flag which indicates whether or not temp_child may become a new child pointer in parent list of node pointed to by parent. In this algorithm, a new child can be accepted by a parent only if there is no situation where the parent and the child exist in the same tree and the child is not at one level lower (level + 1) than the parent.

```
i = pointer to the first element in parent->location list;
while (i != NULL) {
    j = a pointer to the first element in temp_child->location list;
    while (j != NULL) {
        if i->tree id = j->tree id
            and i->level number != j->level number + 1
        then return (FALSE);
        j = j->next;
    } /* end while j */
    i = i->next;
} /* while i */
return (TRUE);
```

- **update_locations_list (*pointer to node* parent, *pointer to node* child)** after a new parent pointer is accepted, new location pointers must be set

up for the node pointed to by child, reflecting the location of the node in relation to its parent's locations. In each case, the node pointed to by child is located beneath the same root node (tree id) as its parent with the child's level number being one more than its parent's level number.

```
i = pointer to the first element in parent->location list;
while (i != NULL) {
        allocate a new location marker, set j to point to it, and place it into
                child->location list;
        j->tree id = i->tree id;
        j->level number = i->level number + 1;
        i = i->next;
}
```

New unique keyword nodes are added to the network using the following algorithm:

Insert_new_keyword

1: The user fills out a form which asks for the following information:

(a) new keyword.

(b) keyword(s) for existing nodes to be parent node(s).

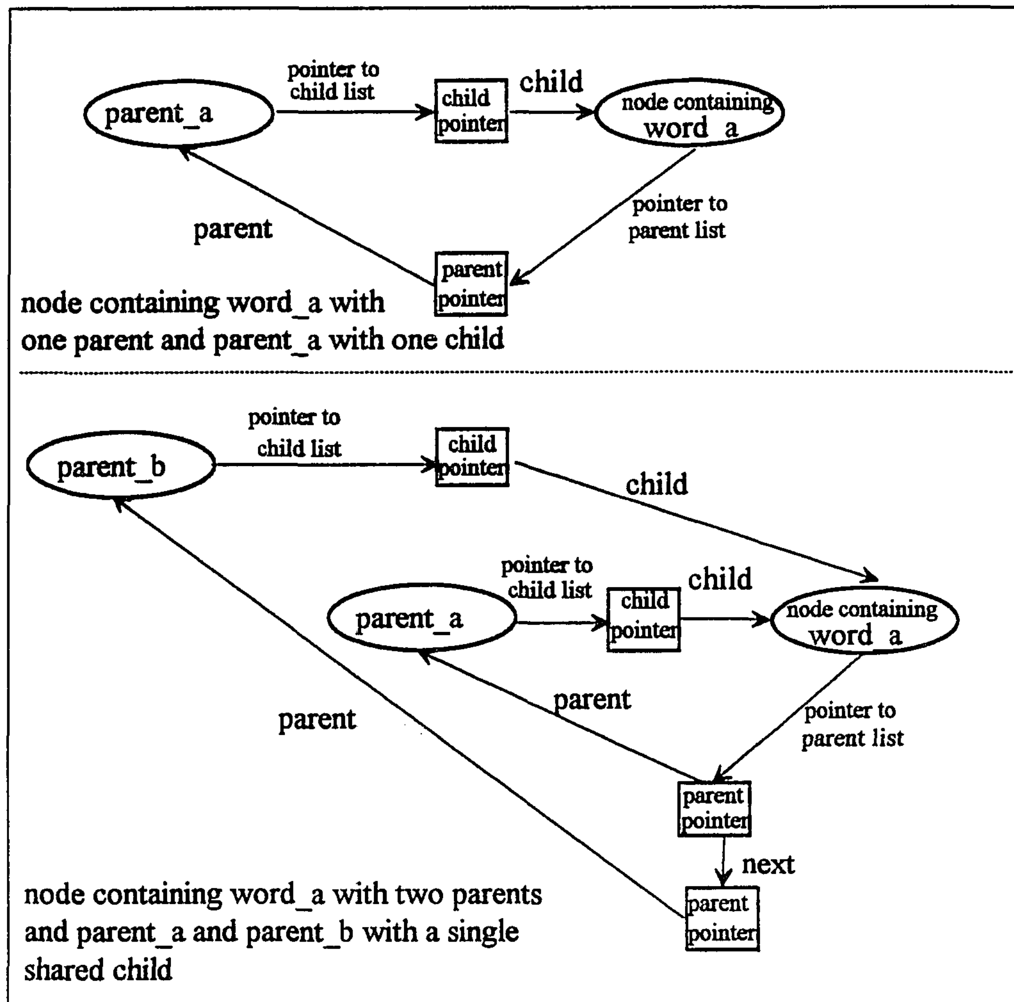
(c) keyword(s) for existing nodes to be child node(s).

If new keyword is an existing keyword of a node in the Node Table exit.

2: Create two lists and initialize them to empty:

```
        (a) rejected parent list

        (b) rejected child list

allocate a new node and set nn to point to it;

nn->keyword = new keyword;

set all pointers in the node pointed to by nn to NULL;

for each p that is a parent keyword (see 1:(b) above){

        temp_parent points to the node whose keyword = p;

        place = check_parent_location(temp_parent,nn->parent list);

        if place {

              update_locations_list(temp_parent,nn);

              allocate a new child pointer for temp_parent, set cp to point

                     to it, and place it into temp_parent->child list;

              cp->pointer to child node = nn;

              allocate a new parent pointer, set pp to point to it, for nn

                     and place it into nn->parent list;

              pp->pointer to parent node = temp_parent;

        } /* end if place */

        else

              insert temp_parent into list of rejected parents;

} /* end for each p */

for each c that is a child keyword (see 1:(c) above){

        temp_child points to the node whose keyword = c;
```

```
        place = check_child_location(temp_child, nn);
        if place { /*this means temp_child may be new_node's child */
            update_locations_list(new_node, temp_child);
            allocate  a new child pointer for new_node, set cp to point
                 to it and place it into new_node's child list;
            cp->pointer to child node = temp_child;
            allocate a new parent pointer for temp_child, set pp to
                 point to it, and place it into temp_child->parent list;
            pp->pointer to parent node = nn;
        }
        else
            insert temp_child into list of rejected children;
} /* end for each c */
if nn->child list = NULL   /* this node may point to packages */
        then nn->terminal_node flag = TRUE;
        else nn->terminal_node flag = FALSE;
insert nn into Node Table preserving alphabetical order;
/* end of adding new keywords algorithm */
```



Adding a Parent to a Node

Figure 5.8.1

5.9 Extracting Information from the Network

There are four methods of extracting information in the SUCat project. As mentioned earlier, there are title, author, keyword and abstract text searches. In our research, making use of the new data structures being proposed, a user will search for and locate information using a keyword/browse search.

The keyword/browse search will search the keywords and then the thesaurus in an attempt to find all possible matches to a user's search string or choice of keywords. When a match is found, the user may request a list of all packages from below this point in the logical tree or may browse down the logical tree in an attempt to retrieve fewer packages.

Algorithm for keyword/browse searching:

User is given the option to:

(a) enter a search word

(b) search all nodes in keyword list (all nodes in Node Table)

(c) search root nodes in keyword list (all root nodes in Node Table)

(d) search leaf nodes in keyword list (all leaf nodes in Node Table)

(e) search thesaurus for synonyms

If choice is (a)

user enters search word;

a text search is used to find a node in the Node Table whose keyword matches search word;

```
        if no match is found the same text search is used to search the thesaurus
                for a match;
        set root of logical tree = pointer to node with matching keyword;
If choice is (b):
        for each w that is a keyword of a node
                display w;
        user chooses a word to search;
        set root of logical tree =  pointer to node whose keyword was chosen
                by user;
If choice is (c):
        for each w that is a keyword of a node containing a
          location marker.level_number = 0 (a network root node)
                display w;
        user chooses a word to search;
        set root of logical tree = pointer to node whose keyword was chosen by
                        user;
If choice is (d):
        for each w that is a keyword of a node whose element
                terminal_node flag = TRUE (a terminal network node)
                display w;
        user chooses a word to search;
```

set root of logical tree = pointer to node whose keyword was chosen by user;

If choice is (e):

for each w that is a word in the thesaurus

display w;

user chooses a word to search;

find the word in the Node Table that corresponds to the matching synonym;

set root of logical tree = pointer to node whose keyword corresponds to chosen word in the thesaurus.

If no match is found return -- search was a failure;

call browse_search (root of logical tree);

/* end keyword/browse search */

An algorithm for browse_search follows:

browse_search (tree_pointer)

allocate a browse node, set bn to point to it, and insert into browse list;

bn->pointer to node = tree_pointer;

bn->bn_parent = NULL; /* root of logical tree */

continue_browsing = TRUE;

while (continue_browsing = TRUE) {

```
            bn = browse_options(bn);
            choose from: (a)continue_browsing = TRUE;
                         (b) continue_browsing = FALSE;
        } /*  end while continue_browsing */
```

Browse_options gives the user various options for navigating in the network and finding out what each keyword means. The following options will be given for non-terminal network node words: (a) retrieve synonym, (b) retrieve definition, (c) browse down, (d) browse up (e) get selected packages (see Figure 5.11.1). These choices will allow the user to browse through the logical tree, look at synonyms, dictionary definitions, expand the node to include its children, or retrieve selected packages emanating from this point in the logical tree or a subset of these packages. The browse down option will be omitted from terminal network nodes.

When the get all packages option is chosen a list of all packages emanating from the active node will be returned. The following options will be given: (a) retrieve abstract, (b) get package and (c) return to tree (see Figure 5.11.2). Users can either look at the abstract of a package, retrieve the package from the repository in which it is located or return to the terminal network node which led them to this package.

Researchers retrieving information from this system will also have the opportunity to write suggestions to the system administrator. Suggestions for new keywords or synonyms can be made to make packages easier to find.

5.10 An Alternate Location List

To avoid endless looping when browsing down a logical tree in our keyword/browse search algorithm we insisted upon the following: a node could not become its own ancestor (or descendant). This was accomplished by making sure that a node can be located in a single logical tree more than once only if its location within the logical tree is always on the same level. Always being on the same level precludes that node from ever becoming its own ancestor (or descendant).

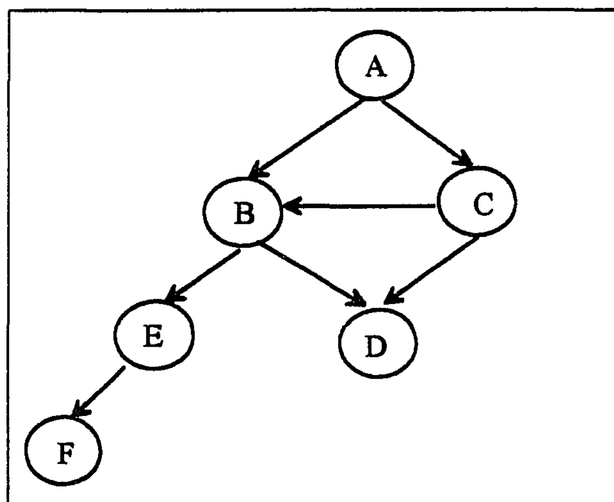
The above method is too restrictive if our goal is to avoid looping. For instance, look at node B in Figure 5.10.1. It would never be allowed to have both node A and node C as parents in our original plan (node A being at level 0 and node C being at level 1). However, as is shown in the logical tree depiction of this network in Figure 5.10.2, there is no reason not to allow both A and C as parents of B. Looping does not occur within this graph; therefore there is no endless loop when browsing down any logical tree within this graph.

We make the following suggestion for determining where nodes may be placed within a network: Each node should contain a list of locations. These locations will be similar to the location markers described in Section 5.6, each containing a tree id. However, instead of a level number, these locations will contain a string of numbers separated by some character (we show a comma),

indicating the exact locations within the logical tree where this node occurs. Figure 5.10.3 shows that one of the locations of node B is (A,2,2). This means that node B is part of a logical tree starting at network root node A, and is root A's second child's (C's) second child. The child's position can be seen more clearly in the logical tree depiction (Figure 5.10.2) of the network. Node B's other location (A,1) means that B is node A's first child. The number of separation characters in each location string indicates the level number of the node within the logical tree (see both logical locations of node B in Figure 5.10.2).

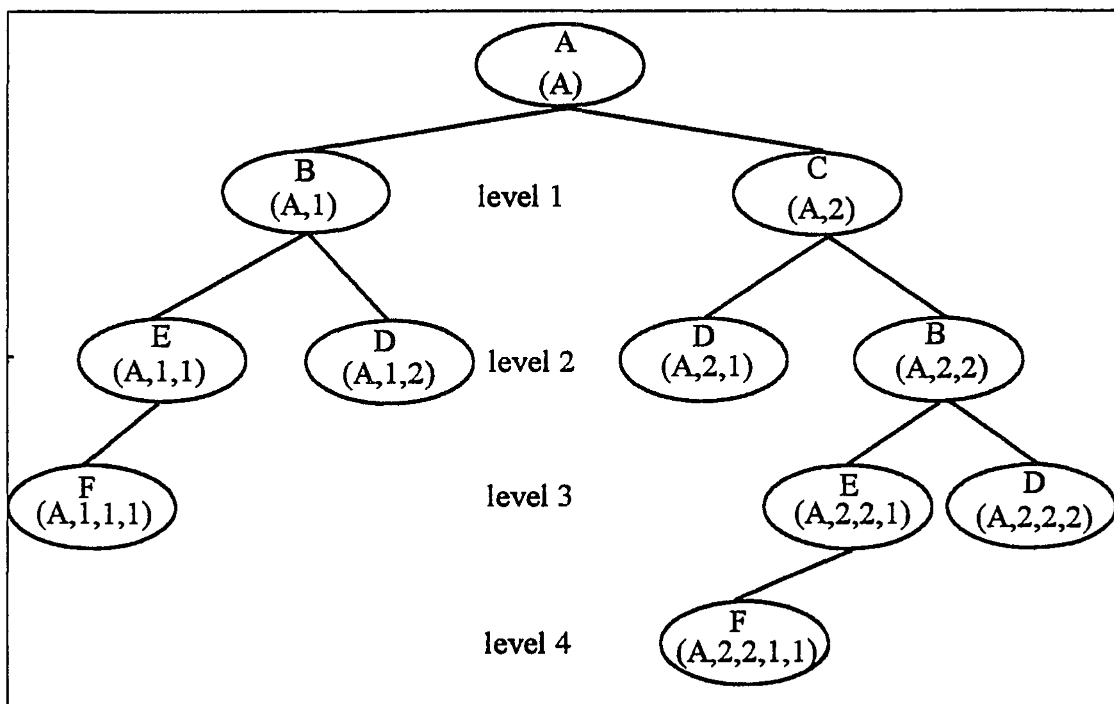
A node may be placed anywhere in a logical tree (and receive a new location string) if the following two conditions are met to insure that there are no cycles in the network:

- no string in the node's location list is a substring of the new location string.
- the new location string is not a substring of another string in the node's location list.



A Network

Figure 5.10.1



Logical Tree Depicting the Network in Figure 5.10.1

Figure 5.10.2

NODE	LOCATIONS	ANCESTORS	DESCENDANTS
A	(A)	NULL	B,C,D,E,F
B	(A,1), (A,2,2)	A,C	D,E,F
C	(A,2)	A	B,D,E,F
D	(A,1,2), (A,2,1), (A,2,2,2)	A,B,C	NULL
E	(A,1,1), (A,2,2,1)	A,B,C	F
F	(A,1,1,1), (A,2,2,1,1)	A,B,C,E	NULL

Tabular View of Figure 5.10.1 and Figure 5.10.2

Figure 5.10.3

A location string, loc a, that is a substring of another location string, loc b, implies that the node located at loc a is an ancestor of the node located at loc b, and that the node located at loc b is a descendant of the node located at loc a. A node having both loc a and loc b as locations generates a cycle. Not allowing a node to have two locations such that one is a substring of another prevents cycles within the network of nodes. Using the locations of a node as suggested above allows the most liberal placement of nodes within a network and insures that no cycles occur in the graph.

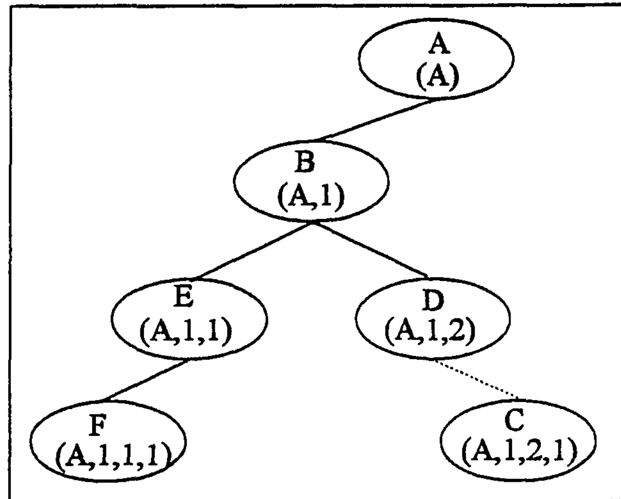
When traversing down the left logical tree emanating from A in Figure 5.10.2, a user may assume it is safe to make node C a child of node D (see Figure 5.10.4). However, this should not be permitted as it would lead to cycles in the graph (see Figure 5.10.5), and therefore, endless looping when traversing down some logical tree emanating from node D. Let us see how our method of preventing cycles will work.

Assume that we decide to make node C a child of node D. D has the following locations (A,1,2), (A,2,1) and (A,2,2,2). We create a list of temporary new locations for C which will indicate its position as D's first child. Therefore, the temporary new locations consist of (A,1,2,1), (A,2,1,1) and (A,2,2,2,1). Now we must compare each of these new locations with C's original location list, which consists of just one location (A,2). Since (A,2), a current location of C, is a substring of (A,2,1,1), one of the temporary new locations, placing C as a descendant of D would create a cycle in the graph, as can be seen in Figure 5.10.5. Therefore, this relationship would not be allowed.

Now let us take a look at how this method will allow legal additions of child nodes. Assume that we decide to make node F a child of node D (see Figure 5.10.6). D has the following locations (A,1,2), (A,2,1) and (A,2,2,2). We create a list of temporary new locations for F which will indicate its position as D's first child. Therefore, the temporary new locations consist of (A,1,2,1), (A,2,1,1) and (A,2,2,2,1). Now we must compare each of these new locations with F's original location list, which consists of two locations (A,1,1,1) and (A,2,2,1,1). Note that (A,1,1,1), the first current location of F, is not a substring of (A,1,2,1), (A,2,1,1) or (A,2,2,2,1) and they are not substrings of (A,1,1,1); also note that (A,2,2,1,1), F's second current location, is also not a substring of one of (A,1,2,1), (A,2,1,1) or (A,2,2,2,1), and they are not substrings of (A,2,2,1,1). This means placing F as a descendant of D would

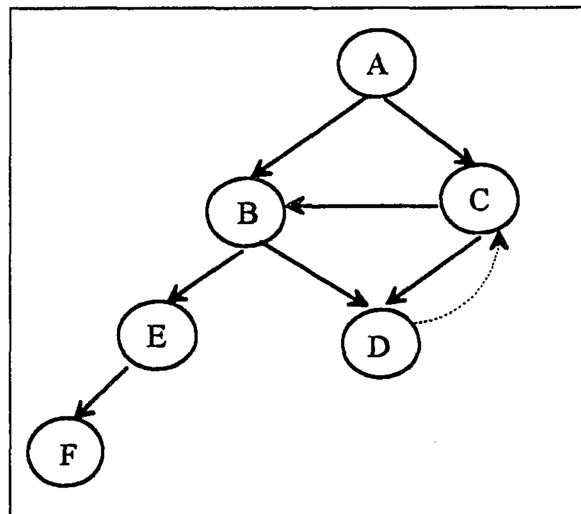
not create a cycle in the graph, as can be seen in Figure 5.10.7. Therefore, this relationship would be allowed.

The efficiency of this locations list should be compared with that of the locations list discussed earlier in this chapter. The benefits and drawbacks of each method should be examined and compared to decide which method is better for the implementation of the proposed system.



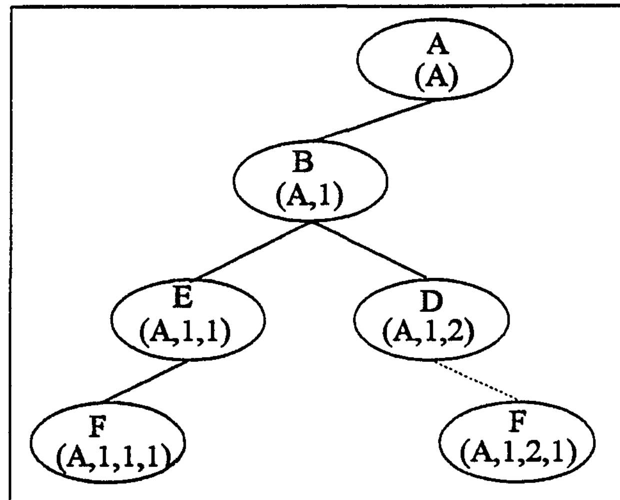
Adding a Child C to Node D in a Logical Tree

Figure 5.10.4



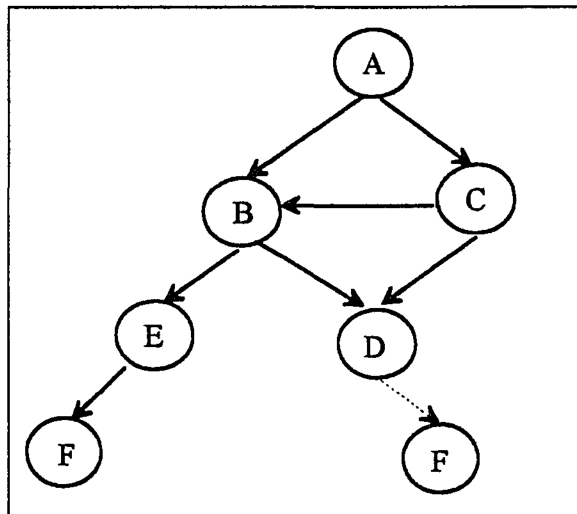
Adding a Child C to Node D in the Network -- A Cycle

Figure 5.10.5



Adding a Child F to Node D in a Logical Tree

Figure 5.10.6



Adding a Child F to Node D in the Network -- No Cycle

Figure 5.10.7

5.11 Using the System -- An Example

In this section, we will discuss how to use the proposed new system. We will continue to consider the same relatively small number of packages accessible to the entire system. A real-world system might very well be much larger.

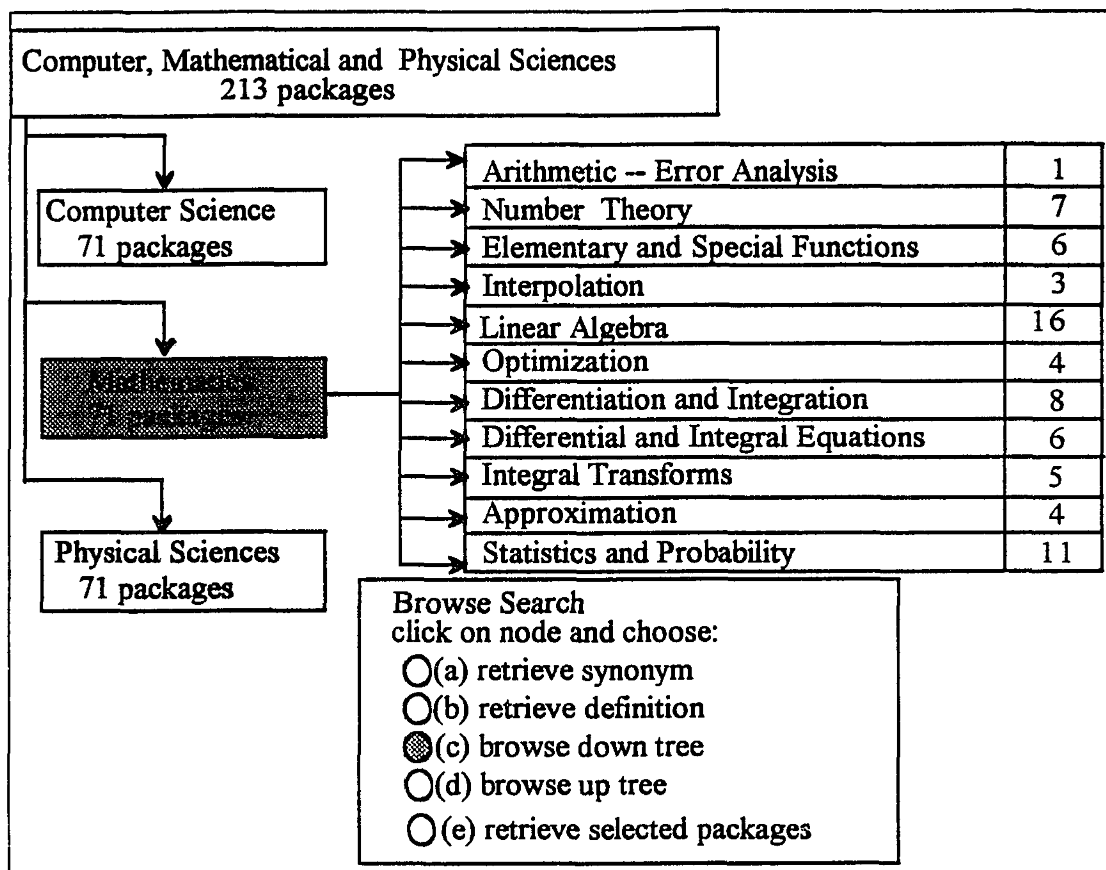
In Section 5.2 we mentioned how a keyword search in the current system will return anywhere from one package to 213 packages. An example of a keyword that returns one package is *Error Handling*. In our proposed system this keyword would be located at the terminal network node level and would return that same package. The major difference would come when dealing with keywords that return too broad a spectrum of packages. The keyword that returns 213 packages is *Computer, Mathematical and Physical Sciences*. This keyword, in our system, would be subdivided into three parts: Computer Science, Mathematics and Physical Science.

Assuming that there is the same number of packages in each of the three subdivisions, choosing one of the three would cut back the number of returned packages to 71. Assume a user is interested in Mathematics. In our new system the user might find the children of Mathematics to be Arithmetic--Error Analysis, Number Theory, Elementary and Special Functions, Linear Algebra, Interpolation, Solution of Nonlinear Equations, Optimization, Differentiation and Integration, Differential and Integral Equations, Integral Transforms, Approximation, and Statistics and Probability. Instead of being overloaded

with 213 packages, a user will now have a choice of accepting all the packages located below this node or browsing down the logical tree to find a smaller and more specific group of packages to examine.

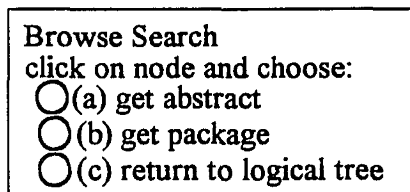
Figure 5.11.1 shows how a user, after expanding *Computer, Mathematical and Physical Sciences* might choose *Mathematics* in browse mode. The user, seeing a return of 71 packages, may then choose to browse down the logical tree emanating from *Mathematics*. Note that the user is able to see how many packages will be returned at any point; thus, at any node the user can continue browsing until a leaf node is reached or choose to retrieve the list of all descendant packages from this node. When a chosen node is a leaf node, the Browse Search box will look like Figure 5.11.2

This is an example of how keyword searching would operate in this scenario. It is easy to see how browsing down the logical tree twice makes the outcome of this search much more manageable than it is at present. Section 5.11 discusses the issue of performance.



User View of Expanding Nodes

Figure 5.11.1



Browse Search Box after Retrieve Selected Packages

Figure 5.11.2

5.12 Performance

Unlike SUCat, which has a static list of keywords, our keyword list will be dynamic. This will allow the system to evolve to satisfy the growing needs of the software community. Adding new keywords can also have a positive effect on the number of packages returned by the system (meaning fewer packages will be returned), as will be seen.

When initializing the network, care will be taken to make sure each non-leaf level node contains at least two children. A parent having just one child would have exactly the same leaf descendants as its child. This would indicate that the child's keyword should be a synonym to the parent's, and not a new keyword. Therefore, the system should be constructed to accept no less than two children as the first children of any node.

Thus, a logical subtree emanating from a node would be, at a minimum, a binary tree. If the entire tree were a complete binary tree, it would take $\log_2 N$ steps to traverse from a node with N leaf node descendants until the leaf level. Suppose there were M packages emanating from the node we are searching. Then, at each leaf node there would be approximately M/N packages to examine.

There are two steps to finding required software: First traverse down the subtree, and then search the packages of the leaf level node which has been chosen. Traversing eliminates many of the packages that need to be searched

(as has been noted in the example in Section 5.11). One keystroke would take a user one level down. Examining each package, to determine whether or not it is useful to us, is a time consuming job. Therefore a hierarchical approach, which divides broad categories into many subcategories (each with fewer packages than the broad category from which it stems), has much to offer.

In conclusion, the more children each node has, the faster one would arrive at the leaf level. This would be the case, since many children at each node would cause the search tree to be wider and shorter, and a shorter tree requires fewer searches to reach leaf level. Our concentration however should be on increasing N (the total number of leaf descendants). The greater the value of N, the fewer packages must be searched. In other words, if we subdivide nodes with general keywords into other nodes with very specific areas, our search time can be vastly improved.

5.13 Summary

In this chapter we discussed several problems which have been found in the traditional indexing schemes used to access information. We then proposed a new taxonomy for a dynamic classification scheme for storing reusable software. We showed that our proposed scheme makes retrieving software much more efficient than it was previously.

Some areas open to further research are discussed in the following chapter.

6. Further Research

6.1 Introduction

Research in the field of software union cataloguing is just at its infancy. As such, there are many topics open to further research.

This chapter suggests further research in the issues mentioned below:

- Automatic generation of MARC records (see Section 6.2)

- Distributed vs. centralized software union cataloguing (see Section 6.3)

6.2 Automatic Generation of USMARC Records

In Chapter 4 we discussed how MARC records are generated semi-automatically. That is, once the information necessary for creating a MARC record was extracted from a software repository, everything else (i.e., inserting the data into an internal MARC structure and the creation of the actual MARC record) was done automatically. See Appendix 4 for details on available procedures.

What remains to be done, so that MARC records can be generated in a totally automated manner, is to develop a tool capable of getting the desired information out of any software repository. For total automation, the tool would have to determine the type of repository it is searching, and the exact location and format of the information it has to extract. In this way, the tool can search the data and pick out appropriate pieces of information and create a MARC record from it. Most current RSLs do not store their information using any strict set of rules, which would allow such a tool to interpret their data. Because of this, preliminary work seems to indicate that some human intervention will be needed in this process.

6.3 Distributed vs. Centralized Software Union Cataloguing

A Software Union Catalogue is analogous to the union catalogue of a consortium of libraries, which tells the user in which library each published book or journal can be located. For example, the HPCC-Software Union Catalogue (HPCC-SUCat) contains various pieces of information on the union of all packages located in the repositories which belong to the HPCC Logical Library System. The use of such a catalogue allows researchers to locate software programs or packages over the set of heterogeneous distributed repositories. Maintenance of software union catalogues is vital. When new packages are installed in member repositories or when a new repository joins, appropriate information about the packages must be included in the catalogue. A union catalogue can store its information in one of three ways:

- The information can be stored centrally, as is the case with the HPCC-SUCat. To update a centralized system, a program would have to run periodically to check all member (both old and new) repositories for new packages. MARC records and other pertinent information would have to be generated for all new packages in all repositories and placed into the union catalogue.

- The union catalogue can be a virtual catalogue which would work as follows: Each repository has its own catalogue. The virtual or distributed catalogue has access to each member's catalogue. When a

user attempts to access information from the union catalogue, the catalogue system will access all the repository catalogues (in parallel) and respond with the union of all responses coming to it. In this situation updating is not necessary from within the union catalogue. It is the individual repositories that are responsible for this task.

- The union catalogue can be a combination of the two methods mentioned above. Some information may be centrally stored. Other information may be stored at individual repositories, to be accessed by the system when necessary. Users of the union catalogue would be unable to detect how the information is actually being stored. Updating the information would depend on whether the union catalogue has the information stored centrally or accesses it from individual repositories.

Several points might sway a decision as to whether the union catalogue should be centralized, distributed or combined.

First, two types of repositories exist. One contains packages which may be replicated in other repositories. We will refer to this type as R repositories. The other is software repositories from project databases (e.g., DoD projects), which do not contain replication because of the private nature of these projects. We will refer to these as NR repositories.

When replication exists, as is the case with SUCat, the union catalogue system must find a method of displaying just one copy of the replicated package. For example, SUCat deals with the problem of replication by assigning a unique identifier to each package. If the package is located in more than one place, the same unique identifier (UI) identifies the package in each location. For each UI, SUCat maintains a locations list. This is a list of all the different repositories which contain this particular package. When a user requires a copy of the software requests locations, a hyper-table of all repositories containing the software is produced. Then a user can select one of these repositories and retrieve the software from there.

As previously mentioned the current version of SUCat is centralized. In a centralized system all package information is stored and controlled on one machine. In this way, duplication can be avoided in the returned packages by inspecting the packages before they are inserted into the central system. Packages that are located in more than one repository enlarge the location list of the package's unique identifier, rather than being included many times in the list of packages in the catalogue. This may be considered an advantage to maintaining a centralized union catalogue.

On the other hand, a distributed union catalogue requires the member software repositories to store the data and do the searching for appropriate packages.

The data returned by the various repositories would be collected within the catalogue and would be made available to the user.

Distributed software union catalogue systems whose members are NR repositories can accomplish this by assigning an id number to each package that is returned. The id number would be the concatenation of the repository name with the repository id number. This type of id number would eliminate the need for a locations list. For example, if a piece of software is found in repository X and its package id# is 123456, then in the union catalogue it will be referred to as X-123456. To retrieve the package, the union catalogue system would extract the repository name and id# from the package id# and send a message to the appropriate repository to return the identified package. The union catalogue system would then return the appropriate package to the user.

Distributed software union catalogue systems whose members are both R and NR repositories would treat the NR repositories in the same way mentioned above. For the R repositories the distributed catalogue system would maintain a locations list just like the centralized version. However, the distributed version would require the id numbers assigned to packages to have a unique prefix, which the union catalogue system would recognize. When this prefix is recognized, the union catalogue system would retrieve locations from the locations list rather than extracting the name of the repository.

One important question to ask is the following: how large is the union catalogue? For example, SUCat has, at this time, software information from four repositories. There are over 200 packages in these repositories. The searching time on 200 items is minimal. However, if many repositories were to join such a system, and if all information on all packages were to be centralized, a search on a vast number of items might take an excessive amount of time. On the other hand, the search time for a distributed union catalogue, where each repository is queried in parallel, will be equal to the time it takes to search the largest (or most inefficiently organized) repository plus system response time. Hopefully, system response time would be kept to a minimum.

Section 5.3 illustrates other advantages of using a distributed union catalogue with some indexing schemes. However, there is still no optimal solution to the question of centralized vs. distributed. Thus, further research into centralized vs. distributed union cataloguing might prove advantageous.

6.4 Summary

This thesis established a method by which distributed heterogeneous software repositories can share assets with the use of a data independent data structure. Then a new taxonomy, for a dynamic classification scheme for classifying reusable software, was designed. Algorithms for inserting new packages and new nodes into the classification data structure were presented. During the course of this research several issues arose which require further investigation. These issues have been discussed in this chapter.

Interoperability among heterogeneous distributed reusable software libraries is a reality. However, there is still a great deal of work to be accomplished in this area.

Appendix 1: Netlib

Creating MARC records for repositories which store their information as text files is done as follows:

- First, get a copy of the repository's index file. This file usually has the names of all the packages included in the repository, a short description of each package and perhaps some other information as well that would be useful in a MARC record. To get Netlib's index file send e-mail to netlib@ornl.gov with the message "send index." Within a few minutes loads of information will be returned to you via e-mail. Here is a copy of what was received October 5, 1993:

Return-Path: <@CUNYVM.CUNY.EDU:netlibd@NETLIB1.EPM.ORNL.GOV> Received: from CUNYVM (NJE origin SMTP@CUNYVM) by CUNYVM.CUNY.EDU... To: MRSBC@cunyvm.cuny.edu Subject: Re: please send index 1 <lots more here> ...

-------quick summary of contents---------
a - approximation algorithms
alliant - set of programs collected from Alliant users
amos - special functions by D. Amos. = toms/644
apollo - set of programs collected from Apollo users
benchmark - various benchmark programs and a summary of timings
bib - bibliographies bihar - Bjorstad's biharmonic solver
blas - machine constants, vector and matrix * vector BLAS
bmp - Brent's multiple precision package
c - another "misc" library, for software written in C
. . .
voronoi - Voronoi diagrams and Delaunay triangulations
xnetlib - X windows interface to netlib
y12m - sparse linear system (Aarhus)
--------a bit more detail--------
The first few libraries here are widely regarded as being of high quality. The likelihood of your encountering a bug is relatively small; if you do, we certainly want to hear about it!
mail ehg@research.att.com

lib a
for approximation algorithms
editor Eric Grosse
master research.att.com

lib alliant
for programs collected from Alliant users
editor Jack Dongarra
master ornl.gov

lib amos
for Bessel functions of complex argument and nonnegative order
by D.E. Amos
ref ACM TOMS 12 (1986) 265-273
algorithm 644

```
master ornl.gov
# The Bessel functions H1, H2, I, J, K, and Y, as well as the
# Airy functions Ai, Bi, and their  derivatives are provided.
# Exponential scaling and sequence generation are optional.

lib     ampl/models
for     example model and data files for linear and nonlinear programming.
#       available only from netlib@research.att.com
editor  David Gay
master  research.att.com

lib     apollo
for     programs collected from Apollo users.
editor  Jack Dongarra
master  ornl.gov

lib     benchmark
for     contains benchmark programs and the table of Linpack timings.
editor  Jack Dongarra
master  ornl.gov

lib     bib
for     bibliographies:  Golub and Van Loan, 2nd ed.
editor  Eric Grosse
master  research.att.com

lib     bihar
for     biharmonic equation in rectangular geometry and polar coordinates
by      Petter Bjorstad
master  nac.no

lib    blas
for    blas (level 1, 2 and 3) and machine constants
rel    excellent
age     stable
editor  Eric Grosse
master  research.att.com

lib     bmp
for     Brent's multiple precision package
master  research.att.com

lib        c
for     miscellaneous codes written in C
#       Not all C software is in this "miscellaneous" library.
#       If it clearly fits into domain specific library, it is assigned there.
editor  Eric Grosse
master  research.att.com

... etc.

lib     voronoi
for     Voronoi regions and Delaunay triangulations
editor  Eric Grosse
```

~~~
master          research.att.com

lib             xnetlib
for             X Windows netlib file retreival application
editor          Reed Wade (wade@cs.utk.edu)
master          ornl.gov

lib             y12m
for             sparse linear systems
by              Zahari Zlatev, Jerzy Wasniewski and Kjeld Schaumburg
ref             Springer LNCS
# Comp Sci; Math Inst; Univ Aarhus; Ny Munkegade; DK 8000 Aarhus
master ornl.gov
CUT HERE........... cat > index <<'CUT HERE...........'
`````````````````````````````````````````````````````````````````````````````````
~~~

The part of the above document called *a bit more detail* was used to form a general index for Netlib. This index was scanned by a program which created MARC records for each of the libraries in the index. The index looks like this:

~~~
`````````````````````````````````````````````````````````````````````````````````

lib             a
for             approximation algorithms
editor          Eric Grosse master research.att.com

lib             alliant
for             programs collected from Alliant users
editor          Jack Dongarra master ornl.gov

lib             amos
by              D.E. Amos
ref             ACM TOMS 12 (1986) 265-273
algorithm       644
 master         ornl.gov
for             Bessel functions of complex argument and nonnegative order
# The Bessel functions H1, H2, I, J, K, and Y, as well as the
# Airy functions Ai, Bi, and their  derivatives are provided.
# Exponential scaling and sequence generation are optional.

lib             ampl/models
for             example model and data files for linear and nonlinear
# programming. available only from netlib@research.att.com
editor          David Gay
master          research.att.com

lib             apollo
for             programs collected from Apollo users.
editor          Jack Dongarra
master          ornl.gov

lib             benchmark
for             contains benchmark programs and the table of Linpack
# timings.
editor          Jack Dongarra
~~~

```
master      ornl.gov

lib         bib
for         bibliographies:  Golub and Van Loan, 2nd ed.
editor      Eric Grosse
master      research.att.com

lib         bihar
for         biharmonic equation in rectangular geometry and polar
# coordinates
by          Petter Bjorstad
master      nac.no

lib         blas
for         blas (level 1, 2 and 3) and machine constants
rel         excellent
age         stable
editor      Eric Grosse
master      research.att.com

lib         bmp
for         Brent's multiple precision package
master      research.att.com

lib         c
for         miscellaneous codes written in C
#Not all C software is in this "miscellaneous" library.
#If it clearly fits into domain specific library, it is
# assigned there.
editor      Eric Grosse
master      research.att.com

... etc...

lib         voronoi
for         Voronoi regions and Delaunay triangulations
editor      Eric Grosse
master      research.att.com

lib         xnetlib
for         X Windows netlib file retreival application
editor      Reed Wade (wade@cs.utk.edu)
master      ornl.gov

lib         y12m
for         sparse linear systems
by          Zahari Zlatev, Jerzy Wasniewski and Kjeld Schaumburg
ref         Springer LNCS
# Comp Sci  Math Inst   Univ Aarhus; Ny Munkegade; DK 8000 Aarhus
master      ornl.gov
```

- To create the MARC record the program must associate the information with a MARC tag number. This correspondence is made in a table that is included in the program as follows:

```
struct infotable{
char name[9];
char tag[4];
char bit[3];
char subf[2];
};

struct infotable pkg[] =          /* corresponding RIG Fields */
{{"lib",    "204",  " ","a"},    /* title                    */
 {"for",    "520",  " ","a"},    /* abstract                 */
 {"#",      "XXX",  " ","a"},    /* continuation char        */
 {"type",   "901",  " ","a"},    /* type of software         */
 {"by",     "100",  " ","a"},    /* author                   */
 {"prec",   "936",  " ","a"},    /* precision                */
 {"see",    "945",  " ","a"},    /* see also                 */
 {"master", "954",  " ","a"},    /* was created by           */
 {"editor", "949",  " ","b"},    /* support contact          */
 {"ref",    "939",  " ","a"},    /* reference                */
 {"age",    "904",  " ","a"},    /* age                      */
 {"lang",   "912",  " ","a"},    /* computer language        */
 {"rel",    "940",  " ","a"},    /* reliability              */
 {"alg",    "924",  " ","a"}};   /* history                  */
```

See Figure 5.2.1 for an example of a completed MARC record. Fields such as ASSET-ID (tag 001), TYPE (tag 901) and SUBJECT CLASSIFICATIONS (tag 650) have been added into all MARC records.

Appendix 2: GAMS

GAMS stores its software information in a relational database. The information is stored in ten relations or tables. The three relations pertaining to application packages, AP, APX and APHIST, are given below (information copied from [Bo89]):

The AP Relation

The AP relation contains machine-independent data which describes each package. Exactly one row appears in the AP relation for each package known to the database. The attribute AP# is a key for the AP relation.
The attributes of the AP relation are:

1. **AP#** (Integer;positive)
   The unique identifier for this package.
2. **AP** (Text,12 characters; upper case)
   The name of this package.
3. **TYPE** (Integer; 1,2,3,4,5 or 6)
   Indicates how the package is organized:
   1=> subprogram library (not divided into sublibraries),
   2=> partitioned subprogram library (library divided into sublibraries),
   3=> homogeneous collection of stand-alone programs (i.e., with a common input syntax),
   4=> collection of commands in an interactive system,
   5=> interactive program,
   6=> heterogeneous collection of stand-alone programs,
   Examples of each type: (1)IMSL, (2) CMLIB, (3) BMDP, (4) Dataplot, (5) MATLAB, (6) Collected Algorithms of the ACM. The difference between 4 and 5 is that the individual commands within Dataplot have been classified using the GAMS Classification Scheme, while those within MATLAB have not.
4. **PORT** (Text, 1 character; E, H, M, or P)
   Indicates restrictions on library usage and ease of transporting the software to other machines:
   E=> Portable; H=> Portable some conversion required;
   M=>machine-specific; P=>proprietary (use of this software is governed by a licensing agreement).
5. **DESC** (Text, variable length; free format)
   A brief description of this package.
6. **LANG** (Text, 12 characters; upper case)
   The computer language in which this package is coded.
7. **DEVELOP** (Text, variable length; free format)
   The name and address of the organization where the library was developed and the name of a contact there.

8. **CITATION** (Text, variable length; free format)
   A reference to a monograph or technical article which describes the package.
9. **DISTRIB** (Text, variable length; free format)
   The name and address of the organization with is currently distributing the software.

The APX Relation

The APX relation contains information about the implementation of packages on each computer system where they are available. Each row corresponds to the implementation of a version of a package on a particular computer system. For example, if a given package is available on three computer systems, there will be three rows in the APX relation corresponding to this package; they will have the same AP#, but different COMP#. More precisely, the attribute pair (AP#, COMP#) is a key for the APX relation.

The attributes of the APX relation are:

1. **AP#** (Integer;positive)
   The unique identifier for this package. This determines the row of the AP relation that provides machine-independent information about this package.
2. **AP** (Text,12 characters; upper case)
   The name of this package corresponding to AP#.
3. **COMP#** (Integer; positive)
   The unique identifier for the computer on which this package has been implemented.
4. **COMP** (Text,6 characters; upper case)
   The name of the computer corresponding to COMP#.
5. **SUPP** (Text,1 character; 1,2 or 3)
   The level of support provided users of the library on this computer: 1=> full support; 2=> limited support; 3=> no formal support.
6. **ACCESS** (Text, variable length; free format)
   Command(s) which access this library on a given computer. Name substitution may be used.
7. **VER#** (Integer; positive)
   A unique identifier for this implementation of the package.
8. **VER** (Text, 6 characters; free format)
   The version name, number or date, of this implementation.
9. **LIBDOC** (Text, variable length; free format)
   Command(s) which retrieve detailed documentation for this implementation of the package on the named computer.
10. **MODDOC** (Text, variable length; free format)
    Command(s) which retrieve detailed documentation for a module in this package on the named computer.
11. **CITATION** (Text, variable length; free format)

A reference to a monograph or technical article which describes this particular implementation of the package.

12.**SAMPLE** (Text, variable length; free format)
A system command which will retrieve a sample usage of modules in this package.

13.**SOURCE** (Text, variable length; free format)
A system command on the named computer which can be used to retrieve the source for modules in the package.

14.**TESTS** (Text, variable length; free format)
A system command which will retrieve test programs for the modules in this package.

The APHIST Relation

This relation is used to keep a record of which versions of each application package are or have been implemented on each computer. For each row in the APX relation there will be one or more rows in the APHIST relation. Each row corresponds to the implementation of a particular version of a package on a single computer. The dates the version was installed and superseded are noted. If the date superseded is null, then that particular version is currently active. The attribute triple (AP#, COMP#, VER#) is a key for the APHIST relation.

The attributes of the APHIST relation are:

1. **AP#** (Integer;positive)
The unique identifier for this package.
2. **AP** (Text,12 characters; upper case)
The name of this package corresponding to AP#.
3. **VER#** (Integer; positive)
The unique identifier associated with this version of the package on the named computer.
4. **VER** (Text, 6 characters; free format)
The name of the version associated with VER#.
5. **COMP#** (Integer; positive)
The unique identifier for the computer on which this version of the package has been implemented.
6. **COMP** (Text,6 characters; upper case)
The name of the computer corresponding to COMP#.
7. **DATEINT** (Integer, 1 unit)
The date this version of the package was introduced. Format is YYMMDD.
8. **DATEDEL** (Integer, 1 unit)
The date this version of the package was superseded. Format is YYMMDD. If this is null, then this version is currently active.
9. **CITATION** (Text, variable length; fee format)
A reference to a monograph or technical article which describes this particular implementation of the package.

An SQL-like command is required to extract information from GAMS. The following commands are used to extract application package information from the GAMS database.

```
OPEN GAMS WIDTH 132
OUTPUT output
SELECT AP# AP TYPE PORT DESC=40 LANG +
 FROM AP WHERE AP# EQ @ap#
SELECT DEVELOP=40 CITATION=40 DISTRIB=40 +
 FROM AP WHERE AP# EQ @ap#
SELECT COMP# COMP SUPP ACCESS=40 VER# VER LIBDOC=40 +
 FROM APX WHERE AP# EQ @ap# AND COMP# EQ @comp# AND VER#
EQ @ver#
SELECT MODDOC=40 CITATION=40 SAMPLE=40 +
 FROM APX WHERE AP# EQ @ap# AND COMP# EQ @comp# AND VER#
EQ @ver#
SELECT SOURCE=40 TESTS=40 +
 FROM APX WHERE AP# EQ @ap# AND COMP# EQ @comp# AND VER#
EQ @ver#
SELECT DATEINT DATEDEL CITATION +
  FROM APHIST WHERE AP# EQ @ap# AND COMP# EQ @comp# AND
VER# EQ @ver#
SELECT LOCATION=30 CONTACT=30 OPSYS=30 COMPILER=30 +
 FROM COMPUTER WHERE COMP# EQ @comp#
BLANK 1
EXIT
```

To create a MARC record from the information extracted from the GAMS database, the program must associate the information with a MARC tag number. This correspondence is made in a table that is included in the program as follows:

```
typedef struct infotable *infotabptr;
struct infotable{
    char name[9];
    char tag[4];
    int len;
    char bit[3];
    char subf[2];
    char *contents;
};
```

```
struct infotable pkg[NUMPKGEL] =
   {{"AP\#",     "999", 11,       "01","b"},
    {"AP",       "204", 13,       "11","a"},
    {"TYPE",     "901", 11,       " ","b"},
    {"P",        "931",  2,       "11","a"},
    {"DESC",     "520", MAXTEXT, "00","a"},
    {"LANG",     "912", 13,       "10","a"},
    {"DEVELOP", "260", MAXTEXT, "11","b"},
    {"CITATION","939", MAXTEXT, "11","a"},
    {"DISTRIB", "920", MAXTEXT, "00","a"},
    {"COMP\#",  "999", 11,       "10","a"},
    {"COMP",    "926",  7,       "11","a"},
    {"S",       "949",  2,       "00","a"},
    {"ACCESS",  "947", MAXTEXT, "11","a"},
    {"VER\#",   "999", 11,       "00","v"},
    {"VER",     "952",  7,       "00","a"},
    {"LIBDOC",  "939", MAXTEXT, " ","a"},
    {"MODDOC",  "939", MAXTEXT, " ","a"},
    {"CITATION","939", MAXTEXT, "11","a"},
    {"SAMPLE",  "950", MAXTEXT, "11","a"},
    {"SOURCE",  "947", MAXTEXT, "00","a"},
    {"TESTS",   "950", MAXTEXT, " ","a"},
    {"DATEINT", "952", 11,       "00","b"},
    {"DATEDEL", "999", 11,       "00","c"},
    {"CITATION","939", MAXTEXT, "11","a"},
    {"LOCATION","999", MAXTEXT, "00","d"},
    {"CONTACT", "999", MAXTEXT, "11","e"},
    {"OPSYS",   "912", MAXTEXT, "11","b"},
    {"COMPILER","912", MAXTEXT, "00","c"}};
```

Appendix 3: StatLib

Creating MARC records for repositories which store their information as text files is done as follows:

- First, get a copy of the repository's index file. This file usually has the names of all the packages included in the repository, a short description of each package and perhaps some other information as well that would be useful in a MARC record. To get StatLib's index file send e-mail to statlib@lib.stat.cmu.edu with the message "send index." Within a few minutes, approximately 5 pages of information will be returned to you via e-mail. Here is a copy of some of what was received October 5,1993:

\\

Return-Path: <@CUNYVM.CUNY.EDU:statlibd@TEMPER.STAT.CMU.EDU> ...Tue, 5 Oct 93 09:56:44 -0400 Date: Tue, 5 Oct 93 09:56:44 -0400 From: Statlib Server <statlibd@stat.cmu.edu> Message-Id: <9310051356.AA05558@temper.stat.cmu.edu> To: MRSBC@CUNYVM.CUNY.EDU Subject: Subject: send index
===== general StatLib index =====
. . . etc.
------quick summary of contents--------
apstat - Selected algorithms transcribed from Applied Statistics
asacert - The ASA certification proposal and discussion.
asascs - Material related to the Statistical Computing
Section of the American Statistical Association.
blss - Macros and fixes for the BLSS statistical package
. . .
S - S functions, device drivers and related software.
s-news - Archives of the S-news mail, in digest format.
sapaclisp- Common Lisp functions for spectral analyis
xlispstat- Luke Tierney's XlispStat system for Unix systems,
updated March 1991.
1993.expo- Data for the 1993 ASA Graphics and Data Exposition.
-------Longer summary of contents--------
apstat The apstat collection contains a nearly complete set of algorithms published in Applied Statistics. The collection is maintained by Alan Miller in Melbourne. Many of the algorithms came directly from the Royal Statistical Society or the authors, but many have also been entered by hand. See also the griffiths-hill collection.

asacert The asacert collection contains the ASA proposal for certification of statisticians and the archives of a mailing list discussion of the topic.

asascs This small collection contains the charter for the ASA Statistical Computing Section and some software developed by the section for searching the electronic version of the Current Index to Statistics.

blss This stub entry will contain submissions of code for the BLSS software package.

. . .

.S Software and extensions for the S (Splus) language. Over 130 separate packages including many novel statistical ideas.

s-news The s-news archive contains the mail messages sent to the S-news mailing list. The messages are grouped together in about 50 kbyte chuncks. To subscribe to the S-news mailing list, send a message to s-news-request@utstat.toronto.edu. A human will respond to you.

sapaclisp Sapaclisp is a collection of Common Lisp functions that can be used to carry out many of the computations described in the book "Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques, by Donald B. Percival and Andrew T. Walden, Cambridge University Press, Cambridge, England, 1993

xlispstat Lisp-Stat is an extensible environment for statistical computing and dynamic graphics based on the Lisp language. XLISP-STAT is a version of Lisp-Stat based on a dialect of Lisp called XLISP. The source code for XLISP-STAT and contributed code for use with the system are available from this archive.

statlibd STAT 10/05/93 Statlib Server MRSBC@CUNYVM.CUNY.E 10/05/93 Subject: send index

When this file was received originally, the longer summary of contents was edited to look like the file below. The format chosen matched the format of information sent from another repository, Netlib. Adopting the same format for this repository made creating MARC records a much simpler task. The programs written to create Netlib MARC records needed only slight modification to create MARC records for StatLib.

```
lib  apstat
for  Selected algorithms transcribed from Applied
#    Statistics
lib  asascs
for  Material related to the Statistical Computing
#    Section of the American Statistical Association.

lib  blss
for  Macros and fixes for the BLSS statistical package

. . .

lib  S
for  S functions, device drivers and related software.

lib  s-news
for  Archives of the S-news mail, in digest format.

lib  xlispstat
for  Luke Tierney's XlispStat system for Unix systems,
#    updated March 1991.

lib  1993.expo
for  Data for the 1993 ASA Graphics and Data Exposition.
```

Please note that StatLib has only two pieces of information in its index file: The library name (lib) and a brief description or abstract (for). The information above also includes a continuation character (#) when the abstract is more than one line long.

- To create the MARC record the program must associate the information with a MARC tag number. This is done in a table that is included in the program as follows:

```
struct infotable{
 char name[9];
 char tag[4];
 char bit[3];
 char subf[2];
};

struct infotable pkg[] =
{{"lib",   "204",  "  ","a"},  /* title                  */
 {"#",     "XXX",  "  ","a"},  /* continuation character  */
 {"for",   "520",  "  ","a"}}; /* abstract               */
```

See Figure 5.2.1 or Appendix 5 for examples of MARC records. Fields such as ASSET-ID (tag 001), TYPE (tag 901) and SUBJECT CLASSIFICATIONS (tag 650) have been added into all MARC records.

Appendix 4: marclib

marclib.h is a library of C functions. The functions work well together and can be used to compose full programs to create, manipulate and output MARC records. In the spirit of this thesis, the library was created with the hope of saving time by reusing code. It has.

One of the header files included in marclib.h is marc_hd.h. It is included below to make marclib.h functions easier to understand.

```
/*marc_hd.h*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define FIELD_TERM   0x1e
#define RECORD_TERM  0x1d
#define SUBF_DEL     0x1f
#define MAXLINE      200
#define TAGLEN       3
#define FIELDLEN     4
#define STARTPOSLEN  5
#define INDICATORLEN 2
#define SUBFLEN      2
#define FIELDTERMLEN 1
#define PACKAGELEN   7
#define LEADER_LEN   24
typedef struct _info *infoptr;
typedef struct _sub_field *subfptr;
typedef struct _record *recordptr;
typedef struct _var_field *varptr;
typedef struct _legend{
  char type_of_rec[2];       /* 06    */
  char bibliog_level[2];     /* 07    */
  char undefined[3];         /* 08-09 */
};
typedef struct _entry_map{
  char len_field_len[2];     /* 20    */
  char len_start_pos[2];     /* 21    */
  char len_imple_defined[2]; /* 22    */
  char undefined2[2];        /* 23    */
};
typedef struct _leader{      /* BYTE# */
  char log_rec_len[6];       /* 00-04 */
  char rec_status[2];        /* 05    */
```

```
  struct _legend legend;      /* 06-09 */
  char indicator_count[2];   /* 10    */
  char subf_code_count[2];   /* 11    */
  char data_base_addr[6];    /* 12-16 */
  char encoding_level[2];    /* 17    */
  char desc_cat_form[2];     /* 18    */
  char linked_rec_code[2];   /* 19    */
  struct _entry_map entry_map;/* 20-23 */
};
typedef struct _record{
  struct _leader leader;
  infoptr infoblock;
};
typedef struct _element{
  char tag[4];
  char field_len[5];
  char start_pos[6];
};
typedef struct _sub_field{
  char subfield[2];
  char *contents;
  struct _sub_field *next_data;
};
typedef struct _var_field{
  int data_control; /* 1 if data field, 0 if control field */
  char bits[3];
  struct _sub_field *sub_field; };
typedef struct _info {
  struct _element element;
  struct _var_field *var_field;
  int length;
  infoptr next;
};
struct _record marcrec;
char leader[25];
```

The following is the index file, marclib.idx, for marclib.h. It describes the function of the subroutines included in marclib.h. In the index below, a * inside function parentheses (e.g., function_name(*)) indicates the function is included in marclib.h. This version of marclib.h is found at /ndads/db_dev/bjacobs/mindy/marctool2.

```
/*marclib.idx*/

#include "marc_hd.h"
```

```
#include "form.tmp"
#include "rig.tmp"
#include "marc.tmp"

#define LINE "===\t==========\t\t========\n"

initialize(infoblock)
infoptr *infoblock;
abs: initializes *infoblock to NULL

makeleader(marcrec,base_addr,directory_len)
struct _record *marcrec;
int *base_addr, *directory_len;
abs: creates a leader for marc record and figures out record
#   length
calls: find_logical_rec_len(*),itoa2(*)
call after: openmarc_read(*)

find_logical_rec_len(infoblock,directory_len)
infoptr infoblock;
int *directory_len;
abs: This procedure walks through the internal marc record
#   to find out its logical record length
calls: atoi()
call after: openmarc_read(*)
int notblanks (line)
char *line;
abs: This procedure returns 1 if line contains a non-blank
#   character, returns 0 if line is all blanks
calls: isspace(),strlen()

concat (ns, s1, s2)
char *ns, *s1, *s2;
abs: concatenates the contents of s1 & s2 and place into ns
calls: strlen()

int data_field(tag)
char *tag;
abs: This procedure checks to see if the tag corresponds to
#   a (VDF) variable data field--any tag not beginning with
#   "00" is a VDF.  IF tag is VDF tag, a 1 is returned.
#   Otherwise, the tag corresponds to a Variable Control
#   Field (VCF) and a 0 is returned.

void createrec(filename,infoblock)
```

```
char *filename;
infoptr infoblock;
abs: This procedure creates an external MARC record to be
#   placed into the file "filename" FROM the internal marc
#   record. Internal MARC must exist in order for this
#  procedure to work. Any time a new MARC record is created
#  or an old one is adjusted, this procedure must be called
#   either to create a new external MARC record or to
#   overwrite the existing one.
call after: openmarc_read(*),read_rest(*)
calls: fopen(), putc(), data_field(*), sprintf(), system(),
#     fflush(), printf()

itoa2(n,s,len)
char s[];
int n, len;
abs: this proc is a takeoff on itoa--it converts the
#    integer n to a char string s of specified length
#    len--0 filled on left. len includes the NULL char
#    itoa2(6,string,3) string then will = "06" or 06\0
calls: reverse(*)

reverse(s)
char s[];
abs: reverses string s
calls: strlen()

openmarc_read(filename,statfile,infoblock,directory_count)
char *filename;
FILE **statfile;
infoptr *infoblock;
int *directory_count;
abs: This procedure opens a MARC file "filename" and reads
#   the directory into the internal marc record. This
#   procedure keeps track of directory_count-- the number
# of tags in the directory. If file "filename" doesn't
#  exist an error message is placed in statfile and the
#      program exits.
calls: fopen(),fprintf(),getc(),malloc(),fclose(),sizeof()

read_rest (filename, statfile, infoblock, directory_count, directory_len,
base_addr)
char *filename;
FILE **statfile;
infoptr *infoblock;
```

```
int *directory_count, *directory_len, *base_addr;
abs: This procedure opens a MARC file "filename" and reads
#   the DATA portion of "filename" into the internal MARC
#   record. The directory must be read into the internal
#   MARC record before using this procedure. Also figures
#  out directory_len and base_address of data portion of
#      record.
call after: openmarc_read(*)
calls: fopen(), fprintf(), malloc(), atoi(), data_field(*),
#   fseek(), fflush(), exit(),strlen()

changetag(from,to,infoblock)
char *from, *to;
infoptr *infoblock;
abs: this procedure changes a tag # from "from" to "to". If
#  they are both data fields or both variable control
#  fields no changes are made to the data part of the
#  field. However, if we are changing from one type to the
#  other accommodations are made for the INDICATORS &
#      SUBFIELDS.
call after: openmarc_read(*) and read_rest(*)
calls: find_taginfo(*), fix_data_field(*), data_field(*),
#     strcpy()

infoptr find_taginfo(from,infoblock)
char *from;
infoptr infoblock;
abs: returns a pointer to the info whose tag equals from
call after: openmarc_read(*) and read_rest(*)
calls: strcmp()

void fix_data_field (p, from_type, to_type,infoblock)
infoptr p;
int from_type, to_type;
infoptr *infoblock;
abs: if from_type != to_type field length must be changed in
#   this subroutine -- also startpos of all information
#   after this field must be updated.
called by:  changetag(*)
calls: atoi(),itoa2(*),fix_startpos(*),strcpy()

fix_startpos(p,i,infoblock)
infoptr p;
int i;
infoptr *infoblock;
```

```
abs: goes through the entire internal MARC record. Any tag
#  whose start position is > than p's start position--comes
#  after p--will have its start position increased by i (i
#   may be a negative number)
called by: fix_data_field(*)--can also be used independently
calls: atoi(),itoa2(*),fix_startpos(*),strcpy()

print_form(outfile,statfile,template,temp_length,infoblock)
FILE **outfile, **statfile;
char *template[];
int temp_length;
infoptr infoblock;
abs: the output of this procedure is dependent upon the
#   template which is one of the parameters. Output sent to
#   outfile will consist of marc record information
#   displayed in the specified template format.
called after: openmarc_read(*) and read_rest(*)
calls: find_taginfo(*), strcpy(), fputc(), fprintf()

print_by_line(outfile,string)
FILE **outfile;
char *string;
abs: takes string and prints it out at end of line--if
#   string is too long for one line, it is continued on
#   next line with proper indentation
called by: print_marc(*)
calls: fputc(), fprintf()

header(outfile)
FILE *outfile;
abs: prints header to readable marc records or when printing
#   just one field of the record.
called by: print_marc(*), print_tag(*)
calls: fprintf()

print_marc(outfile,statfile,infoblock)
FILE **outfile, **statfile;
infoptr infoblock;
abs: the output of this procedure is a readable marc record.
#   Output sent to outfile.
call after: openmarc_read(*) and read_rest(*)
calls: fprintf(), data_field(*), print_by_line(*),
#  get_field_name(*)

print_tag(p,outfile,statfile,tag)
```

```
infoptr p;
FILE **outfile, **statfile;
char *tag;
abs: the output of this procedure is the data portion of the
#   readable marc record specified by tag. Output sent to
#   outfile.
call after: openmarc_read(*) and read_rest(*)
calls: fprintf(),print_by_line(*),get_field_name(*)

char *get_only_tag(p,subf)
infoptr p;
char subf;
abs: returns the data portion of a MARC record corresponding
#  to a given ptr and subfield
call after: openmarc_read(*), read_rest(*)

char *get_field_name(tag, subf)
char *tag;
char subf;
abs: returns the character string that describes the job of
#   the tag and the subfield
called by: print_form(*), print_marc(*), print_tag(*)
calls: data_field(*), strcmp()

sort(infoblock)
infoptr *infoblock;
abs: when a MARC record's tags are not stored in the directory
#   in sorted order this procedure will correct the format
#   and sort. Must call redo_startpos(*) and then
#   createrec(*) after record is sorted.
call after: openmarc_read(*) and read_rest(*)
calls: strcmp()

redo_startpos(infoblock)
infoptr *infoblock;
abs: starting from the beginning of the internal MARC
#   record, this proc redoes the start position of each
#   tag. Must be called after sort(*), since positions are
#   changed after sorting.
call after: openmarc_read(*) and read_rest(*)
calls: atoi(),itoa2(*)

insertinfo(tag,length,bits,subfield,contents,infoblock)
char tag[],bits[],subfield[],contents[];
infoptr *infoblock;
```

```
int length;
abs: this procedure can be used a) to create an internal
#   MARC record from scratch. b)to insert a new field into
#   an existing marc record. The parameters tag,length,
#  bits, contents and infoblock are input parameters. The
#  contents may be retrieved from a database or text file;
#   tag, bits and subfield should be found by looking up
#   this info in a repository specific table, which links
#  the contents' field name to these pieces of information;
#   length can be worked out using strlen function. Each
#   call to insertinfo() places one field into the internal
#   MARC record. MUST be sure to call redo_startpos(*) and
#   createrec(*) after changing or creating a MARC record.
calls: strcmp(), strlen(), makenewinfo(*) ,malloc(),
#   concat(*), free(), data_field(*), strcpy()
call_after: If creating MARC record from scratch call after
#   initialize(*). If inserting new fields call after
#   openmarc_read(*) and readrest(*).

makenewinfo(ip,tag,length,bits,subfield,contents)
infoptr *ip;  char *tag, *bits, *subfield, *contents;
int length;
abs: allocates space for the data portion of a new
#   tag-subfield and initializes it.
calls: data_field(*), malloc(*), memset(), malloc(),
# strlen(), strcpy();
```

Appendix 5: marctool

Four tools have been created to extract information from MARC records. They are invoked by calls to marctool as follows:

- marctool 1 <entry-id> <outputfile> <statusfile>

This will read the MARC record specified by <entry-id> and place the RIG format of this record into the file <outputfile>. The status of the job is placed in <statusfile>.

- marctool 2 <entry-id> <outputfile> <statusfile>

This tool is identical except that the Submission Format of the MARC record is placed in <outputfile>.

- marctool 3 <entry-id> <outputfile> <statusfile>

This tool places a readable MARC record into <outputfile>.

- marctool 4 <entry-id> <outputfile> <statusfile> <tag>

This tool extracts the tag named by <tag> from the MARC record specified by <entry-id> and places it into <outputfile>.

The following is a MARC record from the GAMS Repository for the package called DATAPAC. Its entry-id# is NIST_000011.

01733n 2200241 4500000100130000020400130001326000840002652003560011065002570046690100340072391200260075792000970078392600120088093100300089293900890092293900890101193900890110094700590118994900180124895000530126695200190131999015301338#NIST_000011#1# 1=aDATAPAC# 1=bNIST, Statistical Engineering Division, Gaithersburg, MD 20899 (J.J. Filliben)# 1=aA Fortran subroutine library for probability distribution, density, percent point, and sparsity function evaluation; random number generation; line-printer plotting - histograms, scatter diagrams, probability plots; data manipulation; general statistical analysis; time series analysis; polynomial regression; ANOVA. (Approximately 170 subroutines.)# =aStatistics, Probability: Behavioral and Social Sciences; Measurement, Statistics and Evaluation; Statistics; Computer, Mathematical, and Physical Sciences; Applied Mathematics; Computer Science; Computational Methods; Numerical Analysis; Mathematics; # 2=bSubprogram Library =aPACKAGE# 3=aFortran=bNOS=cFTN5# 1=aS. Bremer, NIST, Bldg 101 Room A337, Gaithersburg, MD 20899 (FTS 879-2845 or 301-975-2845)# 1=a840NOS#1 =aSome Conversion Required #1 =aJ.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.# 1=aJ.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.#1 =aJ.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.# 1 =aATTACH,DPACMOD/UN=CAMLIB,NA.\$MODIFY(P=D PACMOD,...).# 1=aFull

Support# 1 =aATTACH,DATAPAC/UN=CAMLIB,NA.\SLIBRARY,DA TAPAC.# 2=b1986=a870401# 6=b8=a6=v2 =dCYBER 840, Consolidated Scientific Computing System, Boulder =eKatherine Pagoaga-320-5104, Linda Lindgren (CMLIB)-320-5149 =iGAMS-8-6-2#X

The same conventions are used here to display field terminators, subfield delimiters and record terminators as have been used in Section 4.2.

When the call "marctool 1 NIST_000011 <outputfile> <statusfile>" is made, the output is the following HPCC Reuse Software Submission Form:

HPCC Reuse Software Submission Files

TYPE: HPCC Software Union Catalogue;;
Title: DATAPAC ;;
Authors: ;;
Publisher: NIST, Statistical Engineering Division, Gaithersburg, MD 20899 (J.J. Filliben) ;;
Pubdate: ;;
Type: PACKAGE ;;
Parent: ;;
Id: NIST_000011 ;;
END_HEADING;;

ABSTRACT: A Fortran subroutine library for probability distribution, density, percent point, and sparsity function evaluation; random number generation; line-printer plotting - histograms, scatter diagrams, probability plots; data manipulation; general statistical analysis; time series analysis; polynomial regression; ANOVA. (Approximately 170 subroutines.) ;;
END_ABSTRACT;;

ATTRIBUTES:

NAME: Acceptance_Date;;
VALUE: ;;
NAME: Age;;
VALUE: ;;
NAME: Calls/Called_By;;
VALUE: ;;
NAME: Certification;;
VALUE: ;;
NAME: Change_Date;;
VALUE: ;;
NAME: Children;;
VALUE: ;;
NAME: Classification_Mechanism;;
VALUE: ;;
NAME: Compilation_Order;;
VALUE: ;;
NAME: Compliance_To_Standards;;
VALUE: ;;
NAME: Computer_Language;;

VALUE: Fortran ;;
NAME: Copyright;;
VALUE: ;;
NAME: Cost;;
VALUE: ;;
NAME: Cost_Information_Contact;;
VALUE: ;;
NAME: Data;;
VALUE: ;;
NAME: Demo;;
VALUE: ;;
NAME: Dependencies;;
VALUE: ;;
NAME: Derived_From;;
VALUE: ;;
NAME: Design_Spec;;
VALUE: ;;
NAME: Distribution_Statement;;
VALUE: S. Bremer, NIST, Bldg 101 Room A337, Gaithersburg, MD 20899 (FTS 879-2845 or 301-975-2845) ;;
NAME: Encrypted;;
VALUE: ;;
NAME: Feedback;;
VALUE: ;;
NAME: Feedback_From;;
VALUE: ;;
NAME: Feedback_From_Organization;;
VALUE: ;;
NAME: Format;;
VALUE: ;;
NAME: History;;
VALUE: ;;
NAME: Implementation_Citation;;
VALUE: ;;
NAME: Installation_Info;;
VALUE: 840NOS ;;
NAME: Interfaces;;
VALUE: ;;
NAME: Is_Composed_Of;;
VALUE: ;;
NAME: Is_Part_Of;;
VALUE: ;;
NAME: Kind;;
VALUE: ;;
NAME: Limitations_And_Constraints;;
VALUE: Some Conversion Required ;;
NAME: Makefile;;
VALUE: ;;
NAME: Matrics;;
NAME: National_Language;;
VALUE: ;;
NAME: Number_Of_Extractions;;
VALUE: ;;

```
NAME: Other;;
VALUE:  ;;
NAME: Precision;;
VALUE:  ;;
NAME: Problem_Report;;
VALUE:  ;;
NAME: Read_Me;;
VALUE:  ;;
NAME: Reference;;
VALUE: J.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD,
 1977. ;;
NAME: Reference_Number;;
VALUE:  ;;
NAME: Reliability;;
VALUE:  ;;
NAME: Requirements_Spec;;
VALUE:  ;;
NAME: Restrictions_Apply;;
VALUE:  ;;
NAME: Review;;
VALUE:  ;;
NAME: Reviewer;;
VALUE:  ;;
NAME: Security_Restrictions;;
VALUE:  ;;
NAME: See_Also;;
VALUE:  ;;
NAME: Size;;
VALUE:  ;;
NAME: Source_Code;;
VALUE: ATTACH,DPACMOD/UN=CAMLIB,NA.\$MODIFY(P=D PACMOD,...). ;;
NAME: Submission_Date;;
VALUE:  ;;
NAME: Supplemental_Information;;
VALUE:  ;;
NAME: Support;;
VALUE: Full Support ;;
NAME: Support_Contact;;
VALUE:  ;;
NAME: Support_Organization;;
VALUE:  ;;
NAME: Test_Suite;;
VALUE: ATTACH,DATAPAC/UN=CAMLIB,NA.\$LIBRARY,DA TAPAC. ;;
NAME: Userguide;;
VALUE:  ;;
NAME: Version;;
VALUE: 870401 ;;
NAME: Warranties;;
VALUE:  ;;
NAME: Was_Created_By;;
VALUE:  ;;
END_ATTRIBUTES;;

COMPONENTS:
```

```
  NAME: Descrs;;
  TITLE: Description;;
  SCRIPT:
   BEGIN:
   END;;
  NAME: Rigyp;;
  TITLE: Rig_Yellow_Pages;;
  SCRIPT:
   BEGIN:
   END;;
  NAME: Locats;;
  TITLE: Locations;;
  SCRIPT:
   BEGIN:
   END;;
  NAME: Suggs;;
  TITLE: Suggestions;;
  SCRIPT:
   BEGIN:
  END;;
END_COMPONENTS;;
INDEXES:
 NAME: Subject;;
 TERMS: Statistics, Probability: Behavioral and Social Sciences; Measurement, Statistics
and Evaluation; Statistics; Computer, Mathematical, and Physical Sciences; Applied
Mathematics; Computer Science; Computational Methods; Numerical Analysis;
Mathematics; ;;
END_INDEXES;;
```

When the call "marctool 2 NIST_000011 <outputfile> <statusfile>" is made, the output is the following RIG Yellow Pages listing of the MARC record:

```
                 ReUse Library Interoperability Group (RIG)
                          Technical Committee 2
                              Working Draft
(Listed   in   Alphabetical   Order)
*****************************************************************
Attribute : Acceptance Date
Value     :
--------------------------------------------------------------
Attribute : Age
Value     :
--------------------------------------------------------------
Attribute : Authors
Value     :
--------------------------------------------------------------
Attribute : Calls/Called By
Value     :
--------------------------------------------------------------
Attribute : Certification
Value     :
--------------------------------------------------------------
```

Attribute : Change Date
Value :

Attribute : Children
Value :

Attribute : Classification Mechanism
Value :

Attribute : Compilation Order
Value :

Attribute : Compliance to Standards
Value :

Attribute : Copyright
Value :

Attribute : Cost
Value :

Attribute : Cost Information Contact
Value :

Attribute : Data
Value :

Attribute : Demo
Value :

Attribute : Dependencies
Value :

Attribute : Derived From
Value :

Attribute : Design Spec
Value :

Attribute : Distribution Statement
Value : S. Bremer, NIST, Bldg 101 Room A337, Gaithersburg, MD 20899
(FTS 879-2845 or 301-975-2845)

Attribute : Encrypted
Value :

Attribute : Feedback
Value :

Attribute : Feedback From
Value :

Attribute : Feedback From Organization
Value :

Attribute : Format
Value :

Attribute : History
Value :

Attribute : Id
Value : NIST_000011

Attribute : Implementation Citation
Value :

Attribute : Installation Info
Value : 840NOS

Attribute : Interfaces
Value :

Attribute : Is Composed of
Value :

Attribute : Is Part Of
Value :

Attribute : Kind
Value :

Attribute : Limitations and Constraints
Value : Some Conversion Required

Attribute : Makefile
Value :

Attribute : Metrics
Value :

Attribute : National Language
Value :

Attribute : Number Of Extractions
Value :

Attribute : Other
Value :

Attribute : Parent
Value :

Attribute : Problem Report
Value :

Attribute : Publication Date
Value :

Attribute : **Publisher**
Value : **NIST, Statistical Engineering Division, Gaithersburg, MD 20899 (J.J. Filliben)**

Attribute : **Read Me**
Value :

Attribute : **Reference**
Value : **J.J. Filliben. User's Guide to Datapac (version 77.5).**
NBS, Gaithersburg, MD, 1977.

Attribute : Reference Number
Value :

Attribute : Requirements Spec
Value :

Attribute : Restrictions Apply
Value :

Attribute : Review
Value :

Attribute : Reviewer
Value :

Attribute : Security Restriction
Value :

Attribute : See Also
Value :

Attribute : Size
Value :

Attribute : Source Code
Value : ATTACH,DPACMOD/UN=CAMLIB,NA.\$MODIFY(P=D PACMOD,...).

Attribute : Submission Date
Value :

Attribute : Supplemental Information
Value :

Attribute : Support
Value : Full Support

Attribute : Support Contact
Value :

Attribute : Support Organization
Value :

Attribute : Test Suite

```
Value     : ATTACH,DATAPAC/UN=CAMLIB,NA.\$LIBRARY,DA TAPAC.
--------------------------------------------------------------
Attribute : Title
Value     : DATAPAC
--------------------------------------------------------------
Attribute : Type
Value     : PACKAGE
--------------------------------------------------------------
Attribute : User Guide
Value     :
--------------------------------------------------------------
Attribute : Version
Value     : 870401
--------------------------------------------------------------
Attribute : Warranties
Value     :
--------------------------------------------------------------
Attribute : Was Created By
Value     :
--------------------------------------------------------------

            - THE END -
```

When the call "marctool 3 NIST_000011 <outputfile> <statusfile>" is made, the output is the following Readable MARC Record:

```
TAG   FIELD NAME          CONTENTS
===   ==========          ========
001   ASSET-ID            NIST_000011
204 a TITLE               DATAPAC
260 b PUBLISHER           NIST, Statistical Engineering Division,
                          Gaithersburg, MD 20899 (J.J. Filliben)
520 a ABSTRACT            A Fortran subroutine library for probability
                          distribution, density, percent point,
                          and sparsity function evaluation;
                          random number generation; line-printer
                          plotting - histograms, scatter diagrams,
                          probability plots; data manipulation;
                          general statistical analysis; time
                          series analysis; polynomial regression;
                          ANOVA. (Approximately 170 subroutines.)

650 a SUBJECT-CLASS       Statistics, Probability: Behavioral
                          and Social Sciences; Measurement, Statistics
                          and Evaluation; Statistics; Computer,
                          Mathematical, and Physical Sciences;
                          Applied Mathematics; Computer Science;
```

	Computational Methods; Numerical Analysis; Mathematics;
901 b TYPE	Subprogram Library
901 a TYPE	PACKAGE
912 a COMP-LANG	Fortran
912 b COMP-OPSYS	NOS
912 c COMP-COMPILER	FTN5
920 a DISTRIB-STMT	S. Bremer, NIST, Bldg 101 Room A337, Gaithersburg, MD 20899 (FTS 879-2845 or 301-975-2845)
926 a INSTALL-INFO	840NOS
931 a LIMIT-CONSTR	Some Conversion Required
939 a REFERENCE	J.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.
939 a REFERENCE	J.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.
939 a REFERENCE	J.J. Filliben. User's Guide to Datapac (version 77.5). NBS, Gaithersburg, MD, 1977.
947 a SOURCE-CODE	ATTACH,DPACMOD/UN =CAMLIB,NA.\$MODIFY(P=D PACMOD,...).
949 a SUPPORT	Full Support
950 a TEST-SUITES	ATTACH,DATAPAC/ UN=CAMLIB,NA.\$LIBRARY,DA TAPAC.
952 b VERSION	1986
952 a VERSION	870401
999 b AP#	8
999 a COMPUTER-NO	6
999 v VERSION	2
999 d COMPUTER-LOC	CYBER 840, Consolidated Scientific Computing System, Boulder
999 e COMPUTER-CONTACT	Katherine Pagoaga-320-5104, Linda Lindgren (CMLIB)-320-5149
999 i ENTRY-ID	GAMS=8=6=2

When the call " marctool 4 NIST_000011 <outputfile> <statusfile> 520" is made, the output is the following single field, which corresponds to tag 520, of the MARC record in readable form:

TAG	FIELD NAME	CONTENTS
520 a	ABSTRACT	A Fortran subroutine library for probability distribution, density, percent point, and sparsity function evaluation; random number generation; line-printer plotting - histograms, scatter diagrams, probability plots; data manipulation; general statistical analysis; time series analysis; polynomial regression; ANOVA. (Approximately 170 subroutines.)

One additional example of a MARC record and the Readable MARC to explain its contents is listed below.

A MARC Record from the StatLib Repository

00429n 2200085 45000010011000002040007000115200055000186500258000739010012003 31#CMU_000019# =aS # =aS functions, device drivers and related software. # =aStatistics, Probability; Behavioral and Social Sciences; Measurement, Statistics and Evaluation; Statistics; Computer, Mathematical, and Physical Sciences; Applied Mathematics; Computer Science; Computational Methods; Numerical Analysis; Mathematics; # =apackage#X

The Same Record in Readable MARC Format

TAG	FIELD NAME	CONTENTS
001	ASSET-ID	CMU_000019
204 a	TITLE	S
520 a	ABSTRACT	S functions, device drivers and related software.
650 a	SUBJECT-CLASS	Statistics, Probability; Behavioral and Social Sciences; Measurements, Statistics and Evaluation; Statistics; Computer, Mathematical, and Physical Sciences; Applied Mathematics; Computer Science; Computational Methods; Numerical Analysis; Mathematics;
901 a	TYPE	package

References

- [Ag88] W.W. Agresti and F.E.McGarry, The Minnowbrook Workshop on Software Reuse: A Summary Report from *Tutorial: Software Reuse: Emerging Technology*, pp. 33-40, Editor W.Tracz, IEEE Computer Society Press, Los Alamitos, CA, 1990.

- [As91] Asset Library Open Architecture Framework (ALOAF) Version 0.5. DRAFT, STARS, August 1991.

- [Av89] Nordwall, B.D., Software Delays NORAD Upgrade, Increases Costs by $207 Million, *Aviation Week & Space Technology*, pp. 24-25, May 22, 1989.

- [Av90] Hughes,D., Computer Experts Discuss Merits of Defense Dept. Software Plan, *Aviation Week & Space Technology*, pp.65, April 16, 1990.

- [B81] B.W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981.

- [BCC92] V.R.Basili, G.Caldiera & G.Cantone, A Reference Architecture for the Component Factory, *ACM Transactions on Software Eng & Methodology*, Vol.1, No.1, pp.53-80, January 1992.

- [Bo85] R. Boisvert, S.E. Howe, D.K. Kahaner, GAMS: A Framework for the Management of Science Software, *ACM Transactions on Mathematical Software*, Vol. 11, No.4, pp. 313-355, December 1985.

- [Bo89] R. Boisvert, S.E. Howe, J.L. Springmann, *Internal Structure of the Guide to Available Mathematical Software*, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, Md., March 1989.

- [Bo90] R. Boisvert, S.E. Howe, D.K. Kahaner, *The Guide to Available Mathematical Software Problem Classification System,* U.S. Department of Commerce, National Institute of Standards and Technology, November 1990.

- [Bol89] C. Boldyreff, Reuse, Software Concepts, Descriptive Methods and the Practitioner Project, *Software Engineering Notes*, Vol.14, no.2, pp.25-31, April 1989.

- [Br90] J.C.Browne, T.Lee & J.Werth, Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment, *IEEE Transactions on Software Engineering,* Vol.16, No.2, pp. 111-120, February 1990.

- [Bu87] B.A.Burton et al, The Reusable Software Library, *IEEE Software*, pp. 25-32, July 1987 and from *Tutorial: Software Reuse: Emerging Technology*, pp. 33-40, Editor W.Tracz, IEEE Computer Society Press, Los Alamitos, CA, 1990.

- [By91] D. Byrne, *MARC Manual -- Understanding and Using MARC Records*, Libraries Unlimited, Inc., Englewood, Colorodo, 1991.

- [Ch84] T.E.Cheatham Jr., Reusability Through Program Transformations, *IEEE Transactions on Software Engineering*, Vol.SE-10, No.5, pp.589-594, September 1984.

- [Cl88] J. Craig Cleveland, Building Application Generators, *IEEE Software*, pp. 25-33, July 1988.

- [CLP84] T.T.Cheng, E.D.Lock & N.S.Prywes, Use of Very High Level Languages and Program Generation by Management Professionals, *IEEE Transactions on Software Engineering*, Vol.SE-10, No.5, pp.552-563, September 1984.

- [Cu91] M.A.Cusumano, *Japan's Software Factories*, Oxford University Press, Inc., New York, New York, 1991.

- [DBSB91] P.Devanbu, R.J.Brachman, P.G.Selfridge & B.W.Ballard, LaSSIE: A Knowledge-Based Software Information System, *Communications of the ACM*, Vol.34, No.5, pp.35-49, May 1991.

- [HM84] E.Horowitz, J.B.Munson, An Expansive View of Reusable Software, *IEEE Transactions on Software Engineering*, Vol.SE-10, No.5, pp.477-487, September 1984.

- [I90] IBM STARS Repository Guidebook, Prepared by IBM Systems Integration Division, Gaithersburg, Md., April 1990.

- [J92] B.E. Jacobs, An HPCC Software Exchange-Architectural Elements and Metrics, Working Draft, GSFC/NASA, September 1992.

- [J92a] B.E. Jacobs, Locating Software Using the HPCC Logical Library, Working Draft, GSFC/NASA, November 1992.

- [J92b] B.E. Jacobs, Managing a Logical Library System, Working Draft, GSFC/NASA, November 1992.

- [J92c] B.E. Jacobs, Submission Guidelines for the HPCC Software Exchange, Working Draft, GSFC/NASA, November 1992.

- [J92d] B.E. Jacobs, Building Repository Databases Using Books, Working Draft, GSFC/NASA, November 1992.

- [J93a] B.E. Jacobs, Building Clients and Services for Software Repository Databases, Working Draft, GSFC/NASA, January 1993.

- [J93b] B.E. Jacobs, Locating Software Using the HPCC Logical Library, Working Draft, GSFC/NASA, April 1993.

- [J93c] B.E. Jacobs, Building Software Repository Databases, Working Draft, GSFC/NASA, April 1993.

- [J93d] B.E. Jacobs, Implementation Plan for the HPCC Software Exchange-Prototype System, Working Draft, GSFC/NASA, September 1993.

- [Jo84] T.C.Jones, Reusability in Programming: A Survey of the State of the Art, IEEE Transactions on Software Engineering, Vol. SE-10, No.5, pp.488-493, September 1984.

- [K92] C.W.Krueger, Software Reuse, *ACM Computing Surveys* Vol.24, No.2, 131-184, New York, June 1992.

- [L86] R.K.Logan, *The Alphabet Effect,* William Morrow and Company, Inc., New York, 1986.

- [LG84] R.G.Lanergan, C.A.Grasso, Software Engineering with Reusable Designs and Code, *IEEE Transactions on Software Engineering,* Vol.SE-10, No.5, pp.498-501, September 1984.

- [M80] Y.Matsumoto et al, SWB System: A Software Factory, Software Engineering Environments, pp. 305-318, North-Holland Publishing Co., New York, 1980.

- [M84] Y. Matsumoto, Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels, *IEEE Transactions on Software Engineering*, Vol.SE-10, No.5, pp.502-513, September 1984.

- [MBK91] Y.S.Maarek, D.M.Berry & G.E.Kaiser, An Information Retrieval Approach for Automatically Constructing Software Libraries, *IEEE Transactions on Software Engineering*, Vol.17, No.8, pp. 800-813, August 1991.

- [Me93] S. Merritt, DISA/CIM Software Reuse Program - Framework for Certification of Reusable Software Components (Version 1.0), SoftTech, Inc., February 1993.

- [MMH90] M. Moroh, E. Malka & K. Harrow, DBMS's Interface Model: Interfacing between IDMS and DAVID, Computer Science Technical Report #TR-3-90, Brooklyn College, Department of Computer and Information Science, Brooklyn, New York, July 1990.

- [MMH91] M. Moroh, E. Malka & K. Harrow, DBMS's Interface Second Model: An inner interface between IDMS or Focus, and DAVID, Computer Science Technical Report #TR-1-91, Brooklyn College, Department of Computer and Information Science, Brooklyn, New York, March 1991.

- [N91] F.Nishida, S.Takamatsu, Y.Fujita & T.Tani, Semi-Automatic Program Construction from Specifications Using Library Modules, *IEEE Transactions on Software Engineering,* Vol.17, No.9, pp. 853-871, September 1991.

- [Ne84] J.M.Neighbors, The Draco Approach to Constructing Software from Reusable Components, *IEEE Transactions on Software Engineering,* Vol.SE-10, No.5, pp.564-574, September 1984.

- [Ni91a] Nieder, A., RAPID (Reusable Ada Products for Information Systems Development) Implementing a Comprehensive Reuse Program, US Army Software Development Center-Washington, Fort Belvoir, Va., 1991.

- [Ni91b] Nieder, A., RAPID (Reusable Ada Products for Information Systems Development) Qualifying Software Components: Reusability Metrics, US Army Software Development Center-Washington, Fort Belvoir, Va., 1991.

- [Pe91] A.S.Peterson, Coming to Terms with Software Reuse Terminology: a Model-Based Approach, *ACM Sigsoft Software Engineering Notes*, pp.45-51, vol.16, no.2, April 1991.

- [PF87] R. Prieto-Diaz, P. Freeman, Classifying Software for Reusability, *IEEE Software*, pp.6-16, January 1987.

- [PJP91] E.W.Pugh, L.R.Johnson and J.H.Palmer, *IBM's 360 and Early 370 Systems*, The MIT Press, Cambridge, MA, 1991.

- [Pr91] R. Prieto-Diaz, Implementing Faceted Classification for Software Reuse, *Communications of the ACM*, pp.89-97, Vol.34, No.5, May 1991.

- [Pr91b] R. Prieto-Diaz, Making Software Reuse Work: An Implementation Model, *ACM Sigsoft Software Engineering Notes*, Vol.16, No.3, pp.61-68, July 1991.

- [R91] M. Raugh, Examples of Service Components for a National Software Exchange: A Position Statement, Reasearch Institute for Advanced Computer Science NASA, Ames Research Center, RIACS Technical Memorandum 91-A, April 8, 1991.

- [S90] Final Programmer's Guide for the Rapid Center Library System, Submitted to: US Army Engineering Command, Ft.Belvoir, Va., Prepared by: SofTech Inc., Waltham, Ma., August 1990.

- [Sk86] Panel Discussion on Comparison of Evaluation Methods of Cost Assessment in *Computing, from Software System Design Methods*, Edited by J.K. Skwirzynski, NATO ASI Series, Vol. F22, Springer-Verlag Berlin Heidelberg, 1986.

- [T87] W.Tracz, Confessions of a Used Program Salesman from *Tutorial: Software Reuse: Emerging Technology*, Editor W.Tracz, IEEE Computer Society Press, pp. 92-95, Los Alamitos, CA, 1990.

- [Ty86] Tyler, T., Reusable Software Library (RSL) Manual, Ford Aerospace Corp., Houston Texas, December 1986.

- [We88] Software User's Manual for the Library System of Reusable Ada Avionics Software Packages, Prepared for: Avionics Laboratory, by: Westinghouse Electric Corp., contract no. F33615-90-C-1432, December 14, 1990.

- [WJ90] R.J. Wirfs-Brock & R.E.Johnson, Surveying Current Research in Object-Oriented Design, *Communications of the ACM*, Vol.33, No.9, pp.104-124, September 1990.